Computation-in-Memory
From Circuits to Compilers

Yu, J.

DOI
10.4233/uuid:9f2a640e-0f19-4d4d-9feb-e27e3e963fcb

Publication date
2021

Document Version
Final published version

Citation (APA)
Yu, J. (2021). *Computation-in-Memory: From Circuits to Compilers*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:9f2a640e-0f19-4d4d-9feb-e27e3e963fcb

Computation-in-Memory

From Circuits to Compilers

Computation-in-Memory

From Circuits to Compilers

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, prof.dr.ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates
to be defended publicly on Friday, 5 February 2021 at 10:00 o'clock

by

Jintao Yu

Master of Engineering in Computer Science and Technology,
PLA Information Engineering University, China,
born in Heilongjiang, China.

This dissertation has been approved by the

Promotor: Prof. dr. ir. S. Hamdioui
Copromotor: Dr. ir. M. Taouil

Composition of the doctoral committee:

Rector Magnificus,	chairman
Prof. dr. ir. S. Hamdioui,	Delft University of Technology, promoter
Dr. ir. M. Taouil,	Delft University of Technology, copromoter
Independent members:	
Prof. dr. ir. P.F.A. Van Mieghem	Delft University of Technology
Prof. dr.-ing. D. Fey	Friedrich-Alexander-University of Erlangen-Nürnberg, Germany
Prof. dr. G.C. Sirakoulis	Democritus University of Thrace, Greece
Prof. dr. A. Kumar	Dresden University of Technology, Germany
Dr. ir. J.S.S.M. Wong	Delft University of Technology
Reserved members:	
Prof. dr. ir. A.J. van der Veen	Delft University of Technology

Keywords: In-memory computing, memristive devices, automata processing

ISBN 978-94-6384-196-2

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

To my parents

Contents

Summary

Memristive devices are promising candidates as a complement to CMOS devices. These devices come with several advantages such as non-volatility, high density, good scalability, and CMOS compatibility. They enable in-memory computing paradigms since they can be used for both storing and computing. However, building in-memory computing systems using memristive devices is still in an early research stage. Therefore, challenges still exist with respect to the development of devices, circuits, architectures, design automation, and applications.

This thesis focuses on developing memristive device-based circuits, their usage in in-memory computing architectures, and design automation methodologies to create or use such circuits.

Circuit Level – We propose two logical operation schemes based on memristive devices. The first one uses resistive sensing to perform logical operations. It modifies the sense amplifier in such a way that it can compare the overall current with references and output the logical operation result. During sensing, the resistance of memristive devices remains unchanged. Therefore, endurance and lifetime are not reduced. This scheme provides a solution for maintaining a relatively long lifetime in logic operations for memristive devices that have low endurance. The second scheme is the enhanced version of the first one. It uses two different sensing paths for AND and OR operations. In this way, the correctness of logic operations can be guaranteed even if large resistance variation exists in memristive devices.

Architecture Level – We present three in-memory computing architectures based on memristive devices. The first one is a heterogeneous architecture containing an accelerator for vector bit-wise logical operations and a CPU. The accelerator communicates with the CPU or accesses the external memory directly. The second one is to accelerate automata processing. In this architecture, memristive memory arrays store configuration information and conduct computation as well. This architecture outperforms similar ones that are built with conventional memory technologies. The third one is an improved version of the second one. It breaks the routing network into multiple pipeline stages, each processing a different input sequence. In this way, the architecture achieves a higher throughput with a negligible area overhead.

Design Automation – A synthesis flow for computation-in-memory architectures and a compiler for automata processors are presented. The synthesis flow is proposed based on the concept of skeletons, which relates an algorithmic structure to a pre-defined solution template. This solution template contains scheduling, placement, and routing information needed for the hardware generation. After the user rewrites the algorithm using skeletons, the tool generates the desired circuit by instantiating the solution template. The automata processor compiler generates configuration bits according to the input automata. It uses multiple strategies to transform given automata, so that constraint conflicts can be resolved automatically. It also optimizes the mapping for storage utilization.

Samenvatting

Geheugenweerstanden zijn veelbelovende kandidaten als aanvulling op CMOS-transistors. Deze geheugenweerstanden hebben verschillende voordelen, zoals niet-vluchtigheid, hoge dichtheid, goede schaalbaarheid en CMOS-compatibiliteit. Daarnaast kunnen ze gebruikt worden in gegevensverwerking-in-geheugenarchitecturen, omdat ze zowel als opslagmedium als computer gebruikt kunnen worden. De ontwikkeling van gegevensverwerking-in-geheugenarchitecturen met behulp van geheugenweerstanden bevindt zich echter nog in een vroege onderzoeksfase. Daardoor zijn er nog steeds uitdagingen met betrekking tot de ontwikkeling van geheugenweerstanden, circuits, architecturen, ontwerpautomatisering en toepassingen die opgelost moeten worden.

Dit proefschrift richt zich op het ontwikkelen van geheugenweerstandcircuits, het gebruik hiervan in gegevensverwerking-in-geheugenarchitecturen, en ontwerpautomatiseringsmethodologieën om dergelijke circuits te creëren of te gebruiken.

Circuitniveau - We presenteren twee ontwerpen van binaire logica op basis van geheugenweerstanden. Het eerste ontwerp voert logische bewerkingen uit middels een weerstandsmeting. Dit wordt bereikt door de detectieversterker zodanig aan te passen dat deze de totale stroom kan vergelijken met meerdere referenties en vervolgens het resultaat van de logische operatie kan uitvoeren. Tijdens de weerstandsmeting verandert de weerstand van de geheugenweerstanden niet. Hierdoor worden het uithoudingsvermogen en de levensduur niet verminderd. Dit ontwerp biedt dus een oplossing om de levensduur van geheugenweerstanden met een laag uithoudingsvermogen te maximaliseren bij het uitvoeren van logische bewerkingen. Het tweede ontwerp is een verbeterde versie van het eerste. Het gebruikt twee verschillende detectiepaden voor logische EN- en OF-bewerkingen. Op deze manier kan de juiste uitkomst van de logische bewerkingen worden gegarandeerd, zelfs als er grote weerstandsvariaties bestaan in de geheugenweerstanden.

Architectuurniveau - We presenteren drie gegevensverwerking-in-geheugenarchitecturen op basis van geheugenweerstanden. De eerste is een heterogene architectuur bestaande uit een accelerator voor binaire logische bewerkingen en een processor. De accelerator communiceert met de processor of heeft rechtstreeks toegang tot het externe geheugen. De tweede architectuur versnelt de berekening van eindigetoestandsautomaten. In deze architectuur slaat het geheugenweerstandsgeheugen de configuratie-informatie op en voert het ook de berekeningen erop uit. Deze architectuur presteert beter dan vergelijkbare architecturen die zijn gebaseerd op conventionele geheugentechnologieën. De derde architectuur is een verbeterde versie van de tweede. Deze verdeelt het verdeelnetwerk in meerdere

pijplijnfasen, die elk een andere invoer verwerken. Hierdoor bereikt deze architectuur een hogere datadoorvoer ten koste van een verwaarloosbare toename in chipoppervlakte.

Ontwerpautomatisering – We presenteren een syntheseproces voor gegevensverwerking-in-geheugenarchitecturen en een compiler voor eindigetoestandsautomaatprocessors. Het syntheseproces wordt gepresenteerd op basis van skeletsjablonen, die een algoritmische structuur relateren aan vooraf gedefinieerde oplossingssjabloon. Dit oplossingssjabloon bevat informatie over de planning, plaatsing en netwerkstructuur die nodig is om de hardware te genereren. Nadat de gebruiker het algoritme middels de skeletsjablonen heeft herschreven, genereert de tool het gewenste circuit door het oplossingssjabloon te instantiëren. De compiler voor de eindigetoestandsautomaatprocessor genereert configuratiebits op basis van de ingevoerde eindigetoestandsautomaten. Hij gebruikt verschillende strategieën om bepaalde eindigetoestandsautomaten te transformeren, zodat conflicten automatisch kunnen worden opgelost. Tevens optimaliseert de compiler de toewijzing voor het gebruik als een geheugen.

Acknowledgements

The years in Delft are the highlights of my life. I will take this opportunity to express my gratitude to the people who helped me along this incredible journey.

First of all, I would like to acknowledge my promoter Prof. dr. ir. Said Hamdioui. He sets an example of a diligent and rigorous researcher to all of his students. He insisted on having weekly meetings with us despite his fully packed agenda. I have to leave the Netherlands two years ago, but he still cared about me as much as before. Without his guidance, this dissertation would be never finished. I also want to thank Dr. ir. Mottaqiallah Taouil, not only my copromoter but also a colleague and friend. He corrected my papers and helped me get through the difficulties in pursuing the PhD degree. Special thanks go to Prof. dr. ir. Koen Bertels for admitting me into the Computer Engineering group and hosting various social events. I thank Dr. ir. Stephen Wong for working together and being a defense committee member. I have received guidance from Răzvan Nane and Imran Ashraf as well. Thank you all.

Secondly, I shall thank the memristor team in our group, i.e., Lei, Anh, Muath, and Adib. You all have contributions to this dissertation. Tom and Uljana also joined the team shortly. We share ideas, help each other with writings and presentations, and travel together for conferences and meetings. I am lucky to be able to work with all of you.

Thanks to previous office mates including Innocent, Abdulqader, Dániel, and Houssem. We had meaningful discussions and funny chats, which altogether create a productive and joyful environment. It is nice to have you around.

I am grateful to the colleagues on the ninth floor, Lizhou, Daniël, Moritz, Troya, Haji, Guilherme, and Abdullah. We usually have lunch together, which is a sweet, relaxed time on a busy day. In particular, a big thanks to Moritz for translating the summary of this dissertation to Dutch. Then, I extend my thanks to the colleagues on the tenth floor, Jian, Xiang, Joost, Shanshan, Yande, Leon, Lingling, Nicoleta, Savvas, He, Baozhou, and Mahroo. We present our researches during lunch colloquium, and have fun in group barbecue, beer nights, Christmas parties, and other lovely events.

The facilitation from group staff is crucial for my work as well. I thank Erik for maintaining the servers, Lidwina, Joyce, and other sectaries for all kinds of administration work.

I have the warmest memory with the Chinese community in TU Delft. I enjoyed the gatherings with Yue, Jiapeng, Fanyu, Tiantian, HaoHua, Hai, Shuai, Minghe, Qiang, Zhenji, Yu, Wenjie, and many other friends. I wish for your bright futures!

Last but not least, I appreciate the support of my parents. They always encourage me with love and understanding. Therefore, I dedicate this dissertation to them.

1 Introduction

Data-intensive applications are becoming more important and demand more computing power. However, conventional computing architectures and the CMOS technology that they are based on face various challenges such as the bottleneck between CPUs and the memory. In-memory computing paradigms can alleviate such problems by placing computing cores inside the memory. Memristive devices, which support both storage and computing, are promising enablers of the in-memory computing paradigm. We investigate various aspects of building systems based on in-memory computing with memristive devices, including the circuit level, the architecture level, and their design automation to explore the potential of such systems.

This chapter introduces first the motivation behind building in-memory computing architectures using memristive devices. Thereafter, it presents their opportunities and challenges. Subsequently, it describes the research topics of this thesis briefly followed by the main contributions. Finally, it discusses the organization of the remaining chapters.

1.1. Introduction to Memristive Devices

In this section, we first present the motivation of investigating memristive devices in Section 1.1.1. Thereafter, Section 1.1.2 gives an introduction on memristive devices. Subsequently, Section 1.1.3 and Section 1.1.4 summarize the usage of memristive device for logic and memory, respectively.

1.1.1. Motivation

Data-intensive applications have gained more importance in varies domains such as health-care [1], artificial intelligence [2], and economics [3]. They demand more computing power, larger storage, and higher energy efficiency [4]. This motivates the scientific community to innovate the current technologies and architectures to meet these demands.

In the last decades, the advancement of computing systems was mainly driven by CMOS scaling [5]. However, this trend will not sustain forever due to the following three walls [6]:

- **The reliability wall:** As the transistor dimensions are reduced towards their physical limit, CMOS transistors will suffer from a reduced lifetime and increased failure rate [7].
- **The leakage wall:** As the threshold voltage decreases with scaling, the relative sub-threshold leakage increases [8]. Since CMOS is volatile, the static leakage may become dominant and exceed dynamic power [9].
- **The cost wall:** Economical benefits brought by technology scaling is reduced because of the increased design complexity and test difficulty [10].

In addition, current computer architectures also face three walls [11]:

- **The memory wall:** The data processing speed of CPUs is greater than the data bandwidth provided by the memory, as shown in Figure 1.1. As the numbers of cores or processing elements increase over time, the memory bandwidth deficit increases.
- **The power wall:** Due to the constraints of cooling, there is a limit to the operating frequency of microprocessors. This limit has been reached already around 2005 as shown in Figure 1.2. As a consequence, the performance of a single thread saturated. Note that the energy consumption is critical for devices that are powered by batteries, such as laptop computers, tablet, and smartphones. These devices are becoming more important in daily life.
- **The instruction-level parallelism (ILP) wall:** Since around 2005, the main way of improving the performance has been realized by increasing the number of logical cores as shown in Figure 1.2. With more and more cores, the difficulty of extracting sufficient parallelism from the application has significantly increased.

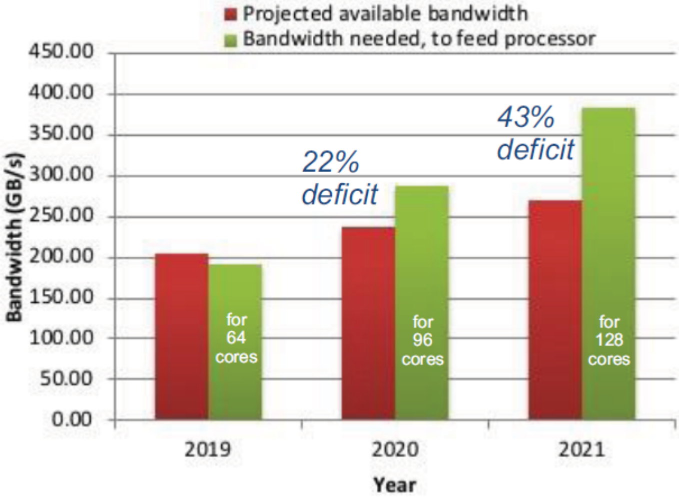
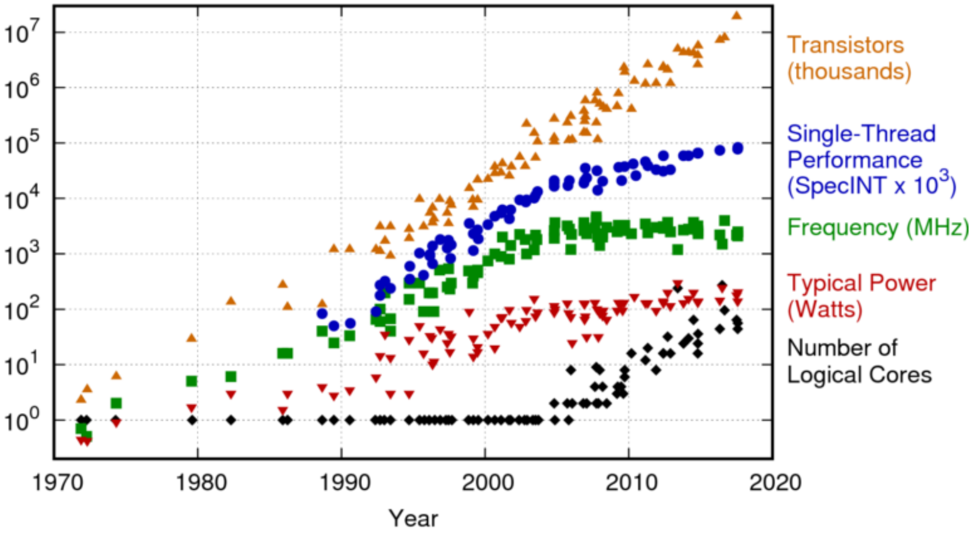


Figure 1.1: Memory bandwidth deficit for feeding processors [12].



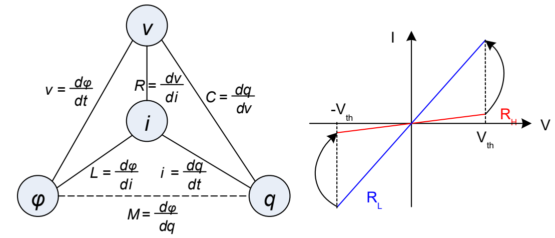
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Figure 1.2: Microprocessor trend data [13, 14].

These walls have decelerated the advancements of conventional computing systems. Therefore, alternative computing paradigms and technologies are explored [15–18] to alleviate the above problems. Among these paradigms, *in-memory computing* is promising as it may overcome the memory wall [19, 20]. It refers to a computing paradigm where information is stored and processed at the same physical location, e.g, in the memory [21]. Memristive devices [22] are promising candidates to build in-memory computing architectures because they support both storage and logical operations [23]. In addition, they have advantages such as non-volatility, high density, good scalability, and CMOS compatibility [24, 25]. Therefore, in-memory computing based on memristive devices has a huge potential and is worth to be investigated thoroughly.

1.1.2. Memristive Devices

A memristive device, or a *memristor* in short, is the fourth fundamental two-terminal element next to the resistor, capacitor, and inductor. Its existence is predicted by Leon Chua in 1970s [26]. He noticed that the relationship between flux and charge was missing, which is indicated by the dashed line in Figure 1.3(a). This relationship can be described using *memristance* M, i.e., $M = d\phi/dq$. When M is a function of the charge q, the memristor yields special properties that cannot be simulated by the combination of other fundamental elements. A crucial characteristic of a memristor is the 'pinched hysteresis loop' current-voltage curve as illustrated in Figure 1.3(b). It presents a memristive device that has two stable states: high R_{H} and low R_{L} resistive states. The device switches from a resistive state to another when the voltage across the memristive device is greater than the absolute value of its threshold voltage V_{th}. Therefore, the internal state of the device is decided by the external voltage history.



(a) The four fundamental elements (b) Pinched hysteresis loop

Figure 1.3: Basic of a memristor

After a silent period of more than 30 years, memristive devices became renowned in 2008 when the first physical memristor device was confirmed by HP Labs [27]. They built a metal-insulator-metal device using a titanium oxide as an insulator sandwiched by two metal electrodes. They successfully identified the memristive behavior over its two-terminal node as described by Leon Chua. When applied with different voltages, the device tunes its resistance by controlling positive charged oxygen vacancies in the insulator layer. The research in this field has rapidly grown since then and many non-volatile memories (NVMs) based on different types of materials, such as $\mathrm{HfO_x}$, $\mathrm{TaO_x}$, $\mathrm{SiO_x}$, have emerged [28, 29]. These NVM elements exhibit many properties of a memristor (e.g., device resistance changes under external stimulation). These cells have two or more stable resistance states. They switch from one state to another when a voltage or current is applied that exceeds the threshold of the device. Although there are differences between the devices (e.g., in some devices the resistance changes abruptly instead of continuously), it is common to refer to them as memristive devices.

1.1.3. Memristive Devices for Logic

In this section, we first classify existing memristor-based logic design styles. Thereafter, the working principle of a logic design style is presented in more details. Finally, we compare these design styles qualitatively.

Many logic design styles have been proposed [30–37] based on memristive devices. They can be divided into several classes using the following criteria:

- **Input Data Representation**: the input data is represented by either a voltage or resistance.
- **Output Data Representation**: the output data is represented by either a voltage or resistance.
- **Processing Elements**: the data is *processed* either based on memristors only or by using a hybrid CMOS/memristor combination. Note that the *control* of the memristors is always done using CMOS circuits.

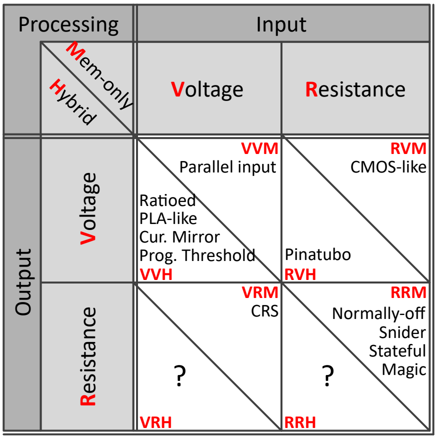


Figure 1.4: Classification of memristor-based logic design styles [38].

The classification result is illustrated in Figure 1.4, including eight classes in total. We name each class according to the input and output representation signals, and the processing element. For instance, Pinatubo [39] is located in the RVH class where R indicates the input data representation, V the output data representation and H hybrid CMOS/memristor processing. It is indicated by the classification result that current logic designs fit in six defined classes, and that two classes have not been explored yet.

- **VVH**: This class includes Memristor Ratioed Logic [30], PLA-like [31], Current Mirror based Threshold Logic [32], and Programmable Threshold Logic [33]. Both input and output data are represented using voltages. CMOS gates, such as inverters [30–32] and D Flip-Flop [33], perform as a threshold function. These logic styles use memristors as either configuration switches [30, 31] or input weights [32, 33].

- **VVM**: A Parallel Input Processing Logic [40] belongs to this class. This logic style uses voltages to represent input and output data. Its processing elements are memristors with asymmetric voltage thresholds that are connected in various ways. Note that it still needs CMOS circuitry for resetting the memristors, summing up the inputs, and reading out the result.

- **RVH**: Pinatubo [39] is the work published in this class. It uses a resistance to represent the input data and a voltage to represent the output data. It performs logical operations by modifying memory read operations.

- **RVM**: This class contains only CMOS-like Logic [34], which represents the input data using a resistance and the output data using a voltage. The MOSFETs in the pull-up and -down network of the conventional CMOS logic are replaced with memristors.

- **VRM**: Complementary Resistive Switching (CRS) Logic [35] is the only published work in this class. The input data is represented using a voltage and the output data is represented using a resistance. The logical operations in CRS logic are performed by modified memory write operations. In another work, the CRS logic gates are extended with other Boolean logic gates to decrease the execution steps [41].

- **RRM**: This class includes Snider [36], Stateful Logic [37], Normally-off Logic [42]. They represent both the input and output data using resistances. Memristors are used as voltage dividers to perform logical operations, which conditionally switch the output memristors. Stateful Logic is extended to support more types of logical operations such as AND-IMP and OR-IMP by Lehtonen *et al.* [43]. Snider Logic is extended to support more types of logical operations such as AND and OR by Kvatinsky *et al.* [44] and Xie *et al.* [45]. Normally-off Logic differs from the others by connecting the memristors sequentially instead of in parallel.

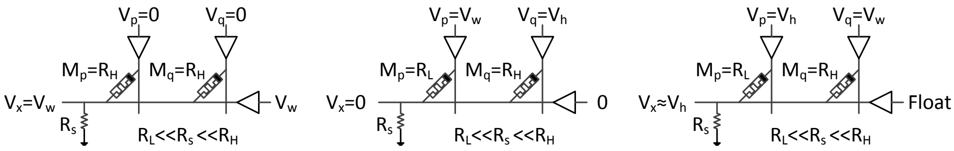
We use Stateful Logic [37] as an example to illustrate the methodologies of implementing logical operations with memristive devices. Stateful Logic supports material implication (IMP) as a primitive logical operation. The IMP operation is denoted by

$$\text{IMP: } q' = p \rightarrow q = \bar{p} + q \tag{1.1}$$

where p and q are inputs while q' is the output. Logic 0 and 1 are represented by R_H and R_L, respectively, for both the inputs and outputs. There are two memristors (i.e., $\mathrm{M_p}$ and $\mathrm{M_q}$) and a resistor R_s ($R_L \ll R_s \ll R_H$) in an IMP gate. $\mathrm{M_p}$ and $\mathrm{M_q}$ store input p and q, respectively, and the output q' is stored in $\mathrm{M_q}$ after the operation. Control voltages V_h and V_w are applied to $\mathrm{M_p}$ and $\mathrm{M_q}$, respectively, to perform the IMP operation. The control voltages typically satisfy the following relationship:

$$0 < V_h = \frac{V_w}{2} < V_{\mathrm{th}} < V_w < 2V_{\mathrm{th}} \tag{1.2}$$

We use the IMP gate with inputs $p = 1$ and $q = 0$ as an example to illustrate the working principle of Stateful Logic. The operation is illustrated in Figure 1.5, which consists of three steps. First, voltages $V_p = V_q = GND$ and $V_x = V_w$ are applied to all the memristors to reset them to R_H (see Figure 1.5(a)). Then, voltages $V_p = V_w$, $V_q = V_h$ and $V_x = 0$ are applied to $\mathrm{M_p}$ to program it to R_L (p is logic 1) (see Figure 1.5(b)). V_h is applied to $\mathrm{M_q}$ to prevent it from undesired switching. Finally, the IMP gate is evaluated by applying voltages $V_p = V_h$, $V_q = V_w$ to $\mathrm{M_p}$ and $\mathrm{M_q}$, respectively, and keeping the row floating (see Figure 1.5(c)). As a result, $V_x{\approx}V_h$ ($R_L{\ll}R_s{\ll}R_H$) and the voltage across $\mathrm{M_q}$ is $V_q - V_x \approx V_w - V_h < V_{th}$. Therefore, $\mathrm{M_q}$ stays in R_H. The output of the IMP gate is interpreted as logic 0. We refer the reader to [37, 43, 44] for more details and the latest progress.



(a) RESET all memristors (b) Program input memristors (c) Evaluate the output memristor

Figure 1.5: The IMP operation in Stateful Logic.

Finally, we use the following metrics to evaluate existing memristor logic design styles qualitatively.

- **Array Compatibility**: whether the logic style is compatible with normal 1R and/or 1T1R memory arrays.
- **CMOS Controller Requirement**: whether the logic style needs a CMOS circuit for control.
- **Nonvolatility**: whether the logic style can store the data when it is powered off.
- **Area**: area-efficiency of the logic style to perform logical operations.
- **Speed**: time-efficiency of the logic style to perform operations.
- **Energy**: energy-efficiency of the logic style to perform logical operations.
- **Scalability**: how well the logic style can be scaled to implement more complex circuits.

Table 1.1: Comparison Between Existing Logic Styles

Style	Class	Array	Control	NV	Speed	Area	Energy	Scalability	Robustness
Memristor Ratioed Logic	VVH	No	No	No	+	++	++	++	+
PLA-like Memristor Logic	VVH	No	No	No	+	++	++	++	++
Current Mirror Threshold Logic	VVH	No	No	No	+	++	++	++	++
Programmable Threshold Logic	VVH	No	No	No	+	++	++	++	++
Parallel Input Processing Logic	VVM	No	No	No	+	++	++	++	+
Pinatubo	RVH	Yes	Yes	Yes	+	+	++	+	+
CMOS-like Logic	RVM	No	Yes	Yes	-	-	-	-	+
CRS Logic	VRM	Yes	Yes	Yes	-	-	-	-	-
Snider Logic	RRM	Yes	Yes	Yes	-	-	-	-	-
Stateful Logic	RRM	Yes	Yes	Yes	-	-	-	-	-
Normally-off Logic	RRM	No	Yes	Yes	-	-	-	-	-

- **Robustness**: how robust the logic style is to be resilient against the variation of CMOS and memristor technology.

The comparison result is listed in Table 1.1. Symbols '-', '+', and '++' represent 'bad', 'medium', and 'good', respectively. Following conclusions can be drawn with respect to the metrics.

- **Array Compatibility**: Array compatibility is an important requirement to implement resistive computing systems. Design styles in the RVH, VRM and RRM (except for Normally-off Logic) classes are compatible with memory arrays. Due to its irregular topology, CMOS-like Logic is not compatible with memory arrays. Since CMOS inverters or D flip-flops need to be added to memory arrays, design styles of VVH are not compatible with 1R/1T1R array. Parallel input processing logic is not compatible with 1R/1T1R array, but can be used in an array with more complex topology [46].

- **CMOS Controller Requirement**: Additional CMOS control units are not required for the logic styles of VVH and VVM since their inputs and outputs are voltages. On the contrary, the data need to be transduced between voltages and resistances in other logic styles. Several logic design styles require multiple execution steps and hence a controller is needed to execute these steps.

- **Nonvolatility**: Since the inputs and outputs of the design styles in the VVH and VVM classes are both represented by voltages, these design styles are volatile. In contrast, other logic styles represent their input and/or output by resistances and hence are nonvolatile.

- **Speed**: The design styles in VVH, VVM, and RVH classes are faster than the others because they can finish logical operations in a single step. Oppositely, other logic design styles are slow as they need multiple steps.

- **Area**: Since CMOS controllers are not needed, the design styles in the VVH and VVM classes require smaller area than the others. Note that Pinatubo only requires a simple controller as the operation is conducted in a single step instead of multiple ones [39].

- **Energy**: Controller necessity, nonvolatiltiy, and speed all impact the energy consumption. Design styles in the VVH and VVM classes do not require CMOS controllers and they are fast; therefore, they are likely to consume less energy compared with the others. Pinatubo is nonvolatile and fast, and hence it is likely to consume less energy as well. For the other design styles, more energy are consumed during the logical operations as they need complex controllers and longer more steps.

- **Scalability**: The scalability is mainly decided by array compatibility and CMOS controller requirement. Design styles of VVH and VVM are the easiest to scale up as CMOS controllers are not required. Pinatubo is relatively easy to scale since it only needs a simple controller. However, the other design styles are difficult to scale up as complex controllers are needed.

- **Robustness**: Since many transistors exist in CMOS controllers, controller necessity impacts the robustness. In addition, if the memristors do not switch during logical operations, this design style is more reliable than the others. The reason is that memristor devices suffer from cycle-to-cycle variation [6]. Design styles of VVH (except for Memristor Ratioed Logic) are likely to be most robust as CMOS controllers and memristor switching are not needed in logical operations. Memristor Ratioed Logic and Parallel Input Processing Logic are less robust because they are more sensitive to the resistance variation of the memristors. Design styles in RVH and RVM classes are more reliable than others since memristors are not switched during logical operations.

In summary, design styles of RVM, VRM, RRM, and RRH are suitable to implement the resistive computing architectures due to their array compatibility. In addition, the design styles in the VVH and RVM classes are potential alternatives for replacing CMOS logic.

1.1.4. Memristive Devices for Memories

Many non-volatile memory elements have been proposed such as resistive RAM (RRAM) [28], ferroelectric field-effect-transistor (FeFET) [47], phase-change memory (PCM) [48], spin-transfer torque magnetic RAM (STT-MRAM) [49]. Each of these device classes are based on different technologies and their working principles differ. As a result, these devices have different benefits and drawbacks, leading to different appropriate use scenarios. In this section, We will briefly overview these memristive devices used as memories.

Figure 1.6 summarizes the storage capacity of recently produced NVM chips based on their classes [50]. The figure shows that many prototypes have been developed, and the NVM technology is an active research field. We refer the readers to the first two chapters of the book *Resistive Switching* [51, 52] for a comprehensive introduction into the the topic of memristive memory and the RRAM technology.

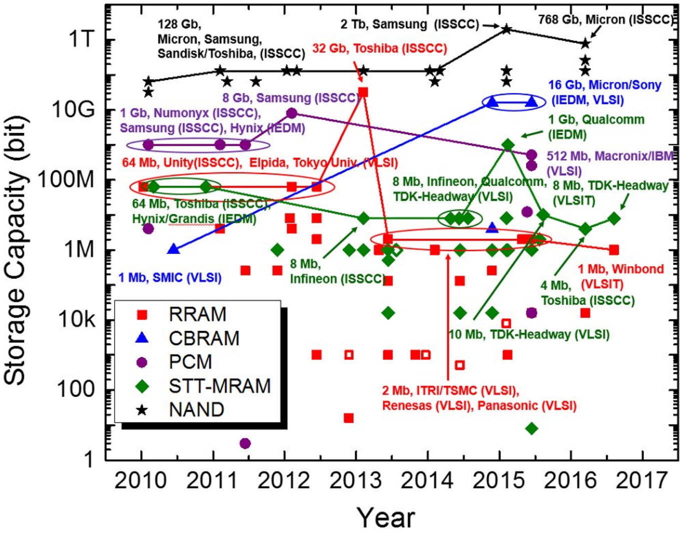


Figure 1.6: Memristive device trend data [50].

PCM devices are based on the usage of chalcogenide materials that can change between an amorphous and a crystalline state [48]. The switching is realized by using a high write current to heat up a conductive rod reaching through the chalcogenide material. When a current is flowing through a PCM device, the amorphous and a crystalline states exhibit different behaviors in their electric resistances. The device is in low resistance state (LRS) when the chalcogenide is in a crystalline state. Otherwise, it is in a high resistance state (HRS). Furthermore, intermediate states may exist between these two extremes, i.e., a combination of LRS and HRS. This possibility leads to the first benefit of such PCM devices, namely its feasible multi-level cell operation. In addition, the manufacturing technology of PCMs is quite mature and it is compatible with CMOS technology. The endurance of PCMs, i.e., the maximum number of possible switching cycles before the device becomes unreliable, is more than 10^9, which is comparable to RRAMs [53]. They have the highest endurance among current NVM devices. However, there are several challenges regarding the controlling of such switching processes, including the necessary high write circuits, a $10\times$ longer switching time than RRAMs due to the slow crystalline process, and the resistance drift in the amorphous state that has to be compensated for at circuit level [53].

STT-RAMs are based on a parallel and anti-parallel configuration of a stack of ferromagnetic layers that form a magnetic tunnel junction (MTJ) structure [49]. The magnetization at the terminals of the MTJ stack is fixed on one side. Therefore, this side is denoted as a *fixed layer*. The magnetization of the other side can be switched between two magnetization directions, which is called the *free layer*. If both layers are in parallel to each other, the electrons that are spin-polarized with opposite orientation can pass through the stack with a high probability. Therefore, the device is in LRS in this case. On the contrary, the probability that an electron can pass both layers is low if the two layers are polarized anti-parallel to each other. The reason for this is that the electrons will always encounter a layer with an opposite polarization relative to its own one, no matter in which direction the electron is spin-polarized. Therefore, the device is in a HRS in this case. The main advantage of STT-RAM is the short switching time [54]. Its manufacturing technology is relatively mature. However, it is challenging to make it compatible with CMOS. The MTJ stack may consist of more than ten layers of ferromagnetic materials, e.g. CoFeB or MgO, which are not easy to handle [53]. However, due to its low energy efficiency STT-MRAM technology is not likely to be used in last-level caches.

We can subdivide the RRAM technology into three categories, i.e., electrochemical memory (ECM), valence change memory (VCM) (see Figure 1.7), and thermochemical memory (TCM) based on their nanoionic switching mechanisms [28]. Different ionic mechanisms are used to generate resistances. In TCMs and ECMs, small metallic bridges are build up and down by a structure called filamentary with the redoxation and oxidation processes in ionized material layers. These layers consist of materials such as TiO_2 or HfO_2, which are entangled between two metal plates as terminals. TCMs are unipolar, i.e., the same voltage is applied to the poles and a filament with low resistance characteristic grows from both sides. Different from that, two opposite voltages are applied to the terminals in ECM, which are normally composed of different metals. Using this bipolar control mechanisms, voltage and reversed voltage signals are used to build up the metallic filament by a redox transitions and to dissolve it again by launching local oxidation processes [55].

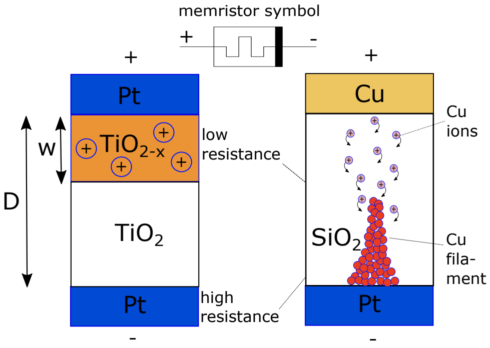


Figure 1.7: ECM (left) and VCM (right) RRAMs.

In VCMs, a variant of RRAM, the exchange of ions builds up and dissolves not only a filament but also a complete metallic layer or an area interface. This technique that was used in the memristors of HP Labs [56]. They are bipolar and offer good scalability because the cell sizes can be made in the namometer range, e.g. 10×10 nm^2 or even less [57]. The reason is that the underlying switching process focuses on much more localized structures. Another advantage is the large HRS / LRS ratio, which requires a simpler CMOS circuit to evaluate the resistance. Fast switching is a feature of RRAM as well. This can be realized in the ns range, and even 100 ps have been demonstrated [58]. This characteristic origins from two facts: 1. the ions have to move in small distances; 2. the high electrical field forces that occur in the nanoscale active region causes an effect called *Joule heating*, and it further increases the ion mobility. A further advantage of RRAMs is the good compatibility with CMOS manufacturing processes. The endurance, which is may be the most important feature for memristive elements concerning their usage either as memory or as switching element in computing circuits, is reported very differently. Values from 10^6 cycles up to more than 10^{12} cycles can be found in literature [53]. The power consumption is in the pf range, which makes RRAMs a good candidate for an use in embedded applications. For example, Panasonic becomes the first semiconductor manufacturer that integrated RRAM into a microcontroller to store firmware in 2013 [59].

1.2. Opportunities and Challenges

This section discusses the opportunities and challenges of building in-memory computing architectures using memristive devices.

1.2.1. Opportunities

Memristive devices have the potential to contribute to computing technologies with respect to the following aspects.

- *Memory hierarchy*: Due to speed, cost, and endurance limitations, emerging resistive memories are not likely to replace mainstream memories such as DRAM and SRAM. However, they may provide other opportunities. Figure 1.8 shows the typical access time of a conventional memory hierarchy versus resistive memories. From the figure it can be observed that a speed gap exists between the DRAM and storage (i.e., solid-state drive (SSD) or hard-disk drive (HDD)). Based on the access time, some of the resistive memories such as RRAM and PCM can fill this gap. Therefore, it is possible to insert these NVMs as a new level of memory to fill the gap, which is referred to as storage-class memory (SCM) by some researchers [60]. As the capacity of NVMs can be larger than DRAM [60], SCM will decrease the average data access time and hence improve the performance of conventional computing systems.

- *In-memory computing*: In Von-Neumann architectures, a lot of time and energy is wasted in fetching data from and storing the results back to the memory. Memristive devices support both storage and computation and hence the communication cost can potentially be reduced. Many researches have shown

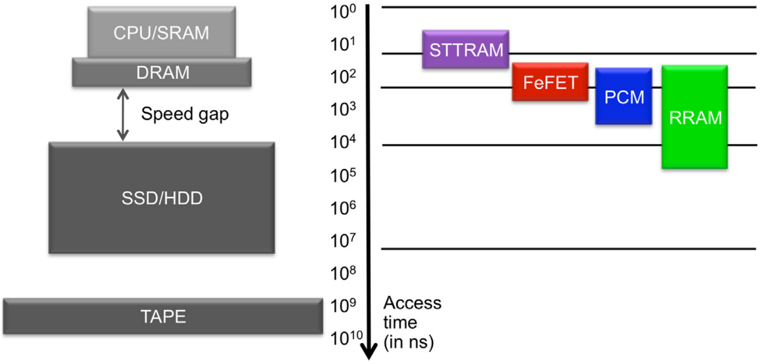


Figure 1.8: Memory hierarchy and typical access speed [24]

the potentials of in-memory computing with memristive devices [39, 61–64]. It is also the topic of this thesis.

- *Neuromorphic computing*: Another active research field is to use memristive devices for the hardware implementation of brain-inspired neuromorphic computing platforms. The multilevel storage capacity of PCRAM and RRAM allows them to serve as analog devices that can emulate the function of plastic synapses in a neural network. Synaptic weights are modified by the timing difference between pre- and post-synapses neuron signals, i.e. spike time dependent plasticity (STDP), which is similar to the resistance changing process of memristive devices [65]. In addition, the multiply accumulate (MAC) function, which is important in neuromorphic computing and artificial intelligence, can be implemented efficiently within an NVM crossbar [66, 67]. Many implementations in this domain have been demonstrated successfully such as digit recognizing [68], image classification [69] and natural language processing [70].

- *Low-power designs*: The non-volatile feature of memristive devices can be utilized to build low-power hardware. Power and energy consumption are becoming more critical for computing systems, especially for those depending on batteries. When a memristive based memory or computing component is idle, it can be turned off without information loss and hence the stand-by power can be eliminated. This enables a "normally-off" working style, which could benefit data centers and Internet-of-things (IoT) devices [24].

- *Hardware security*: The intrinsic variations of memristive devices can be exploited in the domain of hardware security. Stochastic behavior has been observed in the switching process of memristive devices which could be exploited to create a true random number generator (TRNG) [71]. Similarly, the resistance variability of memristive devices provides an alternative source of randomness to implement a physical unclonable function (PUF) [72]. TRNGs and PUFs are both important primitives for hardware security, which can be used to identify or authenticate specific systems.

1.2.2. Challenges

Although memristive devices have many potentials, several challenges still need to be addressed.

- *Switching speed*: Changing the state of a memristive device requires at least tens of nanoseconds [73]. It is approximately the same latency as writing a DRAM cell (see Figure 1.4), and much slower than SRAM. Many logical schemes of memristive devices use the resistance state as output. Therefore, the speed of such schemes is bounded by the writing latency, especially when the writing latency dictates the clock frequency.

- *Dynamic power*: Besides the long latency, the changing of memristive states also requires a high programming voltage (2 V to 5 V), large current (10 μA to 100 μA) and high energy (0.1 pJ to 10 pJ) [53]. The need of high voltages and currents increases the difficulty of the circuit design, and may increase the energy consumption of other components. The energy consumption to write a single bit is one to three orders of magnitude higher than that of DRAM. Nevertheless, NVM does not require a periodical refresh. Therefore, overall they can still be more energy efficient than DRAM.

- *Endurance*: Although STT-MRAM has a desirable endurance (10^{15}), FeFET, PCRAM, and RRAM suffer from low endurance which is typically in the range 10^6 to 10^{12} [53]. In case a memristive device changes its resistance state with a frequency of 1 MHz, its lifetime will be between a second and two weeks. This is clearly not acceptable for practical usage. As a comparison, the endurance of commercial SRAM and DRAM is about 10^{16}.

- *Variability*: Due to the intrinsic stochastic switching process, memristive devices and in particular RRAM, suffer from high variability [74]. The high variability decreases the read margin for sensing amplifiers, leading to more complex circuit design and less storing bits per device. In addition, the high variability impacts the robustness of logical operations. The designer must consider all the corner cases to ensure correct operation.

- *Process compatibility*: Although STT-MRAM has advantages such as a low programming voltage, fast write speed, and high endurance [75], its compatibility with current mainstream CMOS technology is relatively poor [53]. The main reason is that many layers of exotic ferromagnetic materials are used in the MTJ stack. However, PCRAM and RRAM are compatible with CMOS technology.

1.3. Research Topics

Many challenges described in Section 1.2 still need to be addressed. The research carried out in this thesis focuses on the full-stack support for the in-memory computing paradigm built with memristive devices. It covers the circuit level, architecture level, and design automation.

- **Circuit Level:** At least two directions are worth further exploration for circuit designing with memristive devices. First, novel designs are still needed as currently the number of supported operations is limited. Second, the inferior properties of current memristive devices such as low endurance and large resistance variation may affect their usage in industrial products. Therefore, it is worth investigating durable and robust schemes that are resilient to these properties. This thesis explores the methods of using resistive sensing to perform logical operations, as it does not change the states of memristive devices. In addition, we need to guarantee operation correctness even under large resistance variation.

- **Architecture Level:** A circuit has to be integrated into architectures before being able to run applications. This thesis explores efficient architecture designs for different types of applications. Especially, we consider the usage of memristive devices and the computing kernels proposed in the Chapter of Circuit Level. Besides designing the architecture, we investigate the methods for evaluating them and estimating their performance.

- **Design Automation:** Data-intensive applications often lead to large design scales that exceed the capacity of manual designing. Hence, design automaton is essential in this scenario. This thesis investigates the methodologies to assist the users in developing applications on the architectures we propose. Different from existing syntax tools that target small-scale circuits such as an adder, we focus on system-level design. In addition, to generate designs with superior quality, we explore techniques for optimization.

1.4. Contributions

The contributions of this dissertation are directly related to the research topics presented in the previous section.

1.4.1. Circuit Level

We study existing schemes that use memristive devices for logical operations and propose two novel ones. With respect to this research topic, the main contributions are as follows:

1. A durable logical operation scheme [76]. To overcome the short lifetime problem of memristive devices that is caused by their low endurance, we conduct logical operations during resistance sensing. We modify the sense amplifier, making it able to compare the overall sensing current of two input memristive devices with pre-defined references. According to the comparison result, the sense amplifier generates the results of logical operations. In this way, the states of the memristive devices stay unchanged. Thereafter, we evaluate the scheme and compare it with other designs.

2. A robust logical operation scheme [77]. To overcome the high failure rate of logical operations that are caused by the large resistance variation of memristive devices, we improved the previous scheme by sensing the overall current through different paths for AND and or operations, respectively. Similarly, we change the way of setting reference values. Finally, we evaluate this scheme using the Monte Carlo simulation and compare its robustness against the state of the art.

1.4.2. Architecture Level

We investigate two types of data-intensive applications and propose three architectures for processing them. With respect to this research topic, the main contributions are as follows:

1. A heterogeneous architecture for vector bit-wise logical operations [76]. We combine a conventional von Neumann architecture with an accelerator built with memristive devices. The accelerator can communicate with the CPU and directly visit the external memory. It is used to accelerate bit-wise logical operations. We evaluate this heterogeneous architecture with an analytical model and compare the result with a multi-core system.

2. Two architectures for automata processing [78, 79]. We investigate existing hardware accelerators for automata processing and describe a specific group of them using an abstract model. In this model, memory arrays store configuration information and work as computing components at the same time. Then, we instantiate this model using memristive devices. This design is evaluated with SPICE simulation and compared with similar ones that are based on other types of memory technologies. Subsequently, we improve the design with pipelining and time-division multiplexing. These changes increase the working frequency and hence the throughput. We use SPICE to simulate the design to determine its maximum throughput and synthesize key components to estimate the area. Finally, its throughput and area are compared with the state of the art.

1.4.3. Design Automation

We investigate the design automation methodologies for the proposed architectures. With respect to this research topic, the main contributions are as follows:

1. A synthesis flow for CIM architectures [80, 81]. We extend the skeleton concept in the software domain with placement and routing information and apply it to the synthesis flow for CIM architectures. Then, we define four skeletons that represent common algorithmic structures and develop solution templates for them. These solution templates contain scheduling, placement, and routing information. Finally, we verify the synthesis flow with three test cases.

2. A compiler for automata processors [79]. First, we investigate the current compiling tools for automata processors. Then, we build our complier based on a graph-partitioning tool. We develop multiple methods for resolving the

constraint conflicts that may occur during the compilation. In addition, different partitioning strategies are developed for exploring the design space. Finally, we use a standard benchmark suite to evaluate the compiler and compare its equality with the state of the art.

1.5. Thesis Organization

The remainder of this thesis is shown in Figure 1.9 and organized as follows.

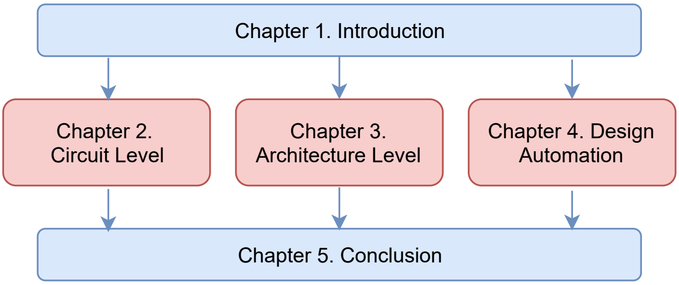


Figure 1.9: Thesis outline.

Chapter 2 discusses the contributions of this dissertation with respect to the circuit level. It presents logical operation schemes that utilize current immature memristive devices.

Chapter 3 discusses the contributions of this dissertation with respect to the architecture level. It first presents an architecture containing the circuit proposed in Chapter 2 as an accelerator. Thereafter, it presents two architectures built for automata processing.

Chapter 4 discusses the contributions of this dissertation with respect to design automation. It first presents a synthesis flow for computation-in-memory architectures. Thereafter, it presents a compiler that maps automata to the architecture proposed in Chapter 3.

Chapter 5 concludes this dissertation and discusses possible future research directions.

2

Circuit Level

This chapter presents two logical operation circuits based on memristive devices. The first one uses resistive sensing to perform logical operations, and hence does not require state changes of memristive devices. It improves the delay and power compared to the state of the art. The second logic scheme enhances the first one by using different sensing paths for AND and OR operations. It guarantees the correctness of logic operations even under the presence of large resistance variations.

The content of this chapter consists of the following research articles:

1. L. Xie, H. A. Du Nguyen, **J. Yu**[1], A. Kaichouhi, M. Taouil, M. Alfailakawi, S. Hamdioui, *Scouting Logic: A Novel Memristor-based Logic Design for Resistive Computing*, IEEE Computer Society Annual Symposium on VLSI (ISVLSI'17), Bochum, Germany, July 2017, pp. 151-156.

2. **J. Yu**, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, S. Hamdioui, *Enhanced Scouting Logic: A Robust Memristive Logic Design Scheme*, The 15th IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'19), Qingdao, China, July 2019, pp. 1-6.

[1] J. Yu contributed to the variation resilient design and SPICE simulation.

2.1. Problem Statement

Memristive devices such as RRAM suffer from a low endurance and large resistance variation [24, 53]. Endurance refers to the number of times the resistance state of a memristive device can change. Low endurance may lead to a short lifetime. Resistance variation refers to the resistance difference between different memristive devices. Large resistance variations may affect the robustness of the computations. To utilize immature memristive devices for logical operations, we must develop logic schemes that are resilient to these drawbacks. This chapter proposes such logic schemes.

- *Durable Logic Scheme*: Many memristive logic design schemes, such as Snider [36, 76], Stateful Logic [31, 37, 43], CRS [32, 35], MRL [30], and MAGIC [44], change the states of memristive devices frequently. For example, in MAGIC the output device (i.e., the device that will store the result) needs to be initialized to low resistance before applying a NOR operation. In case the previous output value of this device was high resistance and the new output value as well, this output device will undergo unnecessary transitions. The endurance of RRAM devices is between 10^6 and 10^{12} [53]. Hence, in case an RRAM device changes its resistance state with a frequency of 1 MHz, the lifetime of this device will be between a second and two weeks. This is clearly not acceptable for practical usage. Therefore, there is a need to develop logic schemes that do not require frequent state changes.

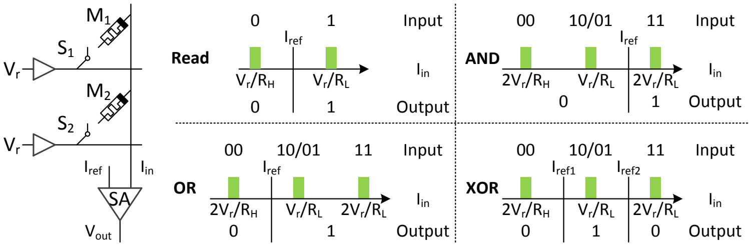
- *Robust Logic Scheme*: The resistance variation of RRAM devices originates from the fluctuations in filament radius and constriction geometry [82]. Therefore, it is an intrinsic characteristic of RRAM devices. Unfortunately, many memristive logic design schemes do not consider the resistance variation in their verification methodology [30, 39, 44, 83], or assume a very small resistance variation (e.g., resistance difference / mean $< 10\%$) [76, 84–87]. However, the resistance variation of current RRAM devices is much larger than these assumed values [88–93]. In some cases, the upper bound of the resistance range is several times higher than the lower bound. Most existing memristive logic design schemes produce wrong results under such variation conditions. Therefore, a novel robust logic scheme is required.

2.2. Main Contributions

The main contributions with respect to the above aspects are as follows.

- *Durable Logic Scheme*: We propose a durable logic scheme referred to as *Scouting Logic* [76]. Its main idea is illustrated in Figure 2.1. It uses resistive sensing to perform logical operations. The input values are stored in the memristive devices inside the memory array in the form of resistance. During logical operations, multiple memory columns are enabled at the same time (see Figure 2.1(a)). The modified sense amplifier compares the overall current with references and outputs the result in the form a voltage (see

Figure 2.1(b)). During this process, the resistance of the memristive devices does not changed. Therefore, these operations do not affect the lifetime. In addition, by avoiding changing the states of memristive devices, Scouting Logic accelerates the operation speed and reduces the energy consumption. Evaluation results show that Scouting Logic achieves less delay and lower power than the state of the art for a similar area overhead.



(a) Memory (b) References of Primitive Operations

Figure 2.1: Main idea of Scouting Logic [76].

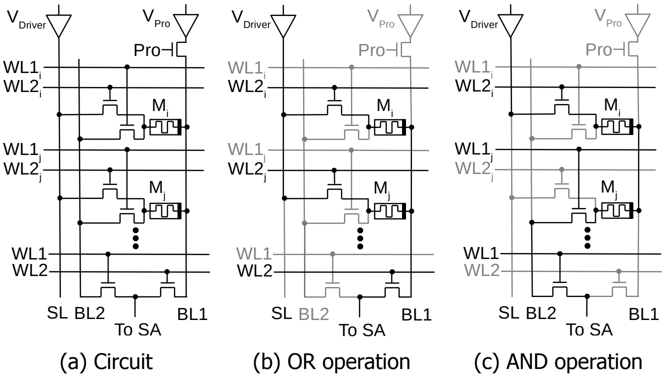
- *Robust Logic Scheme*: Considering robustness against resistance variation, we propose another logic scheme named *Enhanced Scouting Logic* (ESL) [77]. Its circuit is illustrated in Figure 2.2(a). Similar to Scouting Logic, it conducts logical operations during sensing. However, it uses two different paths for AND and OR operations, which connect the input memristive devices in series (see Figure 2.2(b)) and in parallel (see Figure 2.2(c)), respectively. In this way, ESL can guarantee operation correctness even if large variation exists in these devices. Monte Carlo simulations validate that the robustness of ESL exceeds the state-of-the-art schemes as shown in Table 2.1. In this table, the second to the fifth columns list the number of failed test cases under different input Boolean values. ESL provides a method to build reliable logic circuits using today's immature devices.

Table 2.1: The Numbers of Failed Cases in 10,000 Monte Carlo Iterations

	00	01	10	11	Total
Scouting Logic [76]	0	75	97	202	374
Pinatubo [39]	0	142	176	332	650
ESL (This work)	0	0	0	0	0

2.3. Evaluation

In this chapter, we presented two logic schemes that tolerate the drawbacks of current RRAM devices, which were overlooked by many state-of-the-art designs.



(a) Circuit (b) OR operation (c) AND operation

Figure 2.2: ESL circuit [77].

Scouting Logic and ESL can maintain a long lifetime with low-endurance devices and ESL is robust despite large variations. The followings are the consideration to extend ESL and alternative methods to handle the variation challenge:

- The types of operations that can be performed by Scouting Logic and ESL are still limited and hence more research is required. For example, a shift operation is essential for many arithmetic operations and encryption algorithms [94]. A method to implement the shift operation is to add a CMOS shifter in each RRAM array and operate on the output of the sense amplifiers [64, 95]. However, this method leads to a larger chip area and higher power consumption.

- Self-write termination [54] is a promising technology that can alleviate the resistance variation of RRAM devices. It adds a loop-back from the cell to the driver during write operations. When the programmed memristive device reaches the desired resistance, the writing process is terminated. Currently, it cannot be applied to some devices and may lead to unstable resistance states [96]. In addition, it is also difficult to achieve a small variation using SWT schemes for some devices [96]. However, if it can overcome these drawbacks, the logic computing schemes can be simplified.

- If the memristive devices are used in an approximate computing design, the resistance variation would not be an issue. It is because that approximate computing can tolerate inaccuracy to a certain level. In addition, OR and AND can be used to implement a one-bit full adder in the context of approximate computing [97]. It can be further used for implementing some image processing applications [97].

3

Architecture Level

This chapter presents three computation-in-memory architectures based on memristive devices. The first one is a heterogeneous architecture containing a Scouting Logic component to accelerate vector-based bit-wise logical operations. Its performance-energy efficiency is 10× *higher than a multi-core system. The second architecture accelerates automata processing. It outperforms similar ones that are based on conventional memory technologies. The third one is an improved version of the second one. It achieves a higher throughput by pipelining the routing network and using the pipeline in a time-division multiplexing manner. SPICE simulations show that the performance of the last two architectures is higher than prior work.*

The content of this chapter consists of the following research articles:

1. **J. Yu**, H. A. Du Nguyen, L. Xie, M. Taouil, S. Hamdioui, *Memristive Devices for computation-in-memory*, The 21st Design, Automation & Test in Europe Conference & Exhibition (DATE'18), March 2018, pp. 1646-1651.

2. **J. Yu**, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, S. Hamdioui, *Time-division Multiplexing Automata Processor*, The 22nd Design, Automation & Test in Europe Conference & Exhibition (DATE'19), Florence, Italy, March 2019, pp. 794-799.

3.1. Problem Statement

The circuits proposed in Chapter 2 need to be integrated into architectures to solve real-life problems. The researches on such architectures can be divided into two groups according to their scope. The first group focuses on the design of an accelerator, e.g., PLiM [63], PRIME [62], Computation-in-Memory (CIM) [61, 98], and ISSAC [67]. The second group focuses on the system level, considering both the host processor and the memristive device-based accelerator, e.g., AC-DIMM [99], Pinatubo [39], and IMP [64].

An architecture contains multiple components and can be used to run certain applications. We are especially interested in data-intensive applications mainly for two reasons. Firstly, they are important Big-Data problems [100]. Secondly, conventional von-Neumann architectures suffer from the memory wall in these applications [11]. Memristive devices support both data storage and logic operations and thus have the potential to be used in in-memory computing architectures. These architectures can alleviate the memory-wall problem and outperform conventional architectures. This chapter will present novel architectures based on memristive devices for the following applications.

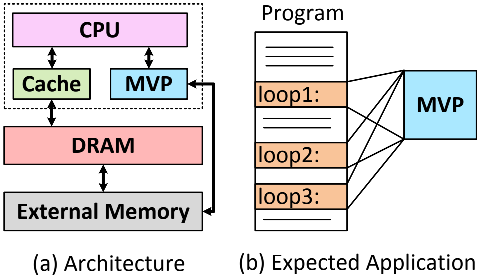
- *Applications containing massive logical operations*: Vector bit-wise logical operations are commonly seen in multiple applications such as database management [101], DNA sequencing [102, 103], and graph processing [104]. The operations on each element are simple; however, the vector length is significant. To process this type of operations on conventional architectures, all the data needs to be loaded to the cache sequentially, which leads to low efficiency and high energy consumption.

- *Automata processing*: Many applications, such as network security [105], bioinformatics [1], and artificial intelligence [2], need to match an input sequence with pre-defined patterns. This type of matching can be modeled using finite-state automata. However, processing automata on conventional architectures is not efficient when the automata size is larger than the cache's capacity. In that case, the bad data locality of automata will cause many cache misses. Implementing automata processing with FPGAs have similar problems as their capacity is limited. Instead, several ASIC-based accelerators have been proposed. Unified automata processor [106] simplifies CPU cores specially for automata processing. However, its throughput is limited when the processing automata contains many active states. HAWK [107] and HARE [108] use logic gates for matching. They process multiple input symbols of a single input stream in each clock cycle, thus achieving a higher throughput. However, they do not support all automata.

3.2. Main Contributions

The main contributions with respect to the above aspects are as follows.

- *Vector bit-wise logical operations* [78]: We build an accelerator for vector bit-wise logical operations based on Scouting Logic [76]. This accelerator is referred to as Memristive Vector Processor (MVP). MVP can communicate with CPU and directly visit external memory as shown in Figure 3.1(a). MVP accelerates the program sections that contain bit-wise logical operations while the rest is still executed by CPU as indicated by Figure 3.1(b). The evaluation shows that MVP achieves a $10\times$ improvement in performance-energy efficiency over a multi-core system.



(a) Architecture (b) Expected Application

Figure 3.1: Memristive Vector Processor architecture.

- *Automata processing* [78, 79]: First, we propose a general architectural model based on existing automata processing accelerators such as Micron's Automata Processor [109] and Cache Automaton [110]. In this model, the memory arrays store configuration information and are also used as computing components. Next, we develop an architecture (shown in Figure 3.3a) based on the proposed model using memristive devices, which is referred to as RRAM-AP [78]. The memory arrays are fragmented across the entire chip and we refer to each fragment as a *tile*. Due to the small intrinsic capacitance of memristive devices, RRAM-AP achieves 35% performance and 59% energy improvement over Cache Automaton. Finally, we propose an architecture that further accelerates automata processing using time-division multiplexing. This architecture breaks the routing network into multiple pipeline stages as shown in Figure 3.3b. Each pipeline stage processes a different input sequence. In this way, the architecture reaches higher throughput with a negligible area overhead. Table 3.1 shows the evaluation results of this architecture against the state of the art.

3.3. Evaluation

In this chapter, we presented three in-memory computing architectures. Unlike previous works such as PLIM [63], ReGP-SIMD [111], and MPU [112], the architectures presented in this chapter can cope with the low endurance problem of RRAM. In MVP and RRAM-AP, the memristive devices are not programmed frequently. In MVP, we assume that the original database or dataset is stored in memristive arrays,

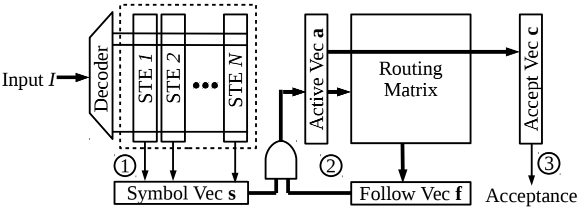
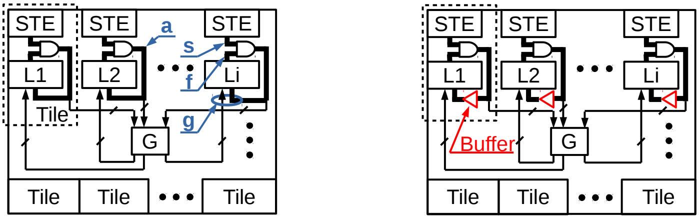


Figure 3.2: General architecture of Automata Processors [78].



(a) RRAM-AP

(b) TDM RRAM-AP

Figure 3.3: RRAM-AP and TDM RRAM-AP.

Table 3.1: Evaluation of Automata Accelerators

	Frequency (GHz)	Throughput (Gbps)	Area (mm^2)
HARE (w=32) [108]	1.0	3.9	80
UAP [106]	1.2	5.3	5.67
Cache Automaton [110]	2.0	15.6	4.3
This work	3.0	24.0	3.16

and hence they do not change frequently. In RRAM-AP, the RRAM array stores the configuration of target automata, which does not change during the processing. In both architectures, the computation occurs during modified read operations, which does not affect the device endurance. The work presented in this chapter can be further improved with respect to the following aspects:

- The evaluation of MVP can be conducted in more detail. In [78], we used an analytic model to evaluate the performance and energy consumption of MVP. However, it would be more realistic if applications are simulated. Therefore, in a later publication, we simulated a similar architecture that was application-aware by using state-of-the-art tools including Cacti, NVSIM, and SiNUCA [113].

- The outputs of RRAM-AP can be improved to provide more information to the user. In RRAM-AP, a column in each local switch and a 64-to-1 OR gate are used to report whether a match occurs in each cycle. However, for some applications such as PROTOMATA, it is also useful to know which state reports the match if a match occurs. It requires additional hardware for such reporting. MAP has implemented such hardware. However, it becomes a bottleneck when matches occur frequently [114]. In those cases, the host CPU cannot process the reported information fast enough, and hence MAP has to decrease the processing speed. The hardware structure of match reporting should be designed carefully to make full use of the IO bandwidth[114].

- The energy consumption of RRAM-AP can be studied in more details. We measured the energy consumption of an RRAM array with respect to one group of input data in RRAM-AP [78]. However, the energy consumption of other components and the RRAM array regarding other inputs is not analyzed. In [110], the authors first estimated the average energy consumption of a one-bit hit in each component. Then, for each benchmark, they simulated the execution of the automata and summed up the energy consumed in each operation. We can adopt this process and improve energy consumption evaluation regarding RRAM-AP.

4

Design Automation

This chapter presents a synthesis flow for CIM architectures and a compiler for automata processors. The synthesis flow is based on the skeleton concept, which relates an algorithmic structure to a pre-defined solution template. This solution template contains scheduling, placement, and routing information. By rewriting the application using predefined algorithmic structures, a CIM circuit can be generated accordingly. The compiler for automata processors uses multiple strategies to transform given automata, so that constraint conflicts can be resolved automatically. It also optimizes the mapping for storage utilization. Evaluation with a standard benchmark suite shows that the proposed compiler outperforms the state of the art.

The content of this chapter consists of the following research articles:

1. **J. Yu**, R. Nane, A. Haron, S. Hamdioui, H. Corporaal, K. L. M. Bertels, *Skeleton-based Design and Simulation Flow for Computation-in-Memory Architectures*, The 12th IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'16), Beijing, China, July 2016, pp. 165-170.

2. **J. Yu**, R. Nane, I. Ashraf, M. Taouil, S. Hamdioui, H. Corporaal, K. L. M. Bertels, *Skeleton-based Synthesis Flow for Computation-In-Memory Architectures*, IEEE Transactions on Emerging Topics in Computing (TETC), Volume 8, Issue 2, 2020, pp. 545-558.

3. **J. Yu**, M. Abu Lebdeh, H. A. Du Nguyen, M. Taouil, S. Hamdioui, *APmap: An Open-Source Compiler for Automata Processors*, submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), undergoing a minor reversion.

4.1. Problem Statement

Design automation is essential for developing large applications. Since the architectures we proposed are different from conventional ones, we should also provide corresponding tools, such as compilers and synthesis tools, to the users.

In the literature, there are mainly two types of design automation tools with respect to memristive circuits and architectures. One type is synthesis tools that generate circuits based on hardware description language (HDL) inputs [115–118]. The other type is compilers that generate instructions for specific architectures, e.g., for PLiM [119] and IMP [64]. When developing these tools, we have to consider the application scale and the features of the architectures. This chapter explores the design automation methodologies for the architectures discussed in previous chapters.

- *CIM architectures*: CIM architectures are designed manually in previous research. In [120], authors implement a parallel adder, which calculates the sum of an array, by placing basic adders in a grid. In [121], authors mapped a matrix multiplication to adders and multipliers that are placed in an H-tree style. However, the application presented in these works are naive. For more complex applications, design automation is required to generate the detailed structure of the desire architecture, such as the position and routing of the circuit components. However, developing such a design automation tool is challenging due to two reasons. First, the application scale is large. As a single memristive computing component is slower than the one with CMOS technology, memristive CIM architectures can only achieve higher performance with a larger number of components operating in parallel. Second, CIM differs from CMOS-based circuits as the memristive devices are passive. This feature affects the way of exchanging data among computing components. Therefore, we cannot reuse the synthesis tools designed for CMOS circuits.

- *Automata Processors*: Automata Processors also require design automation, mainly because of the application scale and hardware complexity. For example, Cache Automaton (CA) [110] contains ten million configurable bits. These bits determine the behavior of the hardware. It is nearly impossible to map a large automaton to such a large number of configurable bits without the usage of a compiler. However, existing compilers for Automata Processors are not satisfactory. The official compiler for Micron's Automata Processor is closed-source [122]. Therefore, it can not be adapted for other automata processors. There is a compiler that can map automata to Cache Automaton as well; however it is also closed-source and secondly not fully automated. The only open-source tool available is ATR [122] which has been developed based on an FPGA routing tool to estimate the hardware resource (e.g., the number of configurable wires between two tiles) needed in an automata processor. As a consequence, it is not accurate and does not generate detailed configurations. Hence, a fully automated open-source tool is still needed for automata processors.

4.2. Main Contributions

The main contributions with respect to the above aspects are as follows.

- *CIM architectures* [80, 81]: We propose a skeleton-based synthesis flow for mapping algorithms to CIM architectures. This flow is illustrated in Figure 4.1. In conventional hardware design flows, scheduling, placement, and routing are separated processes that are conducted sequentially. In CIM architectures, circular dependency exists in these processes as the communication cost is determined by the routing. To generate optimal circuits, we apply the scheduling, placement, and routing processes simultaneously in the form of skeletons. These skeletons are pre-defined solution templates for commonly-used algorithmic structures (Box 2). To use this synthesis methodology, the user first partitions the original program into software and hardware, and rewrites the hardware part using skeletons (Box 1). Then, the CIM circuit will be generated by instantiating pre-defined solution templates with primitive circuits (Box 3). Box 4 shows the process of developing primitive circuits.

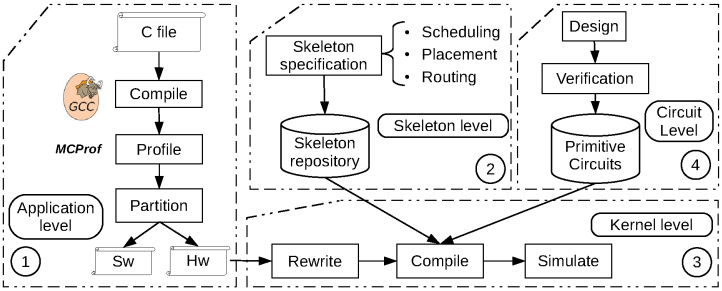


Figure 4.1: Synthesis flow for memristor-based CIM architecture [81].

- *Automata Processors* [123]: We have developed a compiler named APmap for Cache Automaton [110] and RRAM-AP [78]. Cache Automaton's compiler cannot resolve constraint conflicts automatically. However, APmap uses multiple strategies to change the given finite state automaton to equivalent forms, so that constraint conflicts can be automatically resolved. In addition, APmap optimizes the mapping result for storage utilization. This means that the same AP chip can be potentially used for larger automata. Using a standard automata benchmark suite [124], we compare the performance of APmap against Cache Automaton's compiler as shown in Figure 4.2. The legend items 'Ideal CA' and 'Ideal APmap' in the figure refer to the minimum amount of memory required for mapping an automaton in theory, i.e., the product of the state number and the STE size. The evaluation results show that the hardware overhead of APmap's mapping is only 2.79%, which is much smaller than Cache Automaton.

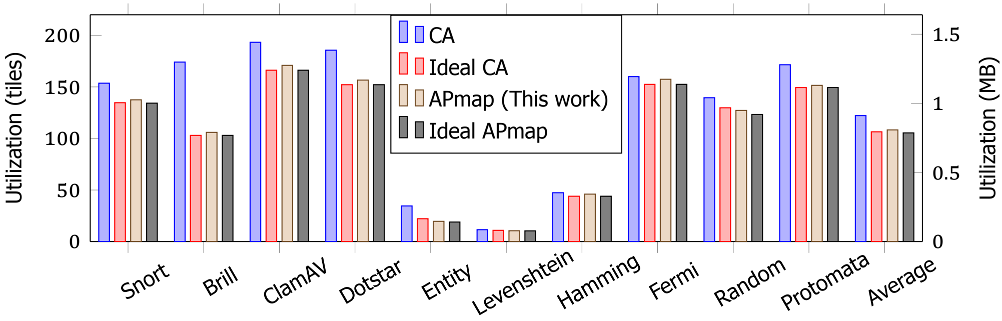


Figure 4.2: Utilization comparison between APmap and Cache Automaton.

4.3. Evaluation

In this chapter, we presented a synthesis flow for CIM architecture and a compiler for automata processors. The synthesis flow generates hardware designs targeting performance optimization. The automata compiler optimizes the storage utilization and outperforms the state of the art. The work presented in this chapter can be further improved on the following aspects:

- The partition algorithm of APmap can be improved. APmap calls METIS for partitioning an automaton into smaller parts. METIS tries to cut the least edges in the partitioning. A cutting edge will be mapped to a global wire in most cases. However, in other cases, several edges can be mapped to the same global wire. For example, if State 1 in Tile 1 activates State 2 and 3 in Tile2, only one global wire is needed between Tile 1 and 2. This type of optimization is not considered by METIS. Therefore, the partitioning result is not optimal, which affects the final compilation result. If the partition algorithm is improved, the quality of APmap would be improved as well.

- We can extend automata simulation tools to utilize the compilation result of APmap. VASim simulates the execution of automata without considering hardware details [125]. Theoretically, it is possible to extend VASim with some hardware details such as the tiles and the switching network. In this way, it can simulate automata processing more precisely, and estimate the energy consumption with respect to different automata testbenches.

- We can further improve the compilation quality by optimizing the mathematical model of automata. VASim has implemented a left minimization algorithm that combines functionally identical elements to reduce the size of the automata [125]. Other algorithms that try to minimize the automata size are also proposed [126]. Applying these algorithms before the partitioning and mapping steps of APmap may further decrease the storage utilization of the applications.

- The generation of automata is also worth investigating. The input of APmap is the automata that describe the application. However, it is not a trivial task to obtain such automata. High-level languages such as RAPID [127] can be used to write applications and compile them into automata. The compiler that works at this level has more optimization chances, such as to reduce the size of the automata. Therefore, the ideal compiler should transfer the user's application into hardware configuration, and conduct various optimization at different stages of the transformation.

5 Conclusion

This chapter summarizes the overall contributions of this dissertation and discusses some future research directions. Section 5.1 presents a summary of the main conclusions of this dissertation. Thereafter, Section 5.2 highlights possible future research directions.

5.1. Summary

Chapter 1, ”Introduction”, briefly introduced in-memory computing with memristive devices. It first presented the motivation of this research field. Thereafter, it surveyed the state of the art with respect to device, circuit, architecture, design automation, and application. It also discussed the opportunities and challenges at each level. Finally, it proposed the contributions of this dissertation, which addresses some of the major circuit, architecture, and design automation challenges.

Chapter 2, ”Circuit Level”, presented two logical operation schemes based on memristive devices. The first one uses resistive sensing to perform logical operations. It modifies the sense amplifier so that it can compare the overall current with references and output the logical operation result. During sensing, the resistance of memristive devices remains unchanged. Therefore, such operations don’t affect the lifetime and the scheme helps in the cases where memristors have a low endurance. The second scheme proposed in this chapter was the enhanced version of the first one. It uses two different sensing paths for AND and OR operations. In this way, the correctness of logic operations can be guaranteed even if large resistance variation exists in memristive devices.

Chapter 3, ”Architecture Level”, presented three computation-in-memory architectures based on memristive devices. The first one was a heterogeneous architecture containing Scouting Logic to accelerate vector bit-wise logical operations. Its performance-energy efficiency is $10\times$ higher than a multi-core system. The second architecture was developed to accelerate automata. In this architecture, memristive memory arrays store configuration information and conduct computation as well. This architecture outperforms similar ones that are built with conventional memory technologies. The third architecture was an improved version of the second one. It breaks the routing network into multiple pipeline stages, each processing a different input sequence. In this way, the architecture achieves a higher throughput with a negligible area overhead. Compared with the-state-of-the-art automata accelerators, this design has the highest performance with the smallest area.

Chapter 4, ”Design Automation”, presented a synthesis flow for computation-in-memory architectures and a compiler for automata processors. The synthesis flow was proposed based on the concept of skeletons, which relates an algorithmic structure to a pre-defined solution template. This solution template contains scheduling, placement, and routing information which is needed for hardware generation. After the user rewrites the algorithm using skeletons, the tool generates the desired circuit by instantiating the solution template. The automata processor compiler generates configuration bits according to the input automata. It uses multiple strategies to transform given automata, so that constraint conflicts can be resolved automatically. It also optimizes the mapping for storage utilization. Evaluation with a standard benchmark suite shows that the compiling results of the proposed compiler occupy less storage than the state of the art.

5.2. Future Research Directions

Several recommendations are suggested to improve the state of the art further. They are organized by the different research topics as listed below.

- **Circuit Level**

    1. Develop circuits with more functionality. The operation types of current circuits that are based on memristive devices are still limited, especially regarding arithmetic operations. To avoid time and energy penalties from memory accesses, the accelerator should be able to process coarse-grained pieces of algorithms. If this piece contains operations that are not supported by memristive devices, data has to be send to the CPU for processing. This breaks the in-memory computing paradigm.
    2. Develop more practical circuits. To continuously advance the memristive device technology, participation from the industry is crucial. Companies will invest more resources in this field if they can develop profitable products based on memristive devices. The precondition of commercializing this technology is to produce reliable devices and circuits. It means that we should consider not only theoretical models, but also practical factors such as yield, variation, and robustness.
    3. Develop circuits for neuromorphic computing. The features of memristive devices, such as multi-level and continuous resistance changing, are similar to the nature of neurons. Therefore, memristive devices have the potential to conduct neuromorphic computing efficiently [65, 66]. We need to investigate various implementations of the neuron, the MAC, and other core components. Compared with conventional technologies, neural networks based on memristive devices are more compact and consume less power.

- **Architecture Level**

    1. Explore more architecture styles. Automata processor is a nice example that demonstrates that simple components can be used to implement complex algorithms. More designs like this are needed. The circuits that are based on memristive devices are not as powerful as CPUs; however, they are more flexible and can work in parallel. It provides more possibilities for building novel architectures.
    2. Investigate in-memory communication. In-memory computing can alleviate the bottleneck between the CPU and the memory. However, as in-memory computing architectures become more and more complex, they also have to be divided into parts. Ideally, these parts should operate independently on different data. At least for some applications data has to be exchanged among different parts. This data exchange may become a bottleneck as well. Therefore, we need to investigate such problems and design better in-memory communication infrastructure.

3. Design artificial intelligence (AI) chips. Neuromorphic circuits and classical CPUs can be integrated to implement AI applications [128]. Innovations and sophisticated optimizations are essential for designing such chips. These memristive device based chips are promising to replace GPUs, which are widely used in current AI systems and power-hungry. The memristive AI chips will be more portable and more energy efficient. This will make AI even more popular in our lives.

- **Design Automation**

  1. Synthesis for robustness. Current synthesis tools only consider the logical aspect, i.e., generating connected gates based on the algorithmic input. However, more factors have to be considered for generating practical circuits. The synthesis tool should be aware of the characteristics of the memristive devices and the user's requirement on the robustness. Thus, it can output the gates with detailed design, such as the width and length of each device.
  2. Develop application profilers. Developing applications targeting in-memory computing architectures requires efforts such as modifying legacy kernels. It would be desirable that the users can estimate the performance before mapping the application to in-memory computing architectures. This can be achieved with the help of profilers. They can analyze the application and recognize the parts that have the potential for acceleration. However, it is not an easy task to develop such profilers. Recognizing specific algorithmic structures requires sophisticated methodologies.
  3. Develop compilers for AI chips. The compilation for AI chips is different from that for CPUs due to the differences of the hardware resources and their constraints [129]. The compilation involves techniques such as graph partitioning and integer linear programming. An optimized compiler can improve the efficiency of application execution significantly.

References

[1] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru, *High performance pattern matching using the automata processor,* in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016) pp. 1123–1132.

[2] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, *Brill tagging on the micron automata processor,* in *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)* (2015) pp. 236–239.

[3] J. Hu and Y. Zhang, *Discovering the interdisciplinary nature of big data research through social network analysis and visualization,* Scientometrics **112**, 91 (2017).

[4] C. P. Chen and C.-Y. Zhang, *Data-intensive applications, challenges, techniques and technologies: A survey on big data,* Information Sciences **275**, 314 (2014).

[5] M. T. Bohr and I. A. Young, *Cmos scaling trends and beyond,* IEEE Micro **37**, 20 (2017).

[6] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs, L. Xie, N. Wald, S. Joshi, H. M. Elsayed, H. Corporaal, and K. Bertels, *Memristor for computing: Myth or reality?* in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017* (2017) pp. 722–731.

[7] A. Kerber, *Reliability of metal gate / high-k devices and its impact on cmos technology scaling,* MRS Advances **2**, 2973–2982 (2017).

[8] Z. Abbas and M. Olivieri, *Impact of technology scaling on leakage power in nano-scale bulk cmos digital standard cells,* Microelectronics Journal **45**, 179 (2014).

[9] B. Hoefflinger, *The energy crisis,* in *Chips 2020: A Guide to the Future of Nanoelectronics*, edited by B. Hoefflinger (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012) pp. 421–427.

[10] S. Borkar, *Exascale computing - a fact or a fiction?* in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013) pp. 3–3.

[11] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. (Elsevier, Amsterdam, Netherlands, 2017).

[12] G. Yeric, *Ic design after moore's law,* in *2019 IEEE Custom Integrated Circuits Conference (CICC)* (2019) pp. 1–150.

[13] S. H. Fuller and L. I. Millett, *Computing performance: Game over or next level?* Computer **44**, 31 (2011).

[14] K. Rupp, *Microprocessor trend data,* (2018).

[15] D. E. Nikonov and I. A. Young, *Overview of beyond-cmos devices and a uniform methodology for their benchmarking,* Proceedings of the IEEE **101**, 2498 (2013).

[16] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, *A microarchitecture for a superconducting quantum processor,* IEEE Micro **38**, 40 (2018).

[17] C. D. James, J. B. Aimone, N. E. Miner, C. M. Vineyard, F. H. Rothganger, K. D. Carlson, S. A. Mulder, T. J. Draelos, A. Faust, M. J. Marinella, J. H. Naegle, and S. J. Plimpton, *A historical survey of algorithms and hardware architectures for neural-inspired and neuromorphic computing applications,* Biologically Inspired Cognitive Architectures **19**, 49 (2017).

[18] S. Mittal, *A survey of techniques for approximate computing,* ACM Computing Surveys **48** (2016), 10.1145/2893356.

[19] P. Siegl, R. Buchty, and M. Berekovic, *Data-centric computing frontiers: A survey on processing-in-memory,* in *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16 (ACM, New York, NY, USA, 2016) pp. 295–308.

[20] H. A. Du Nguyen, J. Yu, M. Abu Lebdeh, M. Taouil, S. Hamdioui, and F. Catthoor, *A classification of memory-centric computing,* J. Emerg. Technol. Comput. Syst. **16** (2020), 10.1145/3365837.

[21] M. Di Ventra and Y. Pershin, *Memcomputing: A computing paradigm to store and process information on the same physical platform,* 2014 International Workshop on Computational Electronics, IWCE 2014 (2012), 10.1109/IWCE.2014.6865809.

[22] L. O. Chua and S. M. Kang, *Memristive devices and systems,* Proceedings of the IEEE **64**, 209 (1976).

[23] S. Hamdioui, M. Taouil, H. A. D. Nguyen, A. Haron, L. Xie, and K. Bertels, *Memristor: the enabler of computation-in-memory architecture for big-data,* in *2015 International Conference on Memristive Systems (MEMRISYS)* (2015) pp. 1–3.

[24] A. Chen, *A review of emerging non-volatile memory (nvm) technologies and applications,* Solid-State Electronics **125**, 25 (2016), extended papers selected from ESSDERC 2015.

[25] S. Hamdioui, *Computation in memory for data-intensive applications: Beyond cmos and beyond von- neumann,* in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15 (ACM, New York, NY, USA, 2015) pp. 1–1.

[26] L. O. Chua, *Memristor-the missing circuit element,* Circuit Theory, IEEE Transactions on **18**, 507 (1971).

[27] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, *The missing memristor found,* nature **453**, 80 (2008).

[28] R. Waser, S. Menzel, and V. Rana, *Recent progress in redox-based resistive switching,* in *2012 IEEE International Symposium on Circuits and Systems (ISCAS)* (2012) pp. 1596–1599.

[29] J. J. Yang, D. B. Strukov, and D. R. Stewart, *Memristive devices for computing,* Nature nanotechnology **8**, 13 (2013).

[30] S. Kvatinsky, N. Wald, G. Satat, A. Kolodny, U. C. Weiser, and E. G. Friedman, *MRL - memristor ratioed logic,* in *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications* (2012) pp. 1–6.

[31] J. Borghetti, Z. Li, J. Straznicky, X. Li, D. A. Ohlberg, W. Wu, D. R. Stewart, and R. S. Williams, *A hybrid nanomemristor/transistor logic circuit capable of self-programming,* Proceedings of the National Academy of Sciences **106**, 1699 (2009).

[32] G. Rose, J. Rajendran, H. Manem, R. Karri, and R. Pino, *Leveraging memristive systems in the construction of digital logic circuits,* Proceedings of the IEEE **100**, 2033 (2012).

[33] L. Gao, F. Alibart, and D. B. Strukov, *Programmable cmos/memristor threshold logic,* IEEE Transactions on Nanotechnology **12**, 115 (2013).

[34] I. Vourkas and G. Sirakoulis, *A novel design and modeling paradigm for memristor-based crossbar circuits,* Nanotechnology, IEEE Transactions on **11**, 1151 (2012).

[35] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, and R. Waser, *Beyond von neumann - logic operations in passive crossbar arrays alongside memory operations,* Nanotechnology **23**, 305205 (2012).

[36] G. Snider, *Computing with hysteretic resistor crossbars,* Applied Physics A **80**, 1165 (2005).

[37] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, *'memristive' switches enable 'stateful' logic operations via material implication,* Nature **464**, 873 (2010).

[38] H. A. D. Nguyen, J. Yu, L. Xie, M. Taouil, S. Hamdioui, and D. Fey, *Memristive devices for computing: Beyond cmos and beyond von neumann,* in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (2017) pp. 1–10.

[39] 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*, DAC '16 (ACM, New York, NY, USA, 2016) pp. 173:1–173:6.

[40] G. Papandroulidakis, I. Vourkas, N. Vasileiadis, and G. C. Sirakoulis, *Boolean logic operations and computing circuits based on memristors,* IEEE Transactions on Circuits and Systems II: Express Briefs **61**, 972 (2014).

[41] T. You, Y. Shuai, W. Luo, N. Du, D. Bürger, I. Skorupa, R. Hübner, S. Henker, C. Mayr, R. Schüffny, T. Mikolajick, O. G. Schmidt, and H. Schmidt, *Exploiting memristive bifeo3 bilayer structures for compact sequential logics,* Advanced Functional Materials **24**, 3357 (2014), https://onlinelibrary.wiley.com/doi/pdf/10.1002/adfm.201303365 .

[42] S. Balatti, S. Ambrogio, and D. Ielmini, *Normally-off logic based on resistive switches - part i: Logic gates,* IEEE Transactions on Electron Devices **62**, 1831 (2015).

[43] E. Lehtonen, J. H. Poikonen, and M. Laiho, *Memristive stateful logic,* in *Memristor Networks*, edited by A. Adamatzky and L. Chua (Springer International Publishing, Cham, 2014) pp. 603–623.

[44] 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 **61**, 895 (2014).

[45] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, *Boolean logic gate exploration for memristor crossbar,* in *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)* (IEEE, 2016) pp. 1–6.

[46] G. Papandroulidakis, I. Vourkas, A. Abusleme, G. Sirakoulis, and A. Rubio, *Crossbar-based memristive logic-in-memory architecture,* IEEE Transactions on Nanotechnology **PP**, 1 (2017).

[47] J. Müller, T. S. Böscke, S. Müller, E. Yurchuk, P. Polakowski, J. Paul, D. Martin, T. Schenk, K. Khullar, A. Kersch, W. Weinreich, S. Riedel, K. Seidel,

A. Kumar, T. M. Arruda, S. V. Kalinin, T. Schlösser, R. Boschke, R. van Bentum, U. Schröder, and T. Mikolajick, *Ferroelectric hafnium oxide: A cmos-compatible and highly scalable approach to future ferroelectric memories,* in *2013 IEEE International Electron Devices Meeting* (2013) pp. 10.8.1–10.8.4.

[48] P. Noé, C. Vallée, F. Hippert, F. Fillot, and J.-Y. Raty, *Phase-change materials for non-volatile memory devices: from technological challenges to materials science issues,* Semiconductor Science and Technology **33**, 013002 (2017).

[49] J. Zhu, *Magnetoresistive random access memory: The path to competitiveness and scalability,* Proceedings of the IEEE **96**, 1786 (2008).

[50] H.-S. P. Wong, C. Ahn, J. Cao, H.-Y. Chen, S. B. Eryilmaz, S. W. Fong, J. A. Incorvia, H. L. Z. Jiang, C. Neumann, K. Okabe, S. Qin, J. Sohn, Y. Wu, S. Yu, and X. Zheng, *Stanford memory trends,* (2019), accessed October 6, 2019.

[51] R. Waser, D. Ielmini, H. Akinaga, H. Shima, H.-S. P. Wong, J. J. Yang, and S. Yu, *Introduction to nanoionic elements for information technology,* in *Resistive Switching* (John Wiley & Sons, Ltd, 2016) Chap. 1, pp. 1–30, https://onlinelibrary.wiley.com/doi/pdf/10.1002/9783527680870.ch1 .

[52] E. Linn, M. Di Ventra, and Y. V. Pershin, *Reram cells in the framework of two-terminal devices,* in *Resistive Switching* (John Wiley & Sons, Ltd, 2016) Chap. 2, pp. 31–48, https://onlinelibrary.wiley.com/doi/pdf/10.1002/9783527680870.ch2 .

[53] S. Yu and P. Y. Chen, *Emerging memory technologies: Recent trends and prospects,* IEEE Solid-State Circuits Magazine **8**, 43 (2016).

[54] W. H. Chen, W. J. Lin, L. Y. Lai, S. Li, C. H. Hsu, H. T. Lin, H. Y. Lee, J. W. Su, Y. Xie, S. S. Sheu, and M. F. Chang, *A 16mb dual-mode reram macro with sub-14ns computing-in-memory and memory functions enabled by self-write termination scheme,* in *2017 IEEE International Electron Devices Meeting (IEDM)* (2017) pp. 28.2.1–28.2.4.

[55] I. Valov, *Interfacial interactions and their impact on redox-based resistive switching memories (rerams),* Semiconductor Science and Technology **32**, 093006 (2017).

[56] R. S. Williams, *How we found the missing memristor,* IEEE Spectrum **45**, 28 (2008).

[57] H. S. P. Wong, H. Y. Lee, S. Yu, Y. S. Chen, Y. Wu, P. S. Chen, B. Lee, F. T. Chen, and M. J. Tsai, *Metal-oxide rram,* Proceedings of the IEEE **100**, 1951 (2012).

[58] B. J. Choi, A. C. Torrezan, K. J. Norris, F. Miao, J. P. Strachan, M.-X. Zhang, D. A. Ohlberg, N. P. Kobayashi, J. J. Yang, and R. S. Williams, *Electrical performance and scalability of pt dispersed sio2 nanometallic resistance switch,* Nano letters **13**, 3213 (2013).

[59] Y. Hayakawa, A. Himeno, R. Yasuhara, W. Boullart, E. Vecchio, T. Vandeweyer, T. Witters, D. Crotti, M. Jurczak, S. Fujii, S. Ito, Y. Kawashima, Y. Ikeda, A. Kawahara, K. Kawai, Z. Wei, S. Muraoka, K. Shimakawa, T. Mikawa, and S. Yoneda, *Highly reliable taox reram with centralized filament for 28-nm embedded application,* in *2015 Symposium on VLSI Circuits (VLSI Circuits)* (2015) pp. T14–T15.

[60] T. Kim and S. Lee, *Evolution of phase-change memory for the storage-class memory and beyond,* IEEE Transactions on Electron Devices **67**, 1394 (2020).

[61] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, *Memristor based computation-in-memory architecture for data-intensive applications,* in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15 (EDA Consortium, San Jose, CA, USA, 2015) pp. 1718–1725.

[62] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, *Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory,* in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016) pp. 27–39.

[63] P. E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. D. Micheli, *The programmable logic-in-memory (plim) computer,* in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)* (2016) pp. 427–432.

[64] D. Fujiki, S. Mahlke, and R. Das, *In-memory data parallel processor,* in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18 (ACM, New York, NY, USA, 2018) pp. 1–14.

[65] D. Kuzum, S. Yu, and H. S. Wong, *Synaptic electronics: materials, devices and applications,* Nanotechnology **24**, 382001 (2013).

[66] M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, *Training andoperation of an integrated neuromorphic network based on metal-oxide memristors,* Nature **521**, 61 (2015).

[67] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, *Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,* in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16 (IEEE Press, Piscataway, NJ, USA, 2016) pp. 14–26.

[68] M. V. Nair and P. Dudek, *Gradient-descent-based learning in memristive crossbar arrays,* in *2015 International Joint Conference on Neural Networks (IJCNN)* (2015) pp. 1–7.

[69] P. M. Sheridan, D. Chao, and W. D. Lu, *Feature extraction using memristor networks,* IEEE Transactions on Neural Networks and Learning Systems **27**, 2327 (2016).

[70] T. F. Wu, B. Q. Le, R. Radway, A. Bartolo, W. Hwang, S. Jeong, H. Li, P. Tandon, E. Vianello, P. Vivet, E. Nowak, M. K. Wootters, H. . P. Wong, M. M. S. Aly, E. Beigne, and S. Mitra, *14.3 a 43pj/cycle non-volatile microcontroller with 4.7us shutdown/wake-up integrating 2.3-bit/cell resistive ram and resilience techniques,* in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)* (2019) pp. 226–228.

[71] S. Balatti, S. Ambrogio, Z. Wang, and D. Ielmini, *True random number generation by variability of resistive switching in oxide-based devices,* IEEE Journal on Emerging and Selected Topics in Circuits and Systems **5**, 214 (2015).

[72] R. Liu, H. Wu, Y. Pang, H. Qian, and S. Yu, *Experimental characterization of physical unclonable function based on 1 kb resistive random access memory arrays,* IEEE Electron Device Letters **36**, 1380 (2015).

[73] G. Sun, J. Zhao, M. Poremba, C. Xu, and Y. Xie, *Memory that never forgets: emerging nonvolatile memory and the implication for architecture design,* National Science Review , nwx082 (2017).

[74] G. Adam, A. Khiat, and T. Prodromakis, *Challenges hindering memristive neuromorphic hardware from going mainstream,* Nature Communication (2018).

[75] L. Thomas, G. Jan, J. Zhu, H. Liu, Y.-J. Lee, S. Le, R.-Y. Tong, K. Pi, Y.-J. Wang, D. Shen, R. He, J. Haq, J. Teng, V. Lam, K. Huang, T. Zhong, T. Torng, and P.-K. Wang, *Perpendicular spin transfer torque magnetic random access memories with high spin torque efficiency and thermal stability for embedded applications (invited),* Journal of Applied Physics **115**, 172615 (2014), https://doi.org/10.1063/1.4870917 .

[76] L. Xie, H. A. D. Nguyen, J. Yu, A. Kaichouhi, M. Taouil, M. AlFailakawi, and S. Hamdioui, *Scouting logic: A novel memristor-based logic design for resistive computing,* in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2017) pp. 176–181.

[77] J. Yu, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, and S. Hamdioui, *Enhanced scouting logic: a robust memristive logic design scheme,* in *2019 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)* (2019) pp. 1–6.

[78] J. Yu, H. A. Du Nguyen, L. Xie, M. Taouil, and S. Hamdioui, *Memristive devices for computation-in-memory,* in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)* (IEEE, 2018) pp. 1646–1651.

[79] J. Yu, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, and S. Hamdioui, *Time-division multiplexing automata processor,* in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)* (IEEE, 2019) pp. 794–799.

[80] J. Yu, R. Nane, A. Haron, S. Hamdioui, H. Corporaal, and K. Bertels, *Skeleton-based design and simulation flow for computation-in-memory architectures,* in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)* (2016) pp. 165–170.

[81] J. Yu, R. Nane, I. Ashraf, M. Taouil, S. Hamdioui, H. Corporaal, and K. Bertels, *Skeleton-based synthesis flow for computation-in-memory architectures,* IEEE Transactions on Emerging Topics in Computing , 1 (2017).

[82] A. Fantini, L. Goux, R. Degraeve, D. J. Wouters, N. Raghavan, G. Kar, A. Belmonte, Y. Y. Chen, B. Govoreanu, and M. Jurczak, *Intrinsic switching variability in hfo2 rram,* in *2013 5th IEEE International Memory Workshop* (2013) pp. 30–33.

[83] W. Kang, H. Wang, Z. Wang, Y. Zhang, and W. Zhao, *In-memory processing paradigm for bitwise logic operations in stt-mram,* IEEE Transactions on Magnetics **53**, 1 (2017).

[84] M. Imani, Y. Kim, and T. Rosing, *Mpim: Multi-purpose in-memory processing using configurable resistive memory,* in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)* (2017) pp. 757–763.

[85] F. Parveen, Z. He, S. Angizi, and D. Fan, *Hielm: Highly flexible in-memory computing using stt mram,* in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)* (2018) pp. 361–366.

[86] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, *Computing in memory with spin-transfer torque magnetic ram,* IEEE Transactions on Very Large Scale Integration (VLSI) Systems **26**, 470 (2018).

[87] F. Parveen, S. Angizi, Z. He, and D. Fan, *Imcs2: Novel device-to-architecture co-design for low-power in-memory computing platform using coterminous spin switch,* IEEE Transactions on Magnetics , 1 (2018).

[88] R. Han, P. Huang, Y. Zhao, Z. Chen, L. Liu, X. Liu, and J. Kang, *Demonstration of logic operations in high-performance rram crossbar array fabricated by atomic layer deposition technique,* Nanoscale Research Letters **12**, 37 (2017).

[89] X. Hong, P. A. Dananjaya, S. Krishnia, W. Gan, D. J. Loy, F. Tan, N. CheeMang, and W. S. Lew, *A novel geometry of ecm-based rram with improved variability,* Journal of Physics D: Applied Physics (2018).

[90] Y. Fang, Z. Yu, Z. Wang, T. Zhang, Y. Yang, Y. Cai, and R. Huang, *Improvement of hfox-based rram device variation by inserting ald tin buffer layer,* IEEE Electron Device Letters **39**, 819 (2018).

[91] A. Mehonic, M. Munde, W. Ng, M. Buckwell, L. Montesi, M. Bosman, A. Shluger, and A. Kenyon, *Intrinsic resistance switching in amorphous silicon oxide for high performance siox reram devices,* Microelectronic Engineering **178**, 98 (2017), special issue of Insulating Films on Semiconductors (INFOS 2017).

[92] A. Bricalli, E. Ambrosi, M. Laudato, M. Maestro, R. Rodriguez, and D. Ielmini, *Siox-based resistive switching memory (rram) for crossbar storage/select elements with high on/off ratio,* in *2016 IEEE International Electron Devices Meeting (IEDM)* (2016) pp. 4.3.1–4.3.4.

[93] H. Lv, X. Xu, P. Yuan, D. Dong, T. Gong, J. Liu, Z. Yu, P. Huang, K. Zhang, C. Huo, C. Chen, Y. Xie, Q. Luo, S. Long, Q. Liu, J. Kang, D. Yang, S. Yin, S. Chiu, and M. Liu, *Beol based rram with one extra-mask for low cost, highly reliable embedded application in 28 nm node and beyond,* in *2017 IEEE International Electron Devices Meeting (IEDM)* (2017) pp. 2.4.1–2.4.4.

[94] S. A. Manavski, *Cuda compatible gpu as an efficient hardware accelerator for aes cryptography,* in *2007 IEEE International Conference on Signal Processing and Communications* (2007) pp. 65–68.

[95] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, *Drisa: A dram-based reconfigurable in-situ accelerator,* in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17 (ACM, New York, NY, USA, 2017) pp. 288–301.

[96] Z. Wang, Y. Liu, A. Lee, F. Su, C. P. Lo, Z. Yuan, J. Li, C. C. Lin, W. H. Chen, H. Y. Chiu, W. E. Lin, Y. C. King, C. J. Lin, P. K. Amiri, K. L. Wang, M. F. Chang, and H. Yang, *A 65-nm reram-enabled nonvolatile processor with time-space domain adaption and self-write-termination achieving* $> 4times$ *faster clock frequency and* $> 6times$ *higher restore speed,* IEEE Journal of Solid-State Circuits **52**, 2769 (2017).

[97] S. Muthulakshmi, C. S. Dash, and S. Prabaharan, *Memristor augmented approximate adders and subtractors for image processing applications: An approach,* AEU - International Journal of Electronics and Communications **91**, 91 (2018).

[98] H. A. D. Nguyen, L. Xie, M. Taouil, R. Nane, S. Hamdioui, and K. Bertels, *Computation-in-memory based parallel adder,* in *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on* (IEEE, 2015) pp. 57–62.

[99] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, *Ac-dimm: Associative computing with stt-mram,* in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13 (ACM, New York, NY, USA, 2013) pp. 189–200.

[100] M. Saecker and V. Markl, *Big data analytics on modern hardware architectures: A technology survey,* in *Business Intelligence: Second European Summer School, eBISS 2012, Brussels, Belgium, July 15-21, 2012, Tutorial Lectures*, edited by M.-A. Aufaure and E. Zimányi (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013) pp. 125–149.

[101] K. Wu, *FastBit: an efficient indexing technology for accelerating data-intensive science,* Journal of Physics: Conference Series **16**, 556 (2005).

[102] R. D. Cameron, T. C. Shermer, A. Shriraman, K. S. Herdy, D. Lin, B. R. Hull, and M. Lin, *Bitwise data parallelism in regular expression matching,* in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14 (ACM, New York, NY, USA, 2014) pp. 139–150.

[103] D. Lavenier, J. Roy, and D. Furodet, *Dna mapping using processor-in-memory architecture,* in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (2016) pp. 1429–1435.

[104] S. Beamer, K. Asanovic, and D. Patterson, *Direction-optimizing breadth-first search,* in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012) pp. 1–10.

[105] M. Roesch, *Snort - lightweight intrusion detection for networks,* in *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99 (USENIX Association, Berkeley, CA, USA, 1999) pp. 229–238.

[106] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, *Fast support for unstructured data processing: The unified automata processor,* in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2015) pp. 533–545.

[107] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch, *Hawk: Hardware support for unstructured log processing,* in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)* (2016) pp. 469–480.

[108] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, *Hare: Hardware accelerator for regular expressions,* in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016) pp. 1–12.

[109] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, *An efficient and scalable semiconductor architecture for parallel automata processing,* IEEE Transactions on Parallel and Distributed Systems **25**, 3088 (2014).

[110] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, *Cache automaton,* in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17 (ACM, New York, NY, USA, 2017) pp. 259–272.

[111] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar, *Resistive gp-simd processing-in-memory,* ACM Trans. Archit. Code Optim. **12**, 57:1 (2016).

[112] R. B. Hur and S. Kvatinsky, *Memristive memory processing unit (mpu) controller for in-memory processing,* in *2016 IEEE International Conference on the Science of Electrical Engineering (ICSEE)* (2016) pp. 1–5.

[113] H. A. Du Nguyen, J. Yu, M. Abu Lebdeh, M. Taouil, and S. Hamdioui, *A computation-in-memory accelerator based on resistive devices,* in *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19 (ACM, Washington, DC, USA, 2019).

[114] J. Wadden, K. Angstadt, and K. Skadron, *Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures,* in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018).

[115] H. A. D. Nguyen, L. Xie, M. Taouil, S. Hamdioui, and K. Bertels, *Synthesizing hdl to memristor technology: A generic framework,* in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)* (2016) pp. 43–48.

[116] F. Riente, G. Turvani, M. Vacca, M. R. Roch, M. Zamboni, and M. Graziano, *Topolinano: a cad tool for nano magnetic logic,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **PP**, 1 (2017).

[117] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, *A mapping methodology of boolean logic circuits on memristor crossbar,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **PP**, 1 (2017).

[118] R. B. Hur, N. Wald, N. Talati, and S. Kvatinsky, *Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic,* in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2017) pp. 225–232.

[119] 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 *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16 (ACM, New York, NY, USA, 2016) pp. 117:1–117:6.

[120] H. A. D. Nguyen, L. Xie, M. Taouil, R. Nane, S. Hamdioui, and K. Bertels, *On the implementation of computation-in-memory parallel adder,* IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**, 2206 (2017).

[121] A. Haron, J. Yu, R. Nane, M. Taouil, S. Hamdioui, and K. Bertels, *Parallel matrix multiplication on memristor-based computation-in-memory architecture,* in *2016 International Conference on High Performance Computing Simulation (HPCS)* (IEEE, 2016) pp. 759–766.

[122] J. Wadden, S. Khan, and K. Skadron, *Automata-to-routing: An open-source toolchain for design-space exploration of spatial automata processing architectures,* in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2017) pp. 180–187.

[123] J. Yu, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, and S. Hamdioui, *Apmap: An open-source compiler for cache automaton,* (2019), submitted.

[124] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, *ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures,* in *2016 IEEE International Symposium on Workload Characterization (IISWC)* (2016) pp. 1–12.

[125] J. Wadden and K. Skadron, *VASim: An open virtual automata simulator for automata processing application and architecture research*, Tech. Rep. (Technical Report CS2016-03, University of Virginia, 2016).

[126] J. Björklund and L. Cleophas, *Aggregation-based minimization of finite state automata,* Acta Informatica (2020), 10.1007/s00236-019-00363-5.

[127] K. Angstadt, W. Weimer, and K. Skadron, *Rapid programming of pattern-recognition processors,* in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16 (Association for Computing Machinery, New York, NY, USA, 2016) p. 593–605.

[128] A. Sebastian, T. Tuma, N. Papandreou, M. Le Gallo, L. Kull, T. Parnell, and E. Eleftheriou, *Temporal correlation detection using computational phase-change memory,* Nature Communications **8**, 1115 (2017).

[129] S. Song, A. Balaji, A. Das, N. Kandasamy, and J. Shackleford, *Compiling spiking neural networks to neuromorphic hardware,* in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '20 (Association for Computing Machinery, New York, NY, USA, 2020) p. 38–50.

A

Publications - Circuit Level

This chapter presents the publications on the circuit level. The following papers are included:

1. L. Xie, H. A. Du Nguyen, **J. Yu**, A. Kaichouhi, M. Taouil, M. Alfailakawi, S. Hamdioui, *Scouting Logic: A Novel Memristor-based Logic Design for Resistive Computing*, IEEE Computer Society Annual Symposium on VLSI (ISVLSI'17), Bochum, Germany, July 2017, pp. 151-156.
2. **J. Yu**, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, S. Hamdioui, *Enhanced Scouting Logic: A Robust Memristive Logic Design Scheme*, The 15th IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'19), Qingdao, China, July 2019, pp. 1-6.

Scouting Logic: A Novel Memristor-Based Logic Design for Resistive Computing

Lei Xie, H.A. Du Nguyen, Jintao Yu, Ali Kaichouhi, Mottaqiallah Taouil, Mohammad AlFailakawi*, Said Hamdioui
Laboratory of Computer Engineering, Delft University of Technology, the Netherlands
*Computer Engineering Department, Kuwait University, Kuwait
Email: {L.Xie,H.A.DuNguyen,J.Yu-1,M.Taouil,S.Hamdioui}@tudelft.nl; alfailakawi.m@ku.edu.kw

***Abstract*—Memristor technology is a promising alternative to CMOS due to its high integration density, near-zero standby power, and ability to implement novel resistive computing. One of the major limitations of these architectures is the limited endurance of memristor devices, especially when a logic gate requires multiple steps/switching to execute the logic operations. To alleviate the endurance requirement and improve the performance, we present a novel logic design style, called *scouting logic* that executes any logic gate by only reading the memristor devices and without changing their states. Hence, no impact on the memristors' endurance. The proposed design is implemented using two styles (current and voltage based). To illustrate the performance of scouting logic based designs, the area, delay, and power consumption are analyzed and compared with state-of-the-art. The results show that scouting logic improves the delay and power consumption by at least a factor of 2.3, while having similar or less area overhead. Finally, we discuss the potential applications and challenges of scouting logic.**

I. Introduction

As CMOS technology is being continuously scaled down towards its physical limits, it suffers from major challenges such as saturated performance improvement, increasing leakage power, etc [1,2]. Emerging technologies, such as memristors, nanotube, graphene transistors [1], are under research as an alternative to CMOS technology. Memristor is one of the most promising candidates due to its great scalability, high integration density, and its near-zero standby power [1,3,4]. Novel memristor-based computing architectures [5–7] have been proposed as an alternative to today's von-Neumann architectures for big-data applications. Preliminary results of these resistive architectures show several orders of magnitude improvement for different metrics such as energy and area efficiency [5,6]. Big-data applications typically need to process large volume of data resulting in frequent device switching which poses as a concern for the endurance-limited memristor technology [3]. As a result, a logic design style with less switching frequency is required.

Recently, three types of memristor-based logic have been proposed; they can be classified into [4]: threshold/majority [8,9], implication [10,11] and Boolean logic [7,12,13]. Since threshold and majority logic use voltages to represent data, they are more applicable to von-Neumann architectures [8,9]. Due to the fact that implication and Boolean logic designs represent data as resistances, it is more efficient to use such approaches in resistive computing architectures [12,13]. However, in such logic designs, a sequence of primitive operations is required to execute a simple logic gate (e.g., XOR) leading to frequent device switching which reduces devices' endurance. Moreover, these logic designs suffer from a considerable delay hence low speed. To operate logic gates with limited endurance, the authors of [7] proposed an approach to implement logic operations using read operations by modifying the sense amplifier. Unlike previous approaches in [7], AND and OR gates are executed in one read operation while the XOR gates executed in two. However, their approach is area-inefficient and requires two steps to execute in the worst case.

In this work, we propose a novel logic design style, called scouting logic to enhance the performance of memristor-based logic circuits by limiting all gate executions to a single read operation. The current through the equivalent gate input resistance or the voltage drop over it is compared with a reference signal. The proposed design style is implemented using two types of sense amplifiers (current and voltage based) and their performance in terms of area, delay and power consumption are investigated. Moreover, designing robust scouting logic in presence of variations is discussed.

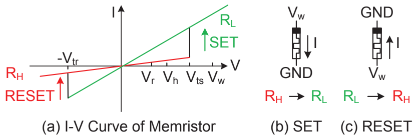
The remainder of this paper is organized as follows. Section II provides a background on memristors and briefly describes the state-of-the-art in memristor-based logic designs. Section III presents the working principle of scouting logic. Section IV verifies the designs of scouting logic using simulations and evaluates their performance. Section V discusses the potential applications and challenges of scouting logic. Finally, Section VI concludes the paper.

II. Background

A. Memristor

Fig. 1 (a) shows the typical current-voltage relation of a bipolar memristor [3]. A memristor switches from one resistive state to another (i.e., a write operation) when the absolute value of the applied voltage (V_w) across the device is greater than its threshold voltage (see Fig. 1(b)-(c)). Otherwise, it stays in its current resistive state. Normally, a memristor requires different switching threshold voltages for SET (V_{ts}) and RESET (V_{tr}) operations where $V_{tr}<V_{ts}$ [3]. Reading the

978-1-5090-6762-6/17/$31.00 © 2017 IEEE



(a) I-V Curve of Memristor (b) SET (c) RESET

Fig. 1: Memristive Behaviour.

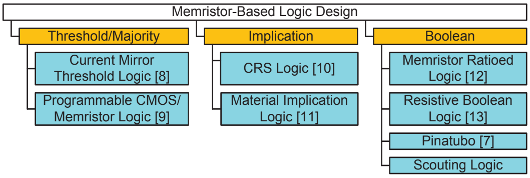

Memristor-Based Logic Design
Threshold/Majority
Current Mirror Threshold Logic [8]
Programmable CMOS/ Memristor Logic [9]
Implication
CRS Logic [10]
Material Implication Logic [11]
Boolean
Memristor Ratioed Logic [12]
Resistive Boolean Logic [13]
Pinatubo [7]
Scouting Logic

Fig. 2: Classification of Memristor-Based Logic

memristor is performed by applying the read voltage V_r; based on the resistance a low or high current will flow through the device. An additional control voltage V_h is required in some of the Boolean and implication logic operations [11,13]. Typically, the control voltages should satisfy the relation: $0<V_r<V_h=V_w/2<V_{\text{tr}} <V_{\text{ts}}<V_w<2V_{\text{tr}}$ (see Fig. 1(a)) [11,13].

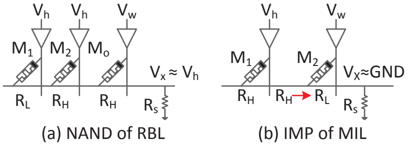
B. Classification of Memristor-Based Logic

Several memristor-based logic designs have been proposed [7–13]. Fig. 2 classifies them into Boolean, implication, and threshold/majority logic [4]. Note that the logic designs using resistance to represent data are suitable for crossbar-based resistive computing. Two of such designs, resistive Boolean logic and material implication logic, will be discussed next as they are the most popular candidates for crossbars.

C. State-of-the-Art Resistive Logic Types

Resistive Boolean Logic (RBL) provides three primitive gates: NOT, AND, and NAND [13]. It uses high resistance R_H and low resistance R_L to represent a logic 1 and 0, respectively. The two-input NAND gate of Fig. 3(a) is used as an example to explain the working principle of RBL. We assume that the inputs are stored on memristor M_1 and M_2, while the output is produced at M_o. The gate requires an extra resistor R_s $(R_L \ll R_s \ll R_H)$. To complete the NAND gate, two steps are performed. First, M_o should be RESET to R_H. Next, control voltages V_h and V_w are applied to the input and output memristors, respectively. In case one of the inputs is 0, the equivalent resistance of M_1 and M_2 is around R_L (see e.g. Fig. 3(a) where M_1=R_L and M_2=R_H). Therefore, the voltage V_x on the nanowire connected to memristors and the resistor is around V_h as $R_L \ll R_s \ll R_H$. As a result, the voltage V_{om} across the output memristor M_o is $V_w - V_\text{x} \approx V_w - V_h = \frac{V_w}{2} < V_{\text{ts}}$, and therefore M_o stays at R_H (logic 1).

Material Implication Logic (MIL) provides a single primitive gate only, which is material implication (IMP) as shown in Fig. 3(b) [11]. MIL uses R_H and R_L to represent logic 0 and 1, respectively. An IMP gate consists of two memristors M_1 and M_2 and a resistor R_s $(R_L \ll R_s \ll R_H)$. M_1 is used as an input, while M_2 is used both as input and output. To perform the IMP operation, the control voltages V_h and V_w should be applied to M_1 and M_2, respectively. In Fig. 3(b), both inputs are assumed to be logic 0 (R_H). After applying V_h and V_w, $V_\text{x} \approx 0$ and the output switches to logic 1. Multiple sequential IMP gates can realize AND, OR, or XOR gates [11].



(a) NAND of RBL (b) IMP of MIL

Fig. 3: Logic Designs Suitable for Resistive Computing

Both RBL and MIL have several shortcomings. First, they require several steps to execute a single gate thereby affecting the delay and power. Second, both require additional CMOS controllers to apply the control voltages and control the sequential steps. This impact the performance of the design. Third, both logic designs require the relative high voltage V_w to program the memristors, and hence, each primitive gate consumes more power/energy as compared to having read operations only [3]. Forth, both designs need to switch the output memristors frequently, and therefore, the entire design is strongly limited by the endurance [1,3]. This motivates us to develop scouting logic as described in the next section.

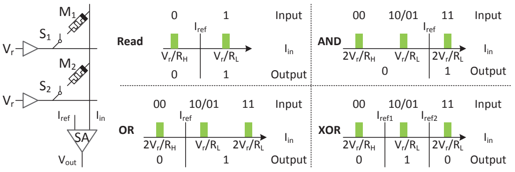
III. Scouting Logic

This section first describes the main idea of scouting logic. Subsequently, it presents two designs of the sense amplifier used to implement scouting logic.

A. Main Idea

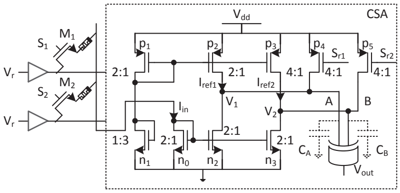
Scouting logic performs its logic operations by modifying the read operation. Fig. 4(a) shows a resistive memory based on 1T1R cells. Normally when a cell is read, say for example Memristor M_1, a read voltage V_r is applied to its row and the switch S_1 is activated. Subsequently, a current I_{in} will flow through the bit line to the input of the sense amplifier (SA). This current is compared to the reference current I_{ref}. If I_{in} is greater than I_{ref} (i.e., when M_1 is R_L state), the output of the SA changes to V_{dd} (logic 1). Similarly, when M_1 is R_H state, I_{in}<I_{ref}, and the output changes to logic 0. For proper operations, I_{ref} should be fixed between high and low currents of Fig. 4(b).

Inspired by this read operation, we demonstrate how to implement OR, AND and XOR scouting logic gates, which are frequently used in bitwise logic operations [14,15]. Instead of reading a single memristor at a time, scouting logic activates the two inputs of the gate simultaneously (e.g., M_1 and M_2 in Fig. 4(a)). As a result, the input current to the sense amplifier is determined by the equivalent input resistance $(M_1//M_2)$. This resistance results in three possible values: $\frac{R_L}{2}$, $\frac{R_H}{2}$ and $R_L//R_H \approx R_L$. Hence, the input current I_{in} also can have only three values. By changing the value of I_{ref} different gates can be realized. To implement an OR gate, I_{ref} should be fixed between $\frac{2V_r}{R_H}$ and $\frac{V_r}{R_L}$ as depicted in



(a) Memory (b) References of Primitive Operations

Fig. 4: Main Idea of Scouting Logic



Switch Configurations

Operation	S_{r1}	S_{r2}
OR/Read	OFF	ON
AND	ON	OFF
XOR	OFF	OFF

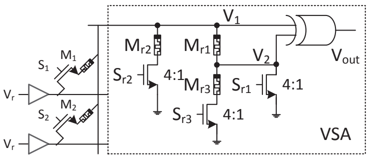
Fig. 5: Current-based SA

Fig. 4(b)). As a result, only when $R_1//R_2{=}\frac{R_H}{2}$ the output is 0. Similarly, to implement an AND operation, I$_{\text{ref}}$ should be fixed between $\frac{2V_r}{R_L}$ and $\frac{V_r}{R_L}$. The XOR operation needs two references and only when $R_1//R_2{\approx}R_L$ the output is logic 1. Note that it is also possible to support multi-fanin logic gates by setting proper reference currents.

The above implies that reading logic circuit should satisfy two requirements. First, it should be operational with a single or two references. Second, it should support a reconfigurable reference signal. Different from Pinatubo [7], scouting logic uses sense amplifiers with a lower delay and smaller area. Next, we describe the two SA designs that satisfy the above requirements.

B. Sense Amplifier Design

We propose two SA designs that both satisfy the two requirements: current-based SA (CSA) shown in Fig. 5, and voltage-based SA (VSA) shown in Fig. 6. CSA generates its reference current using transistors p$_1$ and n$_1$. This reference current is duplicated via p$_2$ and p$_3$ using PMOS current mirrors. Note that the value of I$_{\text{ref2}}$ is twice larger than that of I$_{\text{ref1}}$ as p$_3$ has a double size. The input current I$_{\text{in}}$ is also mirrored twice through n$_2$ and n$_3$. The two pairs of transistors p$_2$-n$_2$ and p$_3$-n$_3$ implement at the same time two current comparators [16]; they compare the two reference currents (i.e., I$_{\text{ref1}}$ and I$_{\text{ref2}}$) with the input current (I$_{\text{in}}$). Transistors p$_4$ and p$_5$ determine the logic operation and decide which reference currents are enabled. The table on the right side of Fig. 5 summarizes how p$_4$ and p$_5$ are configured for the different gates. For instance, when an XOR is performed both p$_4$ and p$_5$ are turned off, i.e. both I$_{\text{ref1}}$ and I$_{\text{ref2}}$ are considered for the comparison. Assume for this gate that M$_1$ is R_H and M$_2$ is R_L. Hence, I$_{\text{ref1}}$<I$_{\text{in}}$<I$_{\text{ref2}}$ (see right bottom of Fig. 4(b)). The parasitic capacitor C$_\text{A}$ of input A is discharged to ground as I$_{\text{ref1}}$<I$_{\text{in}}$, while C$_\text{B}$ is charged to V_{dd} as I$_{\text{in}}$<I$_{\text{ref2}}$. As a result, the output voltage V_{out} is V_{dd}. The AND and OR gates work in a similar way.



Switch Configurations

Operation	S_{r1}	S_{r2}	S_{r3}
OR/Read	ON	OFF	OFF
AND	ON	ON	OFF
XOR	OFF	OFF	ON

M$_{r1}$ = 2R$_\text{L}$
M$_{r2}$ = R$_\text{L}$
M$_{r3}$ = 2.5R$_\text{L}$

Fig. 6: Voltage-based SA

Fig. 6 shows the VSA. It consists of three reference memristors (M$_{\text{ri}}$, $1{\leq}i{\leq}3$), three switches (S_{ri}, $1{\leq}i{\leq}3$), and an XOR gate. The switches are used to select the reference memristors, while the XOR gate is used as a threshold function. The table on the right part of Fig. 6 shows how the switches are configured for different gates. For instance, to perform an XOR operation, S_{r1} and S_{r2} are turned off while S_{r3} is turned on. After switches S_1 and S_2 are turned on, the input (M$_1$ and M$_2$) and the reference memristors form a voltage divider. Let us assume that M$_1$ is R_H and M$_2$ is R_L for the XOR gate. By setting M$_{\text{r1}}$ and M$_{\text{r3}}$ to appropriate values (e.g., M$_{\text{r1}}$=$2R_L$ and M$_{\text{r3}}$=$2.5R_L$), the voltage V_1 will be greater than the threshold voltage of a transistor while V_2 will be less than the threshold voltage. As a result, the output V_{out} approximates 0V. Note that the reference memristors only need to be programed once and may be implemented by resistors.

IV. Evaluation

This section first verifies scouting logic (SL). Thereafter, it provides a comparison with state-of-the-art.

A. Design Verification

The SL gate simulations are verified with Cadence Spectre. The simulation model consists of the 1T1R array of Fig. 4(a) connected to either CSA or VSA; both SAs have been verified. The 1T1R array and SAs are described by a SPICE netlist, while the memristor model [17], CMOS controller and voltage drivers by Verilog-A modules. The simulation parameters are extracted from references [11,13,18] and summarized in Table I. The transistor sizes of the CSA and VSA designs are depicted in Figs. 5 and 6, respectively and are simulated with the PTM 90nm [19] library. Here, we assume that all the memristors already store same data (in 1T1R array).

The AND, OR and XOR gates are verified for all the possible memristor states. Fig. 7(a) and (b) show an example of waveforms for the XOR gate based on CSA and VSA, respectively. Here, the states $M_1{=}R_H$ (logic 0) and $M_2{=}R_L$ (logic 1) are simulated. To evaluate the output of the XOR gate, the switches S_1 and S_2 of Fig. 5 should be turned on to read the memristors M_1 and M_2. To configure the CSA of Fig. 5 as an XOR gate, the switches S_{r1} and S_{r2} should be turned off. As a result, the voltage of node V_1 of CSA is 0.256V<$V_{\text{th,cmos}}$=0.45V while the voltage of node V_2 is 0.59V>$V_{\text{th,cmos}}$=0.45V (see also Fig. 5). The final output V_{out} of the XOR gate is 0.894V (logic 1). VSA-based XOR gate

TABLE I: Parameters

Parameter	Description	Value
	Technology	
	Memristor (TaO_x) [1,18]	
F (nm)	Feature size	90
V_{ts} (V)	Threshold voltage for SET	1.17
V_{tr} (V)	Threshold voltage for RESET	1.06
R_L ($k\Omega$)	Low resistance	200
R_H ($M\Omega$)	High resistance	10
A_{cell} (μm^2)	Area of a 1T1R cell	0.0486
T_{sw}(ns)	Switching time (max of SET and RESET)	1.71
	CMOS	
	UMC 90nm Library	
	Design [11,13]	
N_R	No. of rows in 1T1R array	128
N_C	No. of columns in 1T1R array	32
V_w (V)	Program voltage	1.6
V_h (V)	Half-select voltage	0.8
V_r (V)	Read voltage	0.9
V_{dd} (V)	CMOS power supply	0.9

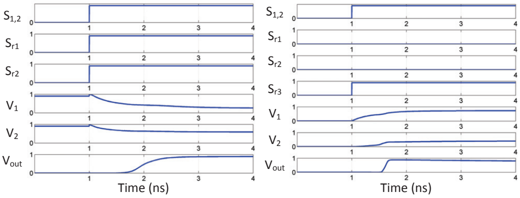
works in a similar way as shown in Fig. 7(b). Note that the VSA based design is faster than the CSA design.

B. Comparison with the State-of-the-art

SL gates (i.e., AND, OR and XOR gates) both based on CSA (SL_CSA) and on VSA (SL_VSA) are compared to RBL and MIL gates in terms of delay, power consumption and area using a 128x32 1T1R memory array. The considered components are the 1T1R memory array, the CMOS controller and SAs. For RBL and MIL, additional memristors required to store intermediate results and implement R_s as shown in Fig. 3 are also considered. The delay and power consumption of the 1T1R memory array are obtained from Cadence Spectre while the area of a single 1T1R cell is taken from ITRS [1]. CMOS controllers are synthesized and evaluated using Cadence RTL Compiler with UMC 90nm library. The delay and power consumption of the SAs are obtained from Cadence Spectre while the area of a single SA is obtained from Cadence Virtuoso using Cadence 90nm PDK [20]. For the modified SA used by SL, only the additional transistors are considered as compared to the current SA of [21].

Fig. 8 presents the results of the comparison:

- **Delay**: CSA-based SL has the lowest delay to execute a single step as well as the lowest total delay (see Fig. 8(a) and (b)); its total delay is at least 2.48 times shorter than RBL and MIL. Although RBL and MIL have a shorter delay per step than VSA-based SL, their total delay is longer as they need multiple steps to execute a logic operation.
- **Power**: VSA-based SL consumes the lowest power for all the gates; it consumes at least 2.36 times less power than RBL and MIL as it can execute a logic operation in a single step. In addition, the controller dominates the power consumed by RBL and MIL, while the SAs dominate the power consumption for SL.
- **Area**: SL does not need more area than RBL and MIL. Although CSA- and VSA-based SL need additional area for the modified SA, it utilizes a simple CMOS controller. In contrast, RBL and MIL require additional area for their more complex controller as they need several steps to execute the gate. The total additional area of the extra cells for MIL and RBL is negligible.



(a) CSA-Based XOR (b) VSA-Based XOR

Fig. 7: SPICE Simulation Results

The comparison results clearly show that SL outperforms RBL and MIL in terms of delay and power consumption while having the same or less area overhead.

Fig. 9 compares SL with Pinatubo [7] in terms of delay and area. However, as the authors of [7] provided a design without implementation details (i.e., $\frac{W}{L}$ ratio of the transistors), we use the number of computation steps as the delay metric and the number of required transistors and memristors as the area metric. Compared to Pinatubo, SL requires only one step to execute the XOR gate. All other gates can be executed in a single step. In addition, the current- and voltage-based sense amplifiers require much less transistors (40% and 64%, respectively). Therefore, SL is potentially faster with a lower area overhead than Pinatubo.

V. Discussion

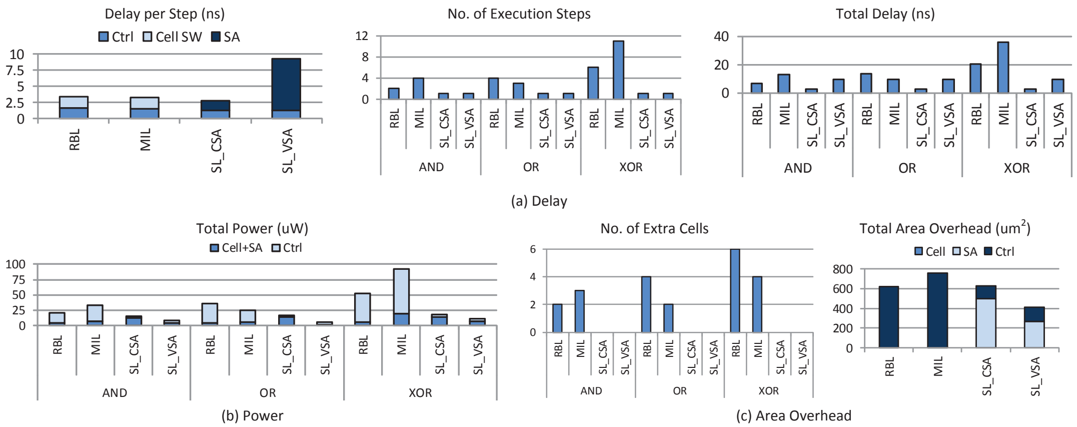
This section discusses potential applications and some challenges and solutions of scouting logic design.

A. Potential Applications

SL can implement many bitwise logic operations, such as AND, OR and XOR, in a very efficient manner as pointed out earlier. Such logic operations are frequently used in many data-intensive applications such as database queries [15], graph processing [14], etc. In traditional settings, these applications need to transfer data between the processor and memory to perform these bitwise logic operations. The movement of such large amount of data input/output of the memory results in considerable amount delay and energy overhead [14,15]. Since SL can directly perform logic operations within the memory array, it can eliminate the need for data movement between processors and memory, thus, significantly reducing execution time and energy consumption. Moreover, scouting logic based memories have very good scaling characteristics due to the fact that only memory controller need to be adapted.

B. Challenges

Memristor technology has been being extensively studied for the past few years due to its applicability in logic and memory designs [4]. Memristor devices have been implemented using different material and cell structures, but nevertheless all suffer from the same two major challenges, namely, limited cell endurance and device variability [4,22]. Performing logic operations using scouting logic requires only reading memristors' states as compared to other memristor-based logic designs that require switching between resistive states.

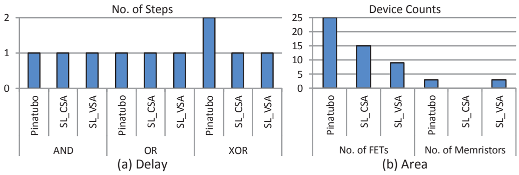


(a) Delay

(b) Power

(c) Area Overhead

Fig. 8: Comparison Between Different Logic Styles



(a) Delay

(b) Area

Fig. 9: Comparison with Pinatubo

Requiring only read operations makes scouting logic designs have better endurance as compared to other approaches resulting in improved device lifetime.

Similar to all memristor based logic, scouting logic suffers from CMOS (i.e. transistor mismatch [23]) and memristor variations (cycle-to-cycle and device-to-device [22]). Such variations may cause circuit failures if not addressed appropriately. Extensive research on CMOS process variations in sense amplifiers has been proposed [23,24]; hence we will elaborate only on memristor variations and how scouting logic can be made more resilient to such variations.

Fig. 10 shows a methodology that can be used to realize a more robust scouting logic design against variations. Firstly, the design is simulated without variations and the range for R_L and R_H are determined. Fig. 11(a) shows the relationship between the equivalent resistance of two input memristors (R_{eq}) and the normalized output voltage ($\frac{V_{\text{out}}}{V_{\text{dd}}}$) of the sense amplifier of Fig. 5 configured for XOR operation; we assume here that the output threshold for logic 1 is 0.6V$_{\text{dd}}$ and 0.4V$_{\text{dd}}$ for logic 0 (see red lines in Fig. 11(a)). Hence, the resistance ranges for R_L and R_H can be determined by sweeping the resistance values of memristors using SPICE simulations.

Since process variations may cause the resistance values to deviate from their intended values [22,25], the equivalent resistance might fall outside the working range of the memristor leading to operational failure. Variations are typically modeled as a normal distribution as shown in Fig. 11(b) [22], where the normalized standard deviation versus the mean (or intended) resistance values is given. The figure shows that R_H suffers from large variance as compared to R_L, and that the standard deviation reduces when smaller values for R_L and R_H are used. It is possible to use the standard deviation to express the failure rate. To realize a failure rate of e.g., 10^{-3}, the value of R_L or R_H should fall in the range of n_σ=3σ of the normal distribution [24] (i.e. μ+3σ and $\mu-3\sigma$). For example, a robust XOR gate design considering the variability should satisfy the following equations:

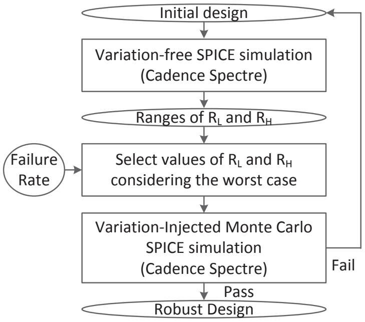


Fig. 10: Variation Resilient Design Methodology

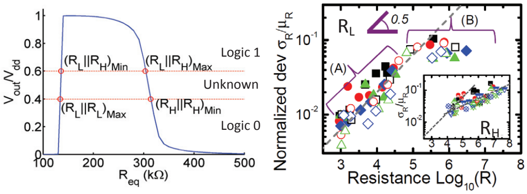
$$\frac{1}{2}(R_L + n_\sigma \sigma_{R_L}) < (R_L||R_L)_{\text{Max}} \tag{1}$$

$$\left[(R_L - n_\sigma \sigma_{R_L})\right]||\left[(R_H - n_\sigma \sigma_{R_H})\right] > (R_L||R_H)_{\text{Min}} \tag{2}$$

$$\left[(R_L + n_\sigma \sigma_{R_L})\right]||\left[(R_H + n_\sigma \sigma_{R_H})\right] < (R_L||R_H)_{\text{Max}} \tag{3}$$

$$\frac{1}{2}(R_H - n_\sigma \sigma_{R_H}) > (R_H||R_H)_{\text{Min}} \tag{4}$$

where σ_{R_L} and σ_{R_H} represent the standard deviation of R_L and R_H, respectively. Eq. 1 applies when both XOR inputs are 1, and hence output is 0. As the maximum value of the low resistance of each input memristor is $R_L + n_\sigma \sigma_{R_L}$, the maximum equivalent resistance $\frac{1}{2}(R_L + n_\sigma \sigma_{R_L})$ should be



(a) Relation between Resistance and Output Voltage

(b) Normalized Deviation of Resistance [15]

Fig. 11: Range and Normalized Deviation of Resistance

less than $(R_L||R_L)_{\text{Max}}$. Eq. 2 and 3 are the constraints for inputs 01/10 while Eq. 2 for inputs 00.

Once R_L and R_H are defined, the design is subsequently simulated using Monte Carlo simulations in Cadence Spectre [20]. The total number of failed simulations are monitored. If failure rate is lower than the requirement, then the design is considered robust; otherwise, the whole process is restarted by selecting new reference currents (i.e. redesign). Designs can be modified in two ways; either to modify mean resistance values, or tune the reference current/voltage value of the sense amplifier (i.e., $\frac{W}{L}$ of transistors in CSA or M_i of VSA). Changing mean resistances values can be accomplished by the writing circuity; e.g., by tuning the voltage pulse duration or amplitude to map logic states to resistance states as needed.

To verify the robust design methodology of Fig. 10, we apply the proposed approach to CSA-based scouting logic approach as a case study. Here, a 3σ design is used which satisfies a failure rate of 10^{-3}. The initial design (input of Fig. 10) is compared with the robust design (output of Fig. 10) in terms of failure rate. To verify the proposed approach, a 1000-iteration Monte Carlo simulation is conducted using Cadence Spectre simulator and the output voltage for each simulation run is monitored. The output voltage for logic 1 requires a voltage of $0.6\times V_{\text{dd}}$ or higher while for a logic 0 should be below $0.4\times V_{\text{dd}}$. Therefore, if a simulation run results in an output that does not fall in either range will be considered as an unknown and hence unreliable operation. If the logic state of the simulated output is different from the expected one, the design fails; otherwise, it passes. Table II summarizes technology and Monte Carlo simulation parameters used in the experiments. Note that the σ of the normal distribution are extracted from [22] depending on the selected resistance value.

The bottom of Table II shows failure rates as given by the Monte Carlo simulations. For the OR gate, both the initial and the robust design versions pass. However, for AND and XOR gates, the initial version fails when logic states are 01/10 while the robust version still passes. Therefore, the methodology proposed in Fig. 10 can enhance the robustness of the design to deal with resistance variation in scouting logic based design.

VI. Conclusion

This paper proposed scouting logic design for resistive computing. Such design does not reduce device endurance. In addition, it outperforms the existing memristor-based logics in terms of delay and power consumption, while using similar or less area.

TABLE II: Parameters and Results of Monte Carlo Simulations

Technology Parameters			
MOSFET	$p_{1,2}$	$p_{3,4,5}$	$n_{0,1,2,3}$
$\frac{W}{L}$	4	8	4

Version	Initial	Robust
R_H/R_L	50	
$R_L(\text{k}\Omega)$	23	30
$R_H(\text{k}\Omega)$	1150	1500

Monte Carlo Simulation Parameters		
Iteration	1000	
Version	Initial	Robust
σ_{R_L}/μ_{R_L} [22]	0.0261	0.0281
σ_{R_H}/μ_{R_H} [22]	0.0406	0.0418
V_{H}	$0.6\times V_{\text{dd}}$	
V_{L}	$0.4\times V_{\text{dd}}$	

Failure Rate						
Version	Initial			Robust		
Input / Gate	00	01/10	11	00	01/10	11
AND	0	19.10%	0	0	0	0
OR	0	0	0	0	0	0
XOR	0	45%	0	0	0	0

References

[1] ITRS ERD report. [Online]. Available: http://www.itrs.net

[2] B. Hoefflinger, *Chips 2020: a guide to the future of nanoelectronics*, 2012.

[3] J. J. Yang *et al.*, "Memristive devices for computing," *Nat. Nano*, 2013.

[4] S. Hamdioui *et al.*, "Memristor for computing: Myth or reality?" in *DATE*. IEEE, 2017.

[5] S. Hamdioui *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *DATE*. IEEE, 2015.

[6] H. A. Du Nguyen *et al.*, "Computation-in-memory based parallel adder," in *NANOARCH*. IEEE, 2015.

[7] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*. IEEE, 2016.

[8] G. S. Rose *et al.*, "Leveraging memristive systems in the construction of digital logic circuits," *Proceedings of the IEEE*, 2012.

[9] L. Gao *et al.*, "Programmable cmos/memristor threshold logic," *TNANO*, 2013.

[10] E. Linn *et al.*, "Beyond von neumannlogic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, 2012.

[11] J. Borghetti *et al.*, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, pp. 873–876, 2010.

[12] S. Kvatinsky *et al.*, "Mrl: memristor ratioed logic," in *CNNA*. IEEE, 2012.

[13] L. Xie *et al.*, "Boolean logic gate exploration for memristor crossbar," in *DTIS*. IEEE, 2016.

[14] V. Agarwal *et al.*, "Scalable graph exploration on multicore processors," in *SC*. ACM, 2010.

[15] J. Chou *et al.*, "Parallel index and query for large scale data analysis," in *SC*. ACM, 2011.

[16] L. Samuel, "Cmos current comparator with regenerative property," *IJECSE*, 2013.

[17] A. Siemon *et al.*, "Simulation of tao x-based complementary resistive switches by a physics-based memristive model," in *ISCAS*. IEEE, 2014.

[18] F. Miao *et al.*, "Anatomy of a nanoscale conduction channel reveals the mechanism of a high-performance memristor," *Adv. Mat.*, 2011.

[19] Predictive transistor model. [Online]. Available: http://ptm.asu.edu/

[20] Cadence ic design kit. [Online]. Available: https://www.cadence.com/

[21] M.-F. Chang *et al.*, "An offset-tolerant fast current-sampling-based sense amplifier for small-cell-current nonvolatile memory," *JSSC*, 2013.

[22] A. Fantini *et al.*, "Intrinsic switching variability in hfo 2 rram," in *IMW*. IEEE, 2013.

[23] W. Dehaene *et al.*, "Variability-aware design of low power sram memories," 2009.

[24] I. Agbo *et al.*, "Quantification of sense amplifier offset voltage degradation due to zero-and run-time variability," in *ISVLSI*. IEEE, 2016.

[25] M. Hu *et al.*, "Geometry variations analysis of tio 2 thin-film and spintronic memristors," in *ASP-DAC*. IEEE, 2011.

Enhanced Scouting Logic: A Robust Memristive Logic Design Scheme

Jintao Yu Hoang Anh Du Nguyen Muath Abu Lebdeh Mottaqiallah Taouil Said Hamdioui
Laboratory of Computer Engineering, Delft University of Technology
Delft, the Netherlands
{J.Yu-1, H.A.DuNguyen, M.F.M.AbuLebdeh, M.Taouil, S.Hamdioui}@tudelft.nl

Abstract—**Memristive devices have the potential to reduce the memory access bottleneck in conventional computer architectures. However, memristive devices also suffer from low endurance and large resistance variation. To address these problems, we present a robust logic scheme named Enhanced Scouting Logic (ESL). It produces logic operation results within the peripheral circuit of the memory array. During the execution of logic operations, the resistance states of memristive devices do not change and hence do not affect the memristor lifetime. ESL senses the resistance of input memristive devices via two different paths when different operations such as AND and OR are performed. These different paths guarantee the operation correctness even under large resistance variations. We verified ESL using SPICE simulations and Monte Carlo analysis. Our simulation results show that ESL is more robust as compared with state-of-the-art logic schemes.**

I. Introduction

Today's big-data applications demand high performance and high energy efficiency, challenging both CMOS technology and conventional computer architectures [1]. In CMOS technology, the energy problem worsens as the leakage power increases with downscaling [2]. In conventional CPU-centric architectures, the time and energy spent in accessing the main memory are significant; it can even exceed those of computations [3]. Therefore, new solutions are needed to alleviate the architecture and technology challenges. Emerging memristive devices have the potential to reduce static power due to their non-volatility. They also enable computation-in-memory as they can be used for both computing and storage [4]. Preliminary researches have exhibited the huge potential of memristive computing systems [5], [6]. Nevertheless, memristive devices also face multiple challenges. For instance, RRAM suffers from large resistance variation and low endurance; PCRAM requires long write time and high write energy; STT-MRAM has relatively poor process compatibility with mainstream silicon CMOS technology, which leads to a high manufacture cost [7], [8].

Most memristive logic schemes, such as Snider [9] and IMP [10], generate the results in the form of resistance states in the array. Therefore, they program the memristive devices frequently, hence reducing the device endurance. To avoid this problem, other schemes, such as Pinatubo [5] and Scouting Logic [11], generate the results in the periphery without switching the resistance states in the array; e.g., the results are produced as voltages at the output of sense amplifiers. These logic schemes do not affect the endurance, and hence, they maintain a longer lifetime. However, these schemes may not guarantee correctness when the input memristive devices exhibit large resistance variation. As far as we know, state-of-the-art logic schemes are not able to address both endurance and variation problems.

In this paper, we propose enhanced scouting logic (ESL), a logic scheme that improves Scouting Logic [11] by addressing the two major challenges when performing in-memory logic operations using RRAM; namely, the low endurance and the high resistance variation. Similarly as in Pinatubo and Scouting Logic, the proposed solution addresses the low endurance problem by producing the results in the memory periphery without changing the resistance states of the involved memory cells. It also addresses the high resistance variation problem by using 2T1R cells allowing two separate reading paths for AND and OR operations. The main contributions of this paper are:

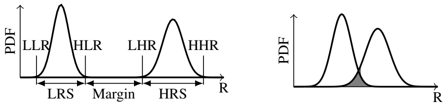
- It reviews existing logic schemes that produce the outputs of logic operations within the memory periphery. Our results show that these schemes were not properly evaluated against realistic resistance variations of the RRAM devices.
- It proposes a memristive logic scheme that is resilient to resistance variation without affecting the memory endurance.
- It validates the proposed scheme and compares the scheme with the state-of-the-art using SPICE simulation and Monte Carlo analysis.

This paper is organized as follows. Section II and Section III review the resistance variation of RRAM devices and the logic styles that perform in memory peripheral, respectively. Subsequently, Section IV presents the working principle and implementation details of ESL. Next, Section V verifies ESL using SPICE simulation. In addition, Section V applies Monte Carlo simulation on ESL and other logic schemes. Finally, Section VI concludes the paper.

II. Resistance Variation in RRAM Devices

A typical RRAM device has two stable resistance states; i.e., a low resistance state (LRS) and a high resistance sate (HRS). The SET voltage and the RESET voltage are applied across the

This work was supported by the European Unions Horizon 2020 Research and Innovation Program through the project MNEMOSENE (Grant 780215).



(a) Separated ranges (b) Overlapped ranges

Fig. 1. Typical resistance distribution of LRS and HRS ranges.

RRAM device to switch its resistance state to LRS and to HRS, respectively. The resistance (either LRS or HRS) of RRAM devices may vary as a result of the cycle-to-cycle (C2C) and device-to-device (D2D) variations [12]. The C2C and D2D variations have similar distributions [12] and can affect the correctness of logic operations. Due to their similarity, we do not distinguish between the C2C and D2D variations.

Fantini *et al.* observed that the distributions of LRS and HRS approximately follow a lognormal distribution [13], as shown in Fig. 1. The x axis represents the resistance in log scale while the y axis represents probability density function (PDF). The lower and upper bounds of LRS are named low low resistance (LLR) and high low resistance (HLR), respectively. Similarly, the lower and upper bounds of HRS are referred to as low high resistance (LHR) and high high resistance (HHR), respectively. We are particularly interested in the devices that have separate LRS and HRS ranges, i.e., LHR is larger than HLR, as shown in Fig. 1a. We refer to the ratio between LHR and HLR as *margin*. If LHR is smaller than HLR (i.e., negative margin) as shown in Fig. 1b, the proposed logic schemes will not work properly.

The ideal RRAM device should have small LRS and HRS ranges (i.e., narrow distributions of LRS and HRS) and a large margin; however, this is usually not the case. The LRS and HRS ranges can be estimated based on their distribution. For the LRS of an HfO_2 RRAM, $\sigma/\log_{10}\mu$ is between 0.2% and 10% [13]. As a result, the ratio between HLR and LLR (HLR/LLR) varies from 1.2 to 1000 considering a 3σ population. For example, when $\sigma/\log_{10}\mu = 2\%$ and $\mu = 10\,\text{k}\Omega$, the range of LRS is between $6.9\,\text{k}\Omega$ and $17\,\text{k}\Omega$ and HLR/LLR$= 2.5$.

Grossi *et al.* assume that LRS variations can be represented by a natural distribution. They summarize the RRAM's resistance variation based on seven publications [14]. In most cases, σ/μ of LRS resistance is larger than 10%. In some cases, this value is nearly 100%. Considering a 3σ population, we observe that HLR/LLR> 1.9. Note that the HRS range is even larger, but less important as it affects the operations less. This will be more clear in the next sections.

Table I shows the LRS and HRS ranges of recently proposed RRAM devices and the materials they are made of. The last column shows the margin between the two resistance states. The table shows that HLR/LLR$\in (1.5, 5)$ and $margin \in (4, 80)$. This indicates that LRS and HRS both have large variations, but also that a wide gap exists between them.

TABLE I
RESISTANCE RANGES OF RECENT RRAM DEVICES

Ref.	Material	LRS range	$\frac{\text{HLR}}{\text{LLR}}$	HRS range	$\frac{\text{HHR}}{\text{LHR}}$	Margin
[15]	Pt-AlO_y/HfO_x-TiN	200-1k	5	80k-2M	25	$80x$
[16]	Cu-MgO-Ru	400-600	1.5	4k-50k	12.5	$8x$
[17]	Ti-TiN-HfO_x	300-600	2	3k-20k	6.6	$10x$
[12]	HfO_2-Hf	30k-80k	2.6	300k-1M	3.3	$4x$
[18]	Au-SiO_x-Mo	200-350	1.8	2k-10k	5	$6x$
[19]	Ti-SiO_x-C	200-350	1.8	2k-10k	5	$6x$
[20]	Ti/Ta-TMO_x	10k-50k	5	500k-500M	1000	$10x$

III. Bulk Bitwise Boolean Logic Schemes

Seshadri *et al.* was the first to propose a mechanism to perform *bulk* bitwise operations using specially designed peripheral circuits of memories [21]. These bitwise operations are performed on large vectors [31]. This idea inspired many researchers, including the usage of novel devices such as memristive devices. They are summarized in Table II.

The table contains for each reference the scheme name, used memory technology, the operations that are supported, whether variations are considered, and the worst case latency of the supported operations. Some articles target general non-volatile memories or memristive devices [5], [11], [25] while some other schemes work specifically for a particular memory such as STT-MRAM [26]–[29]. ESL, added to the last row of the table, can be applied to any type of memristive devices. However, it is most useful for devices with large resistance variations such as RRAM.

The fourth column of the table shows the different Boolean operations that are supported in the articles. Some schemes [26], [28], [30] support NAND/NOR operations by selecting the sense amplifier's $\overline{OUT}$ signal during AND/OR. This is easy to implement in all schemes and not the focus of this paper. Therefore, such operations are omitted in the table for simplicity.

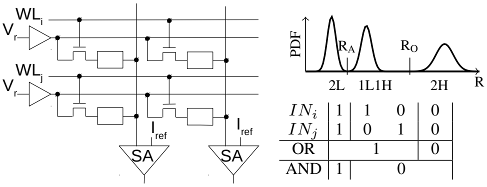
We are particularly interested in the fifth column, i.e., whether resistance variations have been considered. This is however not relevant for DRAM and SRAM, as their variation is very small. In contrast, it is essential for memristive devices as they may introduce errors. This variation is mentioned as a percentage in case reported; the percentage represents (HLR-LLR)/μ_{LRS}. Note that the articles not always describe the variation in this form. For example, Jain *et al.* assume that the MTJ oxide thickness has 2% variation. In such cases, we converted the given variation type into a resistance variation. The only exception is Chen *et al.*, as they verified their schemes using real RRAM devices.

The last two columns show the worst case latency in number of clock cycles required for a logic operation and whether multiple operands are supported simultaneously, respectively. Some articles claim that they support multiple operands (executed by activating multiple rows in the memory), at least for the OR operation [5], [11], [25]. However, the ability of the sense amplifier to be able to distinguish between the outcomes is extremely difficult or even not possible.

All the bulk bitwise operations based on memristive devices that are listed in Table II are implemented with similar hardware structures. Fig. 2a represents such a hardware structure.

TABLE II
BULK BITWISE BOOLEAN LOGIC OPERATIONS

Reference	Scheme name	Technology	Supported operations	Variation considered?	Longest delay	Multirow
[21]	Seshadri *et al.*	DRAM	AND, OR	No	4	No
[22]	Ambit	DRAM	AND, OR, NOT	No	4	No
[23]	DRISA	DRAM	NOR	No	2	No
[24]	Jeloka *et al.*	SRAM	AND, NOR	No	1	Yes
[5]	Pinatubo	NVM	AND, OR, XOR, NOT	No	2	Yes
[11]	Scouting logic	Memristor	AND, OR, XOR	Yes, $< 10\%$	1	Yes
[25]	MPIM	Memristor	AND, OR, XOR	Yes, 10%	1	Yes
[26]	IMP/NMP	STT-MRAM	AND, OR, NOT	No	1	No
[27]	HielM	STT-MRAM	AND, OR, XOR	Yes, $< 5\%$	4	No
[28]	STT-CiM	STT-MRAM	AND, OR, NOT	Yes, $< 10\%$,	1	No
[29]	IMCS2	STT-MRAM	AND, OR, XOR, NOT	Yes, $< 5\%$	1	No
[30]	Chen *et al.*	RRAM	AND, OR, XOR	Yes, measurement	1	No
This work	ESL	RRAM	AND, OR, NOT	Yes, $> 100\%$	1	No



IN_i	1	1	0	0
IN_j	1	0	1	0
OR	1			0
AND	1	0		

(a) Hardware structure

(b) Parallel resistance

Fig. 2. Working principle of memristive bulk bitwise operations.

The bulk bitwise operations are special read operations. Rows are activated by drivers (represented by triangles) and operate at read voltage V_r. The memory cells consist of a 1T1R structure, i.e. a transistor in parallel with a memristive device (represented by a rectangle) such as RRAM or STT-MRAM. During bulk bitwise operations, two or more rows are selected at the same time. The total current through the two cells in each column is compared with a reference current I_{ref} by the sense amplifier. The output of the sense amplifier is the desired result of the required Boolean function. There are also other forms of comparison such as using a voltage reference. However, the essence is to compare the equivalent resistance of two parallel cells with a reference resistance.

The resistance distribution of two cells (IN$_i$ and IN$_j$) that are connected in parallel is shown in the top of Fig. 2b. Here, we assume that a logic one and zero are represented by a low and high resistance, respectively. There are three possible combinations: 1) both cells in HRS (2H), 2) one cell in LRS and one in HRS (1L1H), and 3) both cells in LRS (2L). To conduct an OR operation, we need to set a reference R_O to distinguish between the cases 2H and 1L1H, as indicated by the truth table in Fig. 2b. If the device has a large margin between LRS and HRS like the ones listed in Table I, it would be easy to define R_O. However, it is difficult or impossible to set a reference R_A to distinguish 1L1H from 2L in case an AND operation is implemented. When HLR/LLR>2, which is the case of many devices in Table I, the 2L and 1L1H regions overlap. Thus, a correct AND operation cannot be guaranteed regardless of the value of R_A.

Chen *et al.* proposed a self-write termination (SWT) scheme for RRAM devices to avoid the above mentioned problem [30]. The scheme adds a loop-back from the cell to the driver during write operations. When the programmed memristive device reaches the desired resistance, the writing process is terminated. SWT's main target is to reduce the write energy and the standard deviation of the resistance distribution. However, SWT schemes also have challenges such as premature ending of write operations that may lead to unstable resistance states [32]. In addition, it is also difficult to achieve HLR/LLR<2 using SWT schemes for some devices [32].

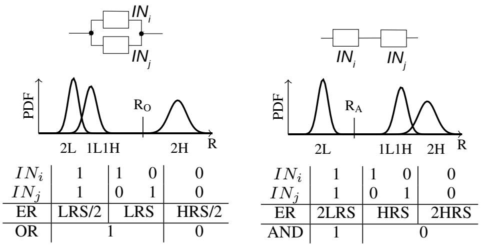
IV. ENHANCED SCOUTING LOGIC

In this section, we first introduce the general concept behind ESL. Thereafter, we present the implementation details.

A. General Concept

To prevent the possibility of overlapping regions of different equivalent resistance values, ESL changes the connection of memristive devices based on the operation type. Despite the large variations, a reference resistance can be still safely selected. As shown in Fig. 2b, there are no issues in selecting a reference resistance R_O even if the regions 2L and 1L1H are overlapping. Hence, this part of the circuit does not have to be changed. This is shown in Fig. 3a. However, for AND operations, ESL changes the circuit structure and connects the input cells in series. The overall resistance's distribution will be similar to the one shown in Fig. 3b. Notably, the equivalent resistance (ER) of 1L1H would be approximately HRS instead of LRS. Therefore, there is a relatively large margin between 2L and 1L1H, and the reference R_A can be easily selected.

Considering the fact that most devices have a large margin, only one reference is needed in ESL. From Fig. 3, we observe that the following relations must satisfy: LRS$<R_O<$HRS$/2$ (as ER of 1L1H$\approx$LRS and 2H$\approx$HRS$/2$ for the OR operation) and 2LRS$<R_A<$HRS (as ER of 2L=2LRS and 1L1H$\approx$HRS for the AND operation). From these relations we can easily observe that for given $R_O = R_A = R_{ref}$ the equation

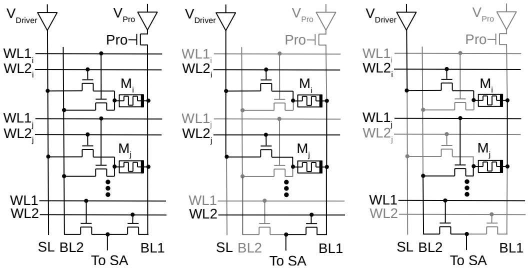


(a) OR operation (b) AND operation

Fig. 3. Resistance distribution of Enhanced Scouting Logic.

TABLE III
CONTROL SIGNALS FOR ESL IMPLEMENTATION

	Write	Read	OR	AND
V_{Driver}	0/V_{RESET}	V_{Read}	V_{Read}	V_{Read}
V_{Pro}	V_{SET}/0	–	–	–
Pro	V_{dd}	0	0	0
$WL1_i$	0	0	0	0
$WL2_i$	V_{dd}	V_{dd}	V_{dd}	V_{dd}
$WL1_j$	0	0	0	V_{dd}
$WL2_j$	0	0	V_{dd}	0
WL1	0	0	0	V_{dd}
WL2	0	V_{dd}	V_{dd}	0



(a) Circuit (b) OR operation (c) AND operation

Fig. 4. ESL circuit.

2LRS<R_{ref}<HRS/2 must hold. Hence, a single reference can be used when the resistance margin is larger than $4x$. Note that all the devices listed in Table I meet this requirement. This feature makes the reference circuit much simpler as compared to those needed in previous bulk bitwise operation schemes [5], [11], [25].

B. Implementation

The circuit to implement ESL is shown in Fig. 4a. For simplicity reasons, Fig. 4a illustrates the concept using only two cells (M_i and M_j) located in a single column. Other cells in this column and other columns have similar structures. Besides the standard bitline BL1 in a 1T1R array, a second bitline BL2 is added to each column to be able to connect two devices in series which is needed for the AND operation. The two bitlines share one sense amplifier using two transistors (controlled by voltages WL1 and WL2, respectively). One more transistor is added to each cell to connect the device to BL2.

The values of the control signals during each operation are shown in Table III. The operations are the normal write/read and the bitwise OR/AND operations. The table contains for each of the control lines in Fig. 4 the voltage values per operation type. When we write M_i (i.e. perform a SET or RESET operation), Pro and $WL2_i$ are enabled. The voltage level of the Driver and V_{Pro} depend on the value to be programmed. For instance, the voltage applied to M_i is V_{SET} if we want to program a logical one. When we read cell M_i, $WL2_i$ and WL2 are enabled and Driver's voltage changes to V_{Read}. In our implementation, V_{Read}=V_{dd}. However, it can also be a different value, e.g., to prevent a drift of the memristor state. During OR/AND operations, the Driver's voltage is also V_{Read}. Pro is disabled to disconnect the programming driver to the circuit. In an OR operation between M_i and M_j, $WL2_i$ and $WL2_j$ are on. As a result, $Driver_i$ and $Driver_j$ drive the two cells in parallel. Fig. 4b illustrates these signals by coloring disabled transistors and unused wires gray. In an AND operation, however, only one driver (e.g., $Driver_i$) is connected to the cells as shown in Fig. 4c. $WL1_j$ and WL1 are enabled to connect the two cells to the SA in series.

We use the state-of-the-art CSB-SA [33] sense amplifier in our design. It operates in three phases. In the first phase, it captures the inputs, i.e., the currents through the memristor cell and the reference resistor are sampled and stored in two capacitors, respectively. In the second phase, the stored charges on the capacitor are amplified. Finally, the output latch is enabled during the third phase and a digital output is generated. Note that other sense amplifiers could also be used. The selected sense amplifier mainly depends on the trade-off between performance and area.

V. SIMULATION RESULTS

In this section, we first present the simulation setup. Subsequently, we validate ESL using SPICE simulation. Finally, we conduct Monte Carlo simulation with ESL and other logic schemes and evaluate their results.

A. Simulation Setup

To verify the proposed scheme, we use SPICE simulations. The circuit in Fig. 4 is implemented and connected to the CBS-SA sense amplifier. The simulation parameters are summarized in Table IV. The supply voltage V_{dd} is set to $0.9\,\mathrm{V}$. The resistive device of the 2T1R cell is implemented using the ASU model [34] configured with the RRAM parameters presented in [20]. The RRAM device is implemented with $28\,\mathrm{nm}$ technology. To match the transistor sizes, we therefore adopt the $32\,\mathrm{nm}$ PTM model [35], which is the closest technology node for CMOS transistors in PTM. The sense amplifier uses a reference resistor that is set to $160\,\mathrm{k\Omega}$, which is the geometric average of HLR and LHR of the RRAM [20]. The three phases of the sense amplifier are set to be $1\,\mathrm{ns}$.

TABLE IV
SPICE SIMULATION PARAMETERS

Parameter		Description	Value
V_{dd}		CMOS power supply	0.9 V
LLR		Low low resistance	10 kΩ
HLR		High low resistance	50 kΩ
LHR		Low high resistance	500 kΩ
HHR		High high resistance	500 MΩ
F		Technology node	28 nm
R_{ref}		Reference resistance	160 kΩ
t_s		Sense amplifier phase	1 ns
Monte Carlo simulation			
N_σ		Number of sigma	3
N		Number of iterations	10,000
R_L	μ	Mean resistance	30 kΩ
	σ	Standard deviation	0.5
R_H	μ	Mean resistance	16.6 MΩ
	σ	Standard deviation	1.68

To verify the robustness of ESL against resistance variations, all the corner cases have been simulated. These corner cases are derived from the extreme resistance values for each of the two inputs. The four extreme resistance values are LLR, HLR, LHR, and HHR. Therefore, there are 16 combinations for two inputs. All of these combinations are simulated for both AND and OR operations, respectively.

To compare our scheme with the with the state-of-the-art schemes, we also conducted Monte Carlo simulations of the AND gates of Scouting Logic [11], Pinatubo [5], and ESL, respectively. We implemented these circuits according to their papers. Subsequently, we configured the parameters of two lognormal distributions as listed in the bottom part of Table IV. Finally, we performed 10,000 Monte Carlo simulations for each design scheme.

B. Validation of ESL

Based on the simulation of the 16 corner cases, we conclude that the proposed ESL implementation is robust against resistance variations for both the AND and OR operations. Fig. 5 shows the waveform of all the 16 test cases for the AND operation. The x axis represents time, and the y axis indicates the voltage. The bitline voltages in the 16 test cases are divided into three groups according to the logic values of the input cells. The 2H group is colored red, 1L1H cyan, and 2L orange. The voltage of the reference bitline is colored black. The three phases can be easily identified in the figure. The resistance of the input cells is sampled in the first phase. The higher the voltage in the graph, the higher the resistance. We refer the readers to [33] for more details about the working principle of the sense amplifier. The reference is set between 1L1H and 2L as shown in Fig. 3b. Note that the 2H and 1L1H groups overlap. For example, HLR+HHR> 2LHR; while HLR+HHR belongs to 1L1H and 2LHR to 2H, the resistance of 1L1H can be still higher than 2H. After the two-step amplification in phases 2 and 3, the 2L cases generate a logical one while the remaining cases a logical zero. These results are consistent with the AND's truth table.

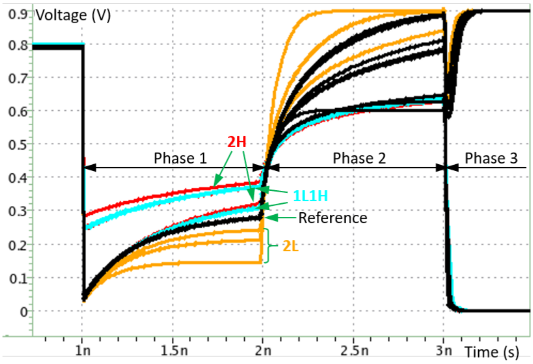


Fig. 5. SPICE simulation results of AND operation.

TABLE V
THE NUMBERS OF FAILED CASES IN 10,000 MONTE CARLO ITERATIONS

	HH	HL	LH	LL	Total
Scouting Logic	0	75	97	202	374
Pinatubo	0	142	176	332	650
ESL (This work)	0	0	0	0	0

The waveform of the OR operation is similar to Fig. 5 and hence not included in the paper for simplicity. Also the 16 cases verified the correctness of ESL.

C. Comparison with State of the Art

Table V summarizes for Scouting Logic, Pinatubo, and ESL the number of failures observed during Monte Carlo simulations. A simulation is considered to fail when a wrong logic value is observed (an output voltage lower than 40% of V_{dd} is considered a logic zero and higher than 60% of V_{dd} a logic one) or when the sense amplifier output ranges between 40% and 60% of the V_{dd} value. Each row in the table represents a logic scheme. The columns are the four resistance combination of the two input memristors, i.e. HH, HL, LH, and LL, and the numbers represent the total failed cases. The table clearly shows that ESL is robust against variations, while the other schemes can practically not be used. Note that we configured the reference resistance for Pinatubo and Scouting Logic with different values and repeated the simulations. Only the simulation groups that had the smallest total number of failed cases are reported in Table V. It is impossible to find a perfect value that avoids failures for these designs, as the input resistance ranges of 1L1H and 2L overlap.

D. Discussion

The main advantage of ESL is that it is able to deal with large resistance variations. Many previous works overlooked this fact and did not consider this during design. ESL has besides this benefit also other benefits. It requires only one reference resistor which makes it simpler than previous works. Finally, like other bulk bitwise logic schemes, ESL does not write to memristors during process; hence, it does not affect the endurance during logic operations.

To enhance the robustness, we designed a 2T1R memory cell structure, which is different from the typical cases such as 1R, 1T1R, or 1S1R. Compared with these structures, 2T1R requires more control signals and has a larger cell area.

VI. Conclusion

Today's RRAM devices suffer from large resistance variations. Existing Boolean logic schemes that based on resistance sensing cannot guarantee the correctness of these operations under such variations. To enhance the robustness, we have proposed a new scheme that senses the resistance via two paths for AND and OR operations, respectively. SPICE simulation and Monte Carlo analysis have proved the robustness of our scheme. This scheme brings the memristive logic circuits a step closer to reality as the robustness is a primary concern.

References

[1] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs *et al.*, "Memristor for computing: Myth or reality?" in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 722–731.

[2] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Amsterdam, Netherlands: Elsevier, Nov. 2017.

[3] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb 2014, pp. 10–14.

[4] S. Hamdioui, L. Xie, H. A. D. Nguyen *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1718–1725.

[5] S. Li, C. Xu, Q. Zou *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*. New York, NY, USA: ACM, 2016, pp. 173:1–173:6.

[6] J. Yu, H. A. Du Nguyen, L. Xie *et al.*, "Memristive devices for computation-in-memory," in *2018 Design, Automation Test in Europe Conference Exhibition*. IEEE, March 2018, pp. 1646–1651.

[7] S. Yu and P. Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, Spring 2016.

[8] A. Chen, "A review of emerging non-volatile memory (NVM) technologies and applications," *Solid-State Electronics*, vol. 125, no. Supplement C, pp. 25 – 38, 2016.

[9] G. Snider, "Computing with hysteretic resistor crossbars," *Applied Physics A*, vol. 80, no. 6, pp. 1165–1172, 2005.

[10] J. Borghetti, G. Snider, P. Kuekes *et al.*, "'Memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[11] L. Xie, H. A. D. Nguyen, J. Yu *et al.*, "Scouting logic: A novel memristor-based logic design for resistive computing," in *2017 IEEE Computer Society Annual Symposium on VLSI*, July 2017, pp. 176–181.

[12] L. Zhang, S. Cosemans, D. J. Wouters *et al.*, "Cell variability impact on the one-selector one-resistor cross-point array performance," *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3490–3497, Nov 2015.

[13] A. Fantini, L. Goux, R. Degraeve *et al.*, "Intrinsic switching variability in HfO2 RRAM," in *5th IEEE International Memory Workshop*, May 2013, pp. 30–33.

[14] A. Grossi, E. Nowak, C. Zambelli *et al.*, "Fundamental variability limits of filament-based RRAM," in *2016 IEEE International Electron Devices Meeting*, Dec 2016, pp. 4.7.1–4.7.4.

[15] R. Han, P. Huang, Y. Zhao *et al.*, "Demonstration of logic operations in high-performance RRAM crossbar array fabricated by atomic layer deposition technique," *Nanoscale Research Letters*, vol. 12, no. 1, p. 37, Jan 2017.

[16] X. Hong, P. A. Dananjaya, S. Krishnia *et al.*, "A novel geometry of ECM-based RRAM with improved variability," *Journal of Physics D: Applied Physics*, 2018.

[17] Y. Fang, Z. Yu, Z. Wang *et al.*, "Improvement of HfOx-Based RRAM device variation by inserting ALD TiN buffer layer," *IEEE Electron Device Letters*, vol. 39, no. 6, pp. 819–822, June 2018.

[18] A. Mehonic, M. Munde, W. Ng *et al.*, "Intrinsic resistance switching in amorphous silicon oxide for high performance siox reram devices," *Microelectronic Engineering*, vol. 178, pp. 98 – 103, 2017, iNFOS.

[19] A. Bricalli, E. Ambrosi, M. Laudato *et al.*, "Siox-based resistive switching memory (RRAM) for crossbar storage/select elements with high on/off ratio," in *2016 IEEE International Electron Devices Meeting*, Dec 2016, pp. 4.3.1–4.3.4.

[20] H. Lv, X. Xu, P. Yuan *et al.*, "Beol based rram with one extra-mask for low cost, highly reliable embedded application in 28 nm node and beyond," in *2017 IEEE International Electron Devices Meeting*, Dec 2017, pp. 2.4.1–2.4.4.

[21] V. Seshadri, K. Hsieh, A. Boroum *et al.*, "Fast bulk bitwise AND and OR in DRAM," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, July 2015.

[22] V. Seshadri, D. Lee, T. Mullins *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2017, pp. 273–287.

[23] S. Li, D. Niu, K. T. Malladi *et al.*, "Drisa: A DRAM-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2017, pp. 288–301.

[24] S. Jeloka, N. B. Akesh, D. Sylvester *et al.*, "A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, April 2016.

[25] M. Imani, Y. Kim, and T. Rosing, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *22nd Asia and South Pacific Design Automation Conference*, Jan 2017, pp. 757–763.

[26] W. Kang, H. Wang, Z. Wang *et al.*, "In-memory processing paradigm for bitwise logic operations in STT-MRAM," *IEEE Transactions on Magnetics*, vol. 53, no. 11, pp. 1–4, Nov 2017.

[27] F. Parveen, Z. He, S. Angizi *et al.*, "Hielm: Highly flexible in-memory computing using STT MRAM," in *23rd Asia and South Pacific Design Automation Conference*, Jan 2018, pp. 361–366.

[28] S. Jain, A. Ranjan, K. Roy *et al.*, "Computing in memory with spin-transfer torque magnetic RAM," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 470–483, March 2018.

[29] F. Parveen, S. Angizi, Z. He *et al.*, "Imcs2: Novel device-to-architecture co-design for low-power in-memory computing platform using coterminous spin switch," *IEEE Transactions on Magnetics*, pp. 1–14, 2018.

[30] W. H. Chen, W. J. Lin, L. Y. Lai *et al.*, "A 16mb dual-mode ReRAM macro with sub-14ns computing-in-memory and memory functions enabled by self-write termination scheme," in *2017 IEEE International Electron Devices Meeting*, Dec 2017, pp. 28.2.1–28.2.4.

[31] D. Knuth, *The Art of Computer Programming: Bitwise Tricks & Techniques*, 1st ed. Boston, MA: Addison-Wesley, Mar. 2009.

[32] Z. Wang, Y. Liu, A. Lee *et al.*, "A 65-nm reram-enabled nonvolatile processor with time-space domain adaption and self-write-termination achieving $> 4\times$ faster clock frequency and $> 6\times$ higher restore speed," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 10, pp. 2769–2785, Oct 2017.

[33] M. F. Chang, S. J. Shen, C. C. Liu *et al.*, "An offset-tolerant fast-random-read current-sampling-based sense amplifier for small-cell-current nonvolatile memory," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 3, pp. 864–877, March 2013.

[34] P. Y. Chen and S. Yu, "Compact modeling of rram devices and its applications in 1T1R and 1S1R array design," *IEEE Transactions on Electron Devices*, vol. 62, no. 12, pp. 4022–4028, Dec 2015.

[35] NIMO Group, ASU, "Predictive technology model." [Online]. Available: http://ptm.asu.edu/

B

Publications - Architecture Level

This chapter presents the publications on the architecture level. The following papers are included:

1. **J. Yu**, H. A. Du Nguyen, L. Xie, M. Taouil, S. Hamdioui, *Memristive Devices for computation-in-memory*, The 21st Design, Automation & Test in Europe Conference & Exhibition (DATE'18), March 2018, pp. 1646-1651.

2. **J. Yu**, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, S. Hamdioui, *Time-division Multiplexing Automata Processor*, The 22nd Design, Automation & Test in Europe Conference & Exhibition (DATE'19), Florence, Italy, March 2019, pp. 794-799.

Memristive Devices for Computation-In-Memory

Jintao Yu, Hoang Anh Du Nguyen, Lei Xie, Mottaqiallah Taouil, Said Hamdioui
Laboratory of Computer Engineering, Delft University of Technology, the Netherlands
Email: {J.Yu-1,H.A.DuNguyen,L.Xie,M.Taouil,S.Hamdioui}@tudelft.nl

***Abstract*—CMOS technology and its continuous scaling have made electronics and computers accessible and affordable for almost everyone on the globe; in addition, they have enabled the solutions of a wide range of societal problems and applications. Today, however, both the technology and the computer architectures are facing severe challenges/walls making them incapable of providing the demanded computing power with tight constraints. This motivates the need for the exploration of novel architectures based on new device technologies; not only to sustain the financial benefit of technology scaling, but also to develop solutions for extremely demanding emerging applications. This paper presents two computation-in-memory based accelerators making use of emerging memristive devices; they are Memristive Vector Processor and RRAM Automata Processor. The preliminary results of these two accelerators show significant improvement in terms of latency, energy and area as compared to today's architectures and design.**

I. Introduction

Today's and new emerging applications, such as data-intensive/big-data applications (e.g., DNA sequencing) and internet-of-things (IoT), are extremely demanding with respect to computing power, energy consumption, and storage. These applications will not only strongly shape our near future, but also impact the semiconductor and computer industry. However, their requirements are difficult to fulfill with today's CMOS based computer architectures, as they face sever challenges both at architectural and device level. Current computer architectures face three walls [1]: (1) the memory wall due to the growing gap between processor and memory speed and the the limited memory bandwidth; (2) the power wall as the practical power budget for cooling has been reached; (3) the instruction-level parallelism (ILP) wall due to the growing difficulties in extracting enough parallelism in software/code that can run on the mainstream parallel hardware today. The CMOS devices also face three walls [2]: (1) the leakage wall as the static power is becoming dominant at small technology nodes (due to volatile technology and low Vdd) and it may even be higher than the dynamic power, (2) the reliability wall as technology scaling leads to reduced device lifetime and higher failure rate; (3) the cost wall as the cost per device from a pure geometric scaling of technology point of view is plateauing. Both architecture and device walls have slowed down the performance gains of CMOS-based architectures. All these motivate the need to look for alternative architectures while considering emerging device technologies.

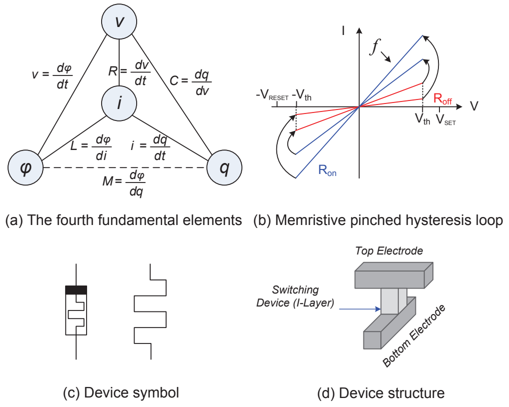
Many alternatives architectures are under investigations. Resistive computing [3–5] and neuromorphic computing architectures [6,7] using memristive devices, and quantum computing using quantum dots [8] are couple of examples. Resistive computing architectures based on memristive devices are attractive, as they enable in-memory computing (reducing the memory wall) [2,9]. In addition, the memristive devices have zero standby power [6] (helps reducing both the leakage and power wall), great scalability (reduces the cost wall), high density (reduces the cost wall), and they are CMOS compatible (reduces the cost wall).

This paper discusses two memristive device based accelerators to demonstrate how computation-in-memory architectures can realize significant improvements, due both to the architecture itself as well as to the used technology to implement them. First, a memristive based vector processor, referred to as Memristive Vector Processor (MVP), is presented; MVP can be used as an accelerator for conventional machines and shows approximately one order of magnitude improvement in performance and energy efficiency. Thereafter, a general model for hardware-based automata processing is introduced and implemented with memristive devices. This implementation is referred to as RRAM-AP; RRAM-AP's key kernel (i.e., the vector dot product operator) outperforms the state-of-the-art SRAM-based implementation by 40% less delay and 27% less energy, at even smaller chip area.

The reminder of this paper is organized as follows. Section II describes briefly the fundamentals of memristive devices. Section III and IV present MVP and RRAM-AP, respectively. Finally, Section V concludes the paper.

II. Basics of Memristive Devices

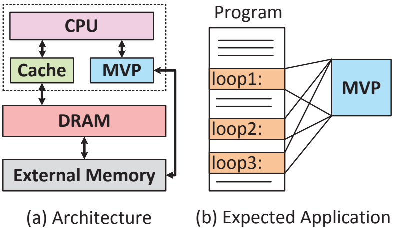
The memristive device, or *memristor* for short, is the fourth type of fundamental two-terminal electrical components, next to the resistor, capacitor, and inductor. It was initially predicted in 1971 by the circuit theorist Leon Chua [10]. He observed a missing element that can be described as a function of flux ϕ and charge q, as shown (with the dashed line) in Fig. 1a. In theory, a memristive device is a passive element that can be described by the current integral (charge q) through or voltage integral (flux ϕ) across its two terminals; The beauty of the memristive device is its ability to memorize the history (i.e., the internal state). The essential fingerprint of memristive devices is the pinched current-voltage hysteresis loop, as illustrated in Fig. 1b. When a memristive device is floating or when the voltage *v(t)* across it equals zero, the current *i(t)* is also zero. Therefore, based on its hysteresis curve, the memristor has at least two distinctive states: a high (R_H) and low (R_L) resistive state. A memristive device switches



(a) The fourth fundamental elements (b) Memristive pinched hysteresis loop

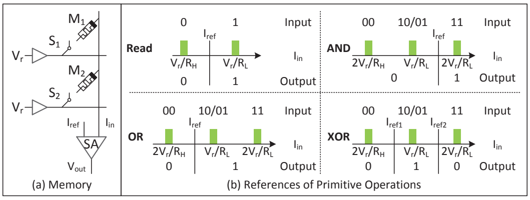
(c) Device symbol (d) Device structure

Fig. 1. Main characteristics of a memristive device.



(a) Architecture (b) Expected Application

Fig. 2. Memristive Vector Processor architecture.



(a) Memory (b) References of Primitive Operations

Fig. 3. Scouting logic [14].

from high (low) to low (high) state by applying a voltage V_{SET} (V_{RESET}) with an absolute value larger than its threshold voltage V_{th}. Another signature of the memristive devices is that the pinched hysteresis loop shrinks with a higher excitation frequency f as shown in Fig. 1b. Fig. 1c shows the two typical symbols used to denote memristive devices; the black square represents the positive terminal.

After a silent period for more than thirty years, a practical memristive device was fabricated and demonstrated by HP in 2008 [11]. HP built a metal-insulator-metal device using titanium oxide as an insulator and identified the memristive behaviour over its two-terminal node as described by Leon Chua; as shown in Fig. 1d. The device resistance is modulated by controlling positive charged oxygen vacancies in the insulator layer using different voltages. After the first memristive device was fabricated, several memristor devices based on different types of materials have been proposed such as spintronic, amorphous silicon, and ferroelectric memristors [6].

III. Memristive Devices for Vector Processing

Memristor-based Computation-In-Memory (CIM) concept was proposed to eliminate the communication between the CPU and memory by leveraging memristors for both storage and computation in the same physical crossbar [3,12,13]. Here, we use the CIM to realize an accelerator we refer to as Memristive Vector Processor (MVP). The rest of this section will describe the working principle of MVP, the targeted applications and some analytical evaluation results to show the potential of such an architecture.

A. Working principle

MVP is proposed to accelerate applications with a huge number of vector operations. It can be used as an accelerator for a conventional processor, as shown in Fig. 2a. Similarly as in conventional architectures, the processor fetches, decodes and executes a program using a memory hierarchy consisting of cache(s), DRAM, and external memory. The part of the program which is memory intensive will be offloaded to MVP. The distinct feature of MVP is its crossbar memory implementation using memristive devices, which enables not the storage of huge amount of data (due to its nano scale size), but also the processing of operations within the memory (i.e., no need for data movement).

The processing in MVP is performed based on scouting logic operations [5,14] ; they transform memory read operations into logical operations. Normally, when a memory cell is being read, a read voltage V_r is applied to the activated row as shown in Fig. 3a. Subsequently, a current will flow through the bit line to the input of the sense amplifier (SA) where it is compared to a reference current. Depending on the cell value (either low (R_L) or high (R_H) resistance), the output of the SA will produce either logic 1 or 0. Inspired by this read operation, scouting logic is able to implement OR, AND and XOR gates. Instead of reading a single memristor at a time, scouting logic activates two (or more) memory rows simultaneously. As a result, the input current to the sense amplifiers is determined by the equivalent input resistance of the activated rows. This resistance results in three possible values: R_H, $R_H//R_L \approx R_L$, or $R_L/2$; by changing the reference current of the SA, different gates can be realized (as shown in Fig. 3b). Therefore, using this scheme allows MVP to perform logical operations by just a small modification of the peripheral circuit of the crossbar mememory. It eliminates the necessity of temporary registers, loading latency and energy to move data from memory to registers. It also increases the parallelism of the architecture and does not impact the the endurance of the memristive devices.

B. Potential targeted applications

With its unique capability, MVP is able to accelerate data intensive applications. These applications consist of intensive memory accesses that consume an enormous amount of energy and degrade the overall performance due to data

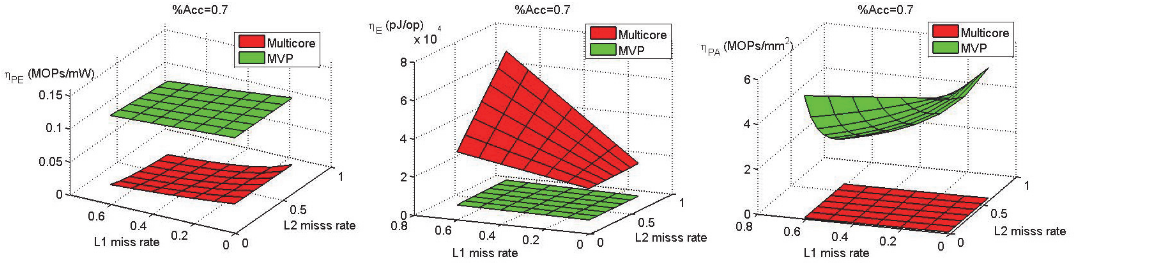


Fig. 4. Evaluation results for MVP and multicore architectures.

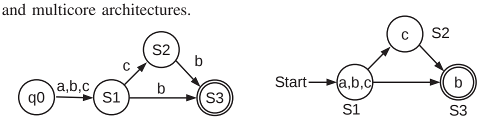
movements through the memory hierarchy; note that loading a word from the on-chip SRAM or off-chip DRAM costs much more energy (50x and 6400x, respectively) as compared with an ALU operation [15,16]. Therefore, eliminating data movements/ communication significantly improves the overall performance.

An example of a program that could benefit from MVP is illustrated in Fig. 2b. The program consists of multiple loops processing a dataset that is preloaded and mapped on MVP. Each time a loop is called, the processor sends a (macro)-instruction to MVP; the instruction is locally decoded and executed. The result is returned to the processor. This feature occurs in multiple applications such as database management [17], DNA sequencing [18–20], and graph processing [21].

C. Evaluation Results

To evaluate MVP architecture, its estimated performance is compared to a multicore architecture. The models and assumptions for the multicore architecture and MVP are similar to those in [3,9]; e.g., the multicore architecture consists of 4 cores (ALU only), two levels of caches (32 KB L1 and 256 KB L2) and 4 GB DRAM. The MVP architecture consists of one core (ALU only), two levels of caches (32 KB L1 and 256 KB L2), 2 GB DRAM, and a MVP with a 2 GB non-volatile crossbar memory with a modified read-out circuity (as explained in [14]) in order to enable computation-in-memory. Three metrics are used for the evaluation: (1) performance energy efficiency η_{PE} (defined by MOPs/mW), (2) energy efficiency η_E (defined by pJ/op), and (3) performance area efficiency η_{PA} (defined by MOPs/mm^2).

Fig. 4 shows the results of the evaluation metrics for both architectures for different L1 and L2 cache misses (up to 60%)and by assuming that 70% of the program instructions can be accelerated on MVP (%Acc=0,7); i.e., the 30% non-accelerated instructions is executed by the conventional processor and the 70% accelerated part by MVP; see Fig. 2. As MVP architecture contains a conventional part (i.e., CPU, caches, DRAM and external memory), only 10x improvement is obtained with respect to the performance-energy efficiency. MVP architecture also achieves one order of magnitude energy efficiency improvement in comparison with the multicore architecture, and has a higher performance area efficiency. Therefore, the MVP architecture has the potential of realizing significant improvements, despite the high switching latency and low endurance of memristor devices. The improvements are the result of a significant reduction of cache and DRAM accesses, and the usage of non-volatile memory. The reduction of memory accesses leads to a lower latency and lower energy consumption, while the non-volatile memory reduces the static power practically to zero.



(a) NFA (b) Homogeneous automata

Fig. 5. Example notations for NFAs and homogeneous automata.

IV. Memristive Devices for Automata Processing

Automata-based processing is widely used in diverse fields, including network security [22], computational biology [23], and data mining [24]. Its hardware implementation, referred to as *automata processors (APs)*, has significant advantages over von Neumann architectures regarding throughput and energy efficiency as they enable computation-in-memory [25–27]. Memristive devices, which are the enablers of Resistive Random-Access Memories (RRAM) and computation-in-memory, are potential candidates for implementing the APs as it will be shown in this section. We will refer to this implementation as RRAM-AP. Moreover, it will be shown that RRAM-AP outperforms the two known hardware implementations of APs, being the Micron Automata Processor [25] which is based on SDRAM, and the Cache Automation [27] which is based on SRAM; we will refer to them by SDRAM-AP and SRAM-AP, respectively, to maintain the naming consistent with RRAM-AP. Next, we will first introduce basic knowledge and notations of automata. Subsequently, we propose a generic model for automata processors. Thereafter, we present RRAM-AP implementation, and show its superiority.

A. Automata Basics

A Non-deterministic Finite Automata (NFA) can be represented by a 5-tuple: $(Q, \Sigma, \delta, q0, C)$. Q represents a finite set of states (which are denoted with circles in the illustrative example of Fig. 5a), Σ is a finite set of possible input symbols (that can be used to generate an input sequence), δ is the transition function describing the set of possible transitions

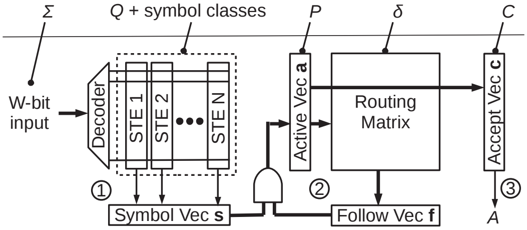


Fig. 6. General architecture for automata processors.

among the states, $q0$ is one of the states from Q and presents the *start state*, C is a subset of Q and contains the *final states* or *accepting states*; they are denoted with a double circle in the state diagram af Fig. 5a as shown for the final state S3.

During operation (i.e., execution of an input sequence), some states can be *active*; they are denoted by P. Initially, P equals to $q0$. At each processing step, the NFA consumes one symbol I from the input sequence. Based on I and δ, P is updated. Once all symbols of the input sequence are processed, the NFA output is determined by P and C. If $P \cap C \neq \emptyset$, then we say that the NFA *accepts* the input sequence; otherwise, the sequence is *rejected*. The acceptance of the input sequence can be represented by a Boolean value A.

Homogeneous automaton is a special type of NFA that is relatively easy to implement by APs [25]. It requires that a state can only be reached by transitions with the *same* input symbol(s). These input symbols belong to the *symbol class* of this state. For example, in the NFA shown in Fig. 5b, S3 can be reached by two transitions (from S1 and S2, respectively) both with the same symbol b; b belongs to the symbol class of S3. Here, the NFA shown in Fig. 5a is a homogeneous automaton and can be therefore redrawn as depicted in Fig. 5b. Note that the input symbols are only related to the *states* in homogeneous automata and not the *state transitions* as is the case for normal NFAs; e.g., the symbol b is not on the incoming edges/transition of the state S3 (see Fig. 5a) but rather within the node representing S3 (see Fig. 5b). Any NFA can be translated into its equivalent homogeneous automaton and therefore implemented using APs [25].

B. Generic Automata Processor Model

Before implementing RRAM-AP, we need to understand the key operations conducted by an AP. Therefore, we next present a generic model for APs to identify these operations. This generic model is shown in Fig. 6 and consists of three major processing steps:

1) Input symbol processing: It decodes each symbol I (presented with W bits) of the input sequence by activating only one of the 2^W wordlines, and identifies all states that have an incoming transition occurring on I. These states and the remaining sates are presented by column vectors called State Transition Elements (STEs), and are pre-configured based on Q and the corresponding symbols (symbol class). Each STE presents one state of the N states of Q. The result of this step is mapped to a vector called Symbol Vector $\mathbf{s}$.
2) Active state processing: It generates: (1) all the possible states that can be reached from the current active states P (stored in a vector called Active Vector $\mathbf{a}$) based on these states and the transition function δ (stored in the routing matrix), and stores the result in the Follow Vector $\mathbf{f}$; (2) the next active states (i.e., Active Vector) by bit-wise ANDing $\mathbf{s}$ and $\mathbf{f}$.
3) Output identification: In order to decide about the value of A (i.e., whether the input sequence is accepted or not), the intersection of $\mathbf{a}$ and the *Accept Vector* $\mathbf{c}$ (pre-configured based on C) is checked. That is, if $P \cap C \neq \emptyset$, then $A = 1$ (accept), otherwise $A = 0$ (reject).

Next we will elaborate the above three processing steps.

1) Input symbol processing: As mentioned, the purpose of this is to calculate the Symbol Vector $\mathbf{s}$ for each input symbol. This is done based on the selected row (from the 2^W rows) and the configuration of STEs. Let's assume that for each input symbol, an *Input Vector* $\mathbf{i}$ of 2^W elements is generated where only one element is high (corresponding to the selected wordline); the remaining elements are 0. In addition, assume that the configuration of STEs can be presented by a matrix $\mathbf{V}$ where each column $\mathbf{V_n}$ presents the STE of the state n. Then the nth element of the Symbol Vector $\mathbf{s}$ corresponding to $\mathbf{V_n}$ can be calculated as:

$$s[n] = \mathbf{i} \cdot \mathbf{V_n} = \sum_{k=0}^{2^W} i[k] v_n[k], \ \forall n \in [1, N] \tag{1}$$

In this equation, the addition and the multiplication represent the Logic OR and AND, respectively. For the example of Fig. 5b, if we assume $\Sigma = \{a, b, c, d\}$, then,

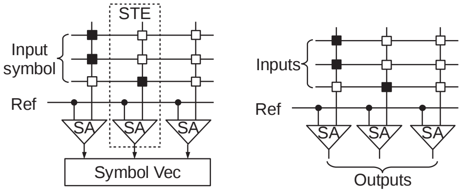
$$\mathbf{V} = \begin{bmatrix} \mathbf{V_1} & \mathbf{V_2} & \mathbf{V_3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

This means that S1's symbol class is $\{a, b, c\}$, S2's is $\{b\}$, and S3's is $\{c\}$. If we further assume that the current input symbol is b, then $\mathbf{i} = [0\ 1\ 0\ 0]$, and $\mathbf{s} = [1\ 0\ 1]$. This means that b is in the symbol classes of S1 and S3.

2) Active states processing: This step calculates the Follow Vector $\mathbf{f}$ which presents the possible states that can be reached from the current active states stored in the Active Vector $\mathbf{a}$. The transition function is implemented by the routing matrix as shown in Fig. 6, and can be conceptually presented as a two-dimensional vector $\mathbf{R}$. Hence, the nth element of Follow Vector $\mathbf{f}$ can be calculated as:

$$f[n] = \mathbf{a} \cdot \mathbf{R_n} = \sum_{i=0}^{N-1} a[i] R_n[i], \ \forall n \in [1, N]. \tag{2}$$

The interpretation of the addition and the multiplication in this equation is the same as in Equation (1). The next active states (to be also stored in the Active Vector $\mathbf{a}$) are easily calculated by using bitwise AND operation.



(a) Used as STEs (b) Used as routers

Fig. 7. Vector dot product operator used as switches and STEs.

$$a[n] = f[n] \ \& \ s[n], \ \forall n \in [1, N]. \tag{3}$$

For the example of Fig. 5b, the matrix $\mathbf{R}$ that belongs to the transit function is

$$\mathbf{R} = \begin{bmatrix}\mathbf{R_1} & \mathbf{R_2} & \mathbf{R_3}\end{bmatrix} = \begin{bmatrix}0 & 1 & 1\\ 0 & 0 & 1\\ 0 & 0 & 0\end{bmatrix}.$$

This means that S1 cannot be reached from all the states ($\mathbf{R_1}$), S2 can only be reached from S1 ($\mathbf{R_2}$), and S3 from both S1 and S2 ($\mathbf{R_3}$). For $\mathbf{a} = [1\ 0\ 0]$ (only S1 is active), $\mathbf{f} = [0\ 1\ 1]$ according to Equation (2). This means S2 and S3 are reachable states from the active states. If we assume the next input symbol is b, which leads to $\mathbf{s} = [1\ 0\ 1]$ as discussed above, then the new active vector $\mathbf{a} = [0\ 0\ 1]$ according to Equation (3). This means that S3 becomes the next active state.

3) Output identification: The output value A of NFA is easily calculated using the Active Vector $\mathbf{a}$ and the Accept Vector $\mathbf{c}$. The former stores the active states generated by the input sequence while the later stores the defined accepting states of NFA.

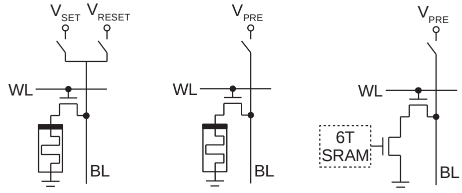
$$A = \mathbf{a} \cdot \mathbf{c}^{\top} = \sum_{n=0}^{N-1} a[n]c[n]. \tag{4}$$

$A = 1$ means that the input symbol sequence is accepted by the NFA; otherwise, the string is rejected. For the example of Fig. 5b, $\mathbf{c} = [0\ 0\ 1]$. This means only S3 is an accepting state. If we assume the same example as above ($\mathbf{a} = [0\ 0\ 1]$), then $A = 1$.

C. RRAM-AP Implementation

The automata processing model described above contains only two types of logic operations, which are vector dot product (Equation 1, 2, and 4) and vector bit-wise AND (Equation 3). In practice, we cannot implement the complete routing matrix of Equation 2, as it requires too much resource. SDRAM-AP and SRAM-AP both use hierarchical routers to implement the routing matrix. Their implementations do not support all NFA transitions; nevertheless, there is enough flexibility to route all possible transitions of typical applications [25,27]. While SDRAM-AP does not reveal many implementation details, SRAM-AP uses a two-level structure that consists of global and local switches [27]. These global and local switches also conduct vector dot product operations.

For our implementation, we adopt SRAM-AP's for the routing matrix, use the hardware structure shown in Fig. 7a for STEs, and the one in Fig. 7b both for global and local switches.



(a) Program circuit (b) RRAM cell (c) SRAM cell [27]

Fig. 8. Different implementations of a configurable bit.

The black and white boxes represent different configuration bits. Each column generates the vector dot product of the input vector and the configuration bits of this column.

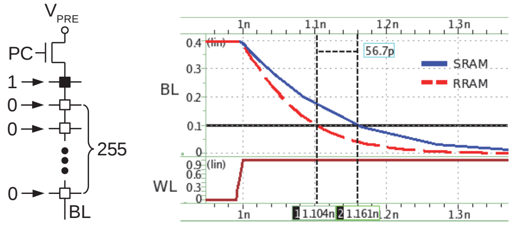
An NFA is configured to RRAM-AP by programming RRAM devices to either low or high resistance. We use one transistor and one RRAM device (1T1R) to implement a configurable bit as shown in Fig. 8b. During the configuration, the word line WL selects the row to be programmed, and the programming voltage is applied to the bit line BL as shown in Fig. 8a. The programming voltage can be either SET or RESET voltage. Logic 1 corresponds to the memristor's low resistance, and logic 0 to high resistance. The bit line is precharged before evaluation, and the word lines are selected, e.g., by the input symbols. Note that for the routing matrix, multiple word lines can be activated in parallel. The vector dot product is calculated when all the word lines are set; if all the corresponding selected cells contain a high resistance (i.e., logic 0), then the pre-charged bit line remains high, and the sense amplifier (SA) will read a logic 0 (inverted output). Similarly, if at least one of the cells contains a low resistance (i.e., logic 1), then BL will be discharged. The SA's output will subsequently be a logic 1.

The characteristics of memristors provide opportunities for RRAM-AP to outperform previous designs. For example, SRAM-AP uses eight transistors to implement the configurable bit as shown in Fig. 8c [27], whose area is much larger than the 1T1R structure. In addition, the SRAM cells also suffers from leakage power. As memristors are non-volatile devices, RRAM-AP can resume the last configured NFA after shut down and reboot without reprogramming it. On the other hand, RRAM-AP also inherits some drawbacks, such as the longer and power-hungry programming phase, and lower endurance, in comparison with SDRAM and SRAM.

D. Preliminary Results

The APs can be built by using only vector dot product and bit-wise AND operators. Except for the vector dot product operator, we assume that the remaining part of RRAM-AP is implemented in a similar way as SRAM-AP (incl. bit-wise AND, wiring, and sense amplifiers). Hence, we compare only the dot product operator. Note that SRAM-AP outperforms SDRAM-AP regarding the throughput and energy consumption; therefore, we limit our comparison to SRAM-AP.

The simulated circuit consists of a single vector dot product operator with a length of 256 as shown in Fig. 9a. We use



(a) Simulated circuit (b) SPICE simulation result

Fig. 9. SPICE simulation results of a vector dot product operator.

32 nm PTM model for CMOS transistors and ASU model [28] for RRAM. We configure RRAM's parameters based on a two-state device, similarly as presented in [29], e.g., the RRAM's high and low resistances are approximately 100 MΩ and 1 kΩ respectively; the SET and RESET threshold voltages are 1.3 V and 0.5 V. To simulate the slowest discharge process, only the first cell is configured to logic 1 (indicated by the black box), and the remaining 255 cells are configured to be 0 (indicated by white boxes). The bit line BL is pre-chared to 0.4 V (lower than RRAM's threshold voltages). When BL is discharged to 0.1 V, the sense amplifier (not included in the circuit) will read a 1. The reference voltage of the SA is set to 0.25 V.

The HSPICE simulation results are shown in Fig. 9b. The word line WL is enabled at 1 ns, and then BL starts discharging. BL's voltages in SRAM and RRAM-based designs are illustrated with solid blue line and dashed red line, respectively. The discharge time through RRAM (104 ps) is 35% less than the SRAM-based implementation (161 ps). This is mainly because transistors have relatively large intrinsic capacitance. During bit-line discharge, the RRAM cell of Fig. 8b has only one transistor in its path while the SRAM-based design has two (See Fig. 8c). The energy consumed during the charge and discharge processes is 2.09 fJ for the RRAM-based design and 5.16 fJ for the SRAM-based design. The former is 59% less than the latter. Considering that the remainder part of RRAM-AP is implemented in a similar way as SRAM-AP, RRAM-AP outperforms SRAM-AP at the chip level regarding latency, energy, and area.

V. Conclusion

In this work, we have discussed two potential applications of memristive devices and computation-in-memory, i.e., Memristive Vector Processor and RRAM Automata Processor. Memristors' unique properties provide us an important opportunity to improve conventional designs at both architectural and device level. However, the drawbacks of memristor technology, such as the impact of endurance, require further research.

References

[1] J. L. Hennessy *et al.*, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[2] S. Hamdioui *et al.*, "Memristor for computing: Myth or reality?" in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 722–731.

[3] ——, "Memristor based computation-in-memory architecture for data-intensive applications," in *DATE'15*. EDA Consortium, 2015, pp. 1718–1725.

[4] P. E. Gaillardon *et al.*, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 427–432.

[5] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC'16*. New York, NY, USA: ACM, 2016, pp. 173:1–173:6.

[6] J. J. Yang *et al.*, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, pp. 13–24, 2013.

[7] S. Furber, "Large-scale neuromorphic computing systems," *Journal of neural engineering*, vol. 13, p. 051001, 2016.

[8] X. Fu *et al.*, "A heterogeneous quantum computer architecture," in *CF'16*. ACM, 2016, pp. 323–330.

[9] H. A. Du Nguyen *et al.*, "On the implementation of computation-in-memory parallel adder," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[10] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on circuit theory*, vol. 18, pp. 507–519, 1971.

[11] D. B. Strukov *et al.*, "The missing memristor found," *nature*, vol. 453, pp. 80–83, 2008.

[12] M. Barbareschi *et al.*, "Memristive devices: Technology, design automation and computing frontiers," in *DTIS'17*. IEEE, 2017, pp. 1–8.

[13] H. A. Du Nguyen *et al.*, "Memristive devices for computing: Beyond cmos and beyond von neumann," in *25TH IFIP/IEEE International Conference on Very Large Scale Integration*. IEEE, 2017, pp. 1–8.

[14] L. Xie *et al.*, "Scouting logic: A novel memristor-based logic design for resistive computing," in *VLSI (ISVLSI), 2017 IEEE Computer Society Annual Symposium on*. IEEE, 2017, pp. 176–181.

[15] A. Danowitz *et al.*, "Cpu db: recording microprocessor history," *Communications of the ACM*, vol. 55, pp. 55–63, 2012.

[16] A. Pedram *et al.*, "Dark memory and accelerator-rich system optimization in the dark silicon era," *IEEE Design & Test*, vol. 34, pp. 39–50, 2017.

[17] K. Wu, "Fastbit: an efficient indexing technology for accelerating data-intensive science," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 556.

[18] K. K. Soni *et al.*, "Efficient string matching using bit parallelism," *International Journal of Computer Science and Information Technologies*, 2015.

[19] R. D. Cameron *et al.*, "Bitwise data parallelism in regular expression matching," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 139–150.

[20] D. Lavenier *et al.*, "Dna mapping using processor-in-memory architecture," in *Workshop on Accelerator-Enabled Algorithms and Applications in Bioinformatics*, 2016.

[21] S. Beamer *et al.*, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, pp. 137–148, 2013.

[22] F. Yu *et al.*, "Fast and memory-efficient regular expression matching for deep packet inspection," in *2006 Symposium on Architecture For Networking And Communications Systems*, Dec 2006, pp. 93–102.

[23] I. Roy *et al.*, "Discovering motifs in biological sequences using the micron automata processor," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 13, pp. 99–111, Jan. 2016.

[24] K. Wang *et al.*, "Sequential pattern mining with the micron automata processor," in *CF'16*. ACM, 2016, pp. 135–144.

[25] P. Dlugosch *et al.*, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 3088–3098, Dec 2014.

[26] C. Bo *et al.*, "Entity resolution acceleration using the automata processor," in *Big Data*, Dec 2016, pp. 311–318.

[27] A. Subramaniyan *et al.*, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 259–272.

[28] P. Y. Chen *et al.*, "Compact modeling of rram devices and its applications in 1t1r and 1s1r array design," *IEEE Transactions on Electron Devices*, vol. 62, pp. 4022–4028, Dec 2015.

[29] X. A. Tran *et al.*, "High performance unipolar aloy/hfox/ni based rram compatible with si diodes for 3d application," in *2011 Symposium on VLSI Technology - Digest of Technical Papers*, June 2011, pp. 44–45.

Time-division Multiplexing Automata Processor

Jintao Yu Hoang Anh Du Nguyen Muath Abu Lebdeh Mottaqiallah Taouil Said Hamdioui
Laboratory of Computer Engineering, Delft University of Technology
Delft, the Netherlands
{J.Yu-1, H.A.DuNguyen, M.F.M.AbuLebdeh, M.Taouil, S.Hamdioui}@tudelft.nl

Abstract—**Automata Processor (AP) is a special implementation of non-deterministic finite automata that performs pattern matching by exploring parallel state transitions. The implementation typically contains a hierarchical switching network, causing long latency. This paper proposes a methodology to split such a hierarchical switching network into multiple pipelined stages, making it possible to process several input sequences in parallel by using time-division multiplexing. We use a new resistive RAM based AP (instead of known DRAM or SRAM based) to illustrate the potential of our method. The experimental results show that our approach increases the throughput by almost a factor of 2 at a cost of marginal area overhead.**

Index Terms—**time-devision multiplexing, automata, parallel processing**

I. Introduction

Finite State Automata (FSA) is a commonly used computing model to match sequences with predefined patterns; examples are network security [1], bioinformatics [2], and artificial intelligence [3]. It can also be used for other functions, such as edit distance calculation [4, 5], tree structure traversal [6], and path recognition [7]. However, executing FSA using von Neumann machines such as CPUs and GPUs is generally not efficient. For example, applications such as Snort [1] and Protomata [2] contain thousands of predefined patterns, which easily exceed the size of first-level caches. Moreover, they have a bad data locality as states can transit to any other state. In addition, automata processing is difficult to parallelize due to a strong input sequence dependency [8]. Hence, there is a need of dedicated and efficient FSA hardware implementations.

Many solutions have been proposed. FPGA-based accelerators [9, 10] are still limited by the FPGA's architecture and capacity. Therefore, their throughput is low and their scalability is limited as compared to ASIC designs [11]. Custom hardware accelerators [11, 12] for FSA can avoid such problems by providing a large memory and having customized optimizations. For instance, Micron Automata Processor (MAP) (based on DRAM technology) [11] stores up to 48k states on a single chip, which is large enough for configuring the automata of most applications including Snort and Protomata [13]. It processes one input symbol in each cycle [11]. For pattern matching applications, this means that all the patterns are matched simultaneously. As a result, these accelerators achieve a much higher throughput as compared with CPU or GPU implementations [12, 13]. Unified Automata Processor (UAP) [12] contains multiple cores that are simplified for automata processing. It processes multiple input streams simultaneously to increase the throughput. For each input stream, however, it processes activated states sequentially. Therefore, its throughput degrades when many states are active. HAWK [14] and HARE [15] use logic gates for matching. They process multiple input symbols of a single input stream in each clock cycle, thus achieving a higher throughput. However, they are designed for regular expression matching only. To the best of our knowledge, Cache Automaton [16] has the highest single-stream throughput reported. It is based on SRAM technology and is much faster than DRAM-based MAP [11]. Cache Automaton uses a two-stage pipeline to process an input symbol. One of these stages contains a hierarchical switching network that consists of global and local routers; the switching network implements the automata's state transitions. It is relatively complex as the number of states can be huge. Therefore, this pipeline stage is the bottleneck that limits Cache Automaton's speed and throughput. RRAM-AP concept [17] shows the potential of building an automata accelerator using Resistive Random Access Memory (RRAM) arrays, which are even faster than SRAM arrays. However, a complete design was not given.

In this paper, we further improve the throughput of automata accelerators. The main contributions of this paper are:

- It proposes a methodology to process multiple input streams simultaneously with a higher frequency using Time-Division Multiplexing (TDM). We realize this by pipelining the hierarchical switching network and adding multiplexing circuitry. At any moment, each stage processes a symbol of a different input stream without affecting the other streams. Although the processing time for a single input stream remains nearly the same, multiple input streams are processed in parallel. Therefore, the overall throughput is increased significantly at the cost of marginal area overhead.
- It implements an RRAM-based automata accelerator and integrates the proposed TDM methodology in it. Note that the methodology can be implemented with any technology (DRAM, SRAM and RRAM).
- It evaluates the performance and area overhead of the TDM RRAM-based automata accelerator. In addition, it compares the automata accelerator to existing solutions.

The rest of the paper is organized as follows. In Section II, we explain the basic principle of a popular automata accelerator type. Section III presents the TDM methodology and how it can be integrated in an RRAM-based automata accelerator.

This work was supported by the European Unions Horizon 2020 Research and Innovation Program through the project MNEMOSENE (Grant 780215).

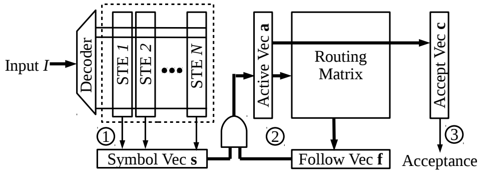


Fig. 1. General architecture of Automata Processors [17].

Section IV evaluates the performance and the area overhead of the TDM RRAM-based automata accelerator. Section V contains a brief discussion. Finally, VI concludes the paper.

II. Background

In this section, we provide a background on automata processors and highlight their performance bottleneck.

A. Automata Processors

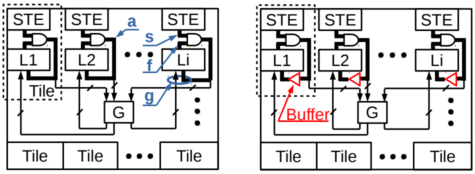
The TDM methodology proposed in this paper can only be applied to a specific type of automata accelerators, referred to as *Automata Processors* (APs); they have similar working principle as MAP. MAP is one of the most successful hardware implementations of Non-deterministic Finite Automata (NFA), whose high efficiency has been proved by many researches [2–7, 12]. Recent works such as Cache Automaton [16] and RRAM-AP [17] intend to improve MAP by using different memory technologies while maintaining its basic structure. These designs have the following features in common:

- They all model *homogeneous automata*; in these automata, a state can only be reached by transitions with the *same* input symbol(s). Any NFA can be translated into its equivalent homogeneous automaton and therefore implemented using APs [11].
- They all use memory arrays in the implementation. MAP uses DRAM, Cache Automaton uses SRAM, and RRAM-AP uses RRAM.
- They all use hierarchical switching networks for implementing the state transitions.

The generalized architecture of APs is shown in Fig. 1 [17]. An input symbol I is processed using three major steps:

① **Input symbol matching.** In this step, all states that have an incoming transition occurring on I are identified. The N states are presented by column vectors called State Transition Elements (STEs) which are pre-configured based on the targeted automaton. Each input symbol activates one wordline and the content in an STE cell specifies for that particular state whether the current input symbol has an incoming transition. The result of this step is mapped to a vector called Symbol Vector $\mathbf{s}$.

② **Active state processing.** It generates: (1) all the possible states that can be reached from the current active states (stored in Active Vector $\mathbf{a}$) based on the transition function (stored in the routing matrix), and stores the result in the Follow Vector $\mathbf{f}$; (2) the next active states (i.e., Active Vector) by bit-wise ANDing $\mathbf{s}$ and $\mathbf{f}$.



(a) Original (b) Adapted for TDM

Fig. 2. Adapting the hierarchical switching network for TDM.

③ **Output identification.** In order to decide whether the input sequence is accepted or not, the intersection of $\mathbf{a}$ and pre-configured Accept Vector $\mathbf{c}$ is checked; it contains the states that the automaton accepts.

Among these steps, Step ② is the most critical and time consuming. In the existing designs, this step is implemented using a routing matrix. In the next subsection, we explain its working principle.

B. Routing Matrix

As the STE matrix can be huge, it is fragmented across the entire chip and we refer to each fragment as a tile. To determine the next states, existing AP designs use hierarchical switching networks to implement the routing matrix. For example, Cache Automaton uses a network that consists of global and local switches as shown in Fig. 2a [16]. If the communication takes place inside a tile, only local routing is used; otherwise, global routing is used as well.

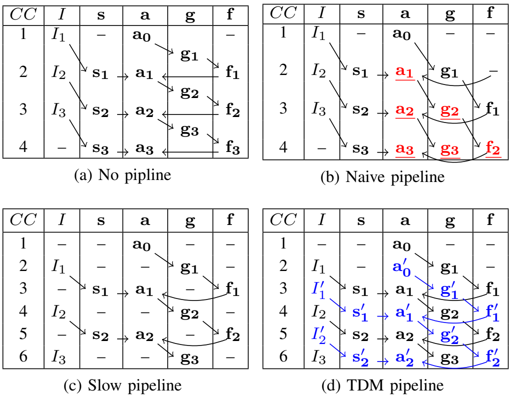
In the figure, the Active Vector $\mathbf{a}$ is divided into several groups. Each group has some signals that enter global switches (represented by the box G in the figure) which are used for inter-tile communication. The outputs of the global switches combined with the initial vector $\mathbf{a}$ forms a vector (referred to as *Global Vector* $\mathbf{g}$) and is used as the input to the local switches, which are presented by boxes *L1*, *L2*, and *L3*. The outputs of local switches form the Follow Vector $\mathbf{f}$. As the global switches are used to form an interconnection between the different tiles, they suffer from long global wires. They affect the latency of the *active state processing* step (Step ② in Section II-A) as it is determined by the sum of the latency of global and local switches. It is the performance bottleneck of MAP and Cache Automaton. In the following section, we will show how to improve its performance using pipelining and TDM.

III. Time-Division Multiplexing AP

In this section, we first introduce the TDM methodology. Thereafter, we present the hardware implementation required to support TDM. Finally, we provide the implementation details of the RRAM-AP combined with TDM.

A. Methodology

In this section, we first examine the data flow of an AP without pipelining and then apply TDM on it. We use Fig. 3a to explain Cache Automaton's working principle. Each row of



CC	I	$\mathbf{s}$	$\mathbf{a}$	$\mathbf{g}$	$\mathbf{f}$
1	I_1	–	$\mathbf{a}_0$		–
				$\mathbf{g}_1$	
2	I_2	$\mathbf{s}_1$	$\mathbf{a}_1$		$\mathbf{f}_1$
				$\mathbf{g}_2$	
3	I_3	$\mathbf{s}_2$	$\mathbf{a}_2$		$\mathbf{f}_2$
				$\mathbf{g}_3$	
4	–	$\mathbf{s}_3$	$\mathbf{a}_3$		$\mathbf{f}_3$

(a) No pipline

CC	I	$\mathbf{s}$	$\mathbf{a}$	$\mathbf{g}$	$\mathbf{f}$
1	I_1	–	$\mathbf{a}_0$	–	–
2	I_2	$\mathbf{s}_1$	$\underline{\mathbf{a}_1}$	$\mathbf{g}_1$	–
3	I_3	$\mathbf{s}_2$	$\underline{\mathbf{a}_2}$	$\underline{\mathbf{g}_2}$	$\mathbf{f}_1$
4	–	$\mathbf{s}_3$	$\underline{\mathbf{a}_3}$	$\underline{\mathbf{g}_3}$	$\underline{\mathbf{f}_2}$

(b) Naive pipeline

CC	I	$\mathbf{s}$	$\mathbf{a}$	$\mathbf{g}$	$\mathbf{f}$
1	–	–	$\mathbf{a}_0$	–	–
2	I_1	–	–	$\mathbf{g}_1$	–
3	–	$\mathbf{s}_1$	$\mathbf{a}_1$	–	$\mathbf{f}_1$
4	I_2	–	–	$\mathbf{g}_2$	–
5	–	$\mathbf{s}_2$	$\mathbf{a}_2$	–	$\mathbf{f}_2$
6	I_3	–	–	$\mathbf{g}_3$	–

(c) Slow pipeline

CC	I	$\mathbf{s}$	$\mathbf{a}$	$\mathbf{g}$	$\mathbf{f}$
1	–	–	$\mathbf{a}_0$	–	–
2	I_1	–	$\mathbf{a}'_0$	$\mathbf{g}_1$	–
3	I'_1	$\mathbf{s}_1$	$\mathbf{a}_1$	$\mathbf{g}'_1$	$\mathbf{f}_1$
4	I_2	$\mathbf{s}'_1$	$\mathbf{a}'_1$	$\mathbf{g}_2$	$\mathbf{f}'_1$
5	I'_2	$\mathbf{s}_2$	$\mathbf{a}_2$	$\mathbf{g}'_2$	$\mathbf{f}_2$
6	I_3	$\mathbf{s}'_2$	$\mathbf{a}'_2$	$\mathbf{g}_3$	$\mathbf{f}'_2$

(d) TDM pipeline

Fig. 3. Pipelining of global and local switches for APs.

the table represents a clock cycle (CC), while the columns contain the values of the input symbol I and key vectors introduced in Section II (i.e., $\mathbf{s}$, $\mathbf{a}$, $\mathbf{g}$, and $\mathbf{f}$). The arrows indicate the data flow; for example, the arrow from I_1 to $\mathbf{s}_1$ means that I_1 determines the value of $\mathbf{s}_1$, and the arrows from $\mathbf{s}_1$ and $\mathbf{f}_1$ to $\mathbf{a}_1$ mean that the value of $\mathbf{a}_1$ is derived based on those of $\mathbf{s}_1$ and $\mathbf{f}_1$. Dashes (–) represent don't cares. It is important to note that the vector $\mathbf{g}$ is generated between each two cycles; e.g., in Fig. 3a, $\mathbf{a}_0$ is initialized at $CC = 1$, $\mathbf{s}_1$, $\mathbf{f}_1$, and $\mathbf{a}_1$ are generated at $CC = 2$, while $\mathbf{g}_1$ is generated between $CC = 1$ and $CC = 2$.

To increase the clock frequency of the AP, we can convert the routing matrix into a pipeline by processing the vector $\mathbf{g}$ in a full clock cycle. However, without other necessary modifications, the AP will produce wrong results as shown in Fig. 3b. When both the global and local switches work as successive pipeline stages, the Follow Vector, e.g. $\mathbf{f}_1$, is only ready two cycles after the Active Vector $\mathbf{a}_0$. Meanwhile, two input symbols (I_1 and I_2) have entered the AP. Therefore, the dependency between the two input symbols has been destroyed. As a result, all the values colored in red (underline) are incorrect, including $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{a}_3$.

We can solve this problem by decreasing the input frequency as shown by Fig. 3c. If an input symbol is processed every two cycles instead of every cycle, then it can match the speed of the switching network. The dependency among all the values are the same as Fig. 3a. Therefore, the results are correct. However, it requires more cycles to process all the three input symbols (which is not completely shown in Fig. 3c). All the stages make meaningful use (and produce results) of only half of the cycles; e.g., local switches produce outputs $\mathbf{f}_1$ and $\mathbf{f}_2$ at $CC = 3$ and 5 while they are idle at $CC = 4$ and 6.

To make full use of the hardware, we use TDM methodology and insert another input sequence to the original one as shown in Fig. 3d. The values related to the second sequence are marked with a prime and colored in blue, e.g., I'_1. Both sequences do not interfere with each other as there are no arrows connecting black and blue values. The switches process one input sequence in odd cycles and one in even cycles.

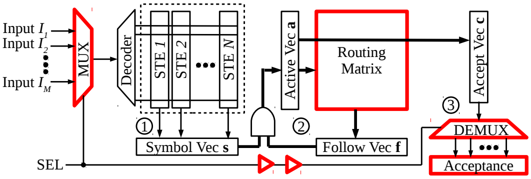


Fig. 4. Architectural modifications required to support TDM in AP.

Fig. 3 clearly shows that the TDM methodology may improve the performance. Both Fig. 3a and Fig. 3d process one symbol every cycle; nevertheless, the clock frequency of the implementation in Fig. 3d can be much higher. The length of the clock period for Fig. 3d equals the latency of a single switching operation (i.e. the worse case latency between the global and local switches), while the one for Fig. 3a is approximately twice as long (sum of global and local switches). Although we use a two-phase switching network as an example, the TDM can be generalized for networks with more phases. The number of different active input sequences equals the number of switching phases. Note that the proposed TDM scheme is independent from the memory technology it uses; therefore, it can be applied to MAP (based on DRAM), Cache Automaton (based on SRAM), and RRAM-AP (based on RRAM).

B. Hardware Adaption

To support TDM in APs, we need to modify several components of the architecture as indicated by the red colored (bold) components in Fig. 4. First, a multiplexer (MUX) is added prior to Step ① (input symbol matching). Assuming the switching network works in M phases, the MUX merges M input streams into a single one by fetching in each cycle a symbol from an input stream in a round-robin fashion. For example, the example provided in Fig. 3d shows that the MUX of Fig. 4 will have two input streams I and I'. The merged sequence will be decoded and processed in the same way as executed in a normal AP.

Next, the routing matrix (implemented by a hierarchical switching network) needs to be updated as shown in Fig. 2b for two phases (M=2). The control signals of global and local switches (not shown in the figure) should be changed due to the additional $M - 1$ pipelines. Extra buffer stages have to be inserted between the Active Vector $\mathbf{a}$ and the local switches in order to balance all paths between Active Vector $\mathbf{a}$ and Follow Vector $\mathbf{f}$ to two clock cycles.

Finally, a demultiplexer (DEMUX) is added to split the acceptance bit stream into multiple ones as shown in Fig. 4. Each output stream corresponds to the input stream that is provided two cycles earlier, due to one cycle latency of Step ① to produce $\mathbf{a}$ and one cycle latency of Step ③ to produce *Acceptance*. Therefore, DEMUX can share the same control signals with MUX but delayed with two buffers.

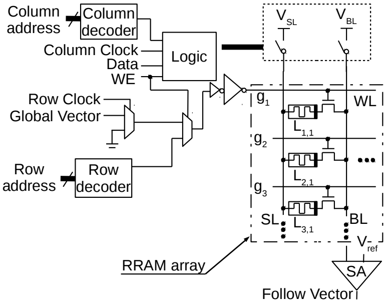


Fig. 5. Local switch implementation.

C. RRAM-based Implementation

We develop an RRAM-based AP to demonstrate the proposed TDM methodology. Its top level structure is shown in Fig. 2b, which generally follows the performance-optimized design used in Cache Automaton [16]:

- The chip contains 64 tiles, 8 global switches, and the circuitry enabling TDM (a multiplexer, a demultiplexer, and two buffers between them; see also Fig. 4).
- A tile consists of an *STE array* (containing 256 STEs and a decoder), a local switch, an Accept Vector, a bit-wise AND gate, and a buffer (storing 256 bits).
- The sizes of global and local switches are 128×128 and 280×256, respectively.

The STE arrays, global and local switches, and the Accept Vector are all implemented with one-Transistor-one-RRAM (1T1R) arrays. These arrays compute a vector-matrix product where the binary vector is applied as input to the word lines and the binary configuration matrix is stored in RRAMs [17]. This operation is performed by special read instructions. For example, for the RRAM array in a local switch as shown in the dashed box in Fig. 5, this operation is between Global Vector $\mathbf{g}$ and the array's configuration $\mathbf{L}$. During a read operation, the bit lines are first precharged to a high voltage. Subsequently, $\mathbf{g}$ is applied to the word lines; note that multiple word lines can be activated simultaneously. Each column computes the inner product of $\mathbf{g}$ and a column vector of $\mathbf{L}$. If at least one RRAM cell is configured as a low resistance (logic 1 in the configuration matrix) and its word line is active (logic 1 in the input vector), then the bit line discharges to a low voltage; otherwise, the bit line remains high. Note that before any processing, the RRAM arrays must be configured.

In this paper, we present the design of a local switch as an example. The STE arrays and global switches are implemented in a similar way. Fig. 5 illustrates our implementation of the local switch. It consists of a 1T1R memory array and peripheral circuits around it. Its bit line (BL) and source line (SL) are connected to column voltage drivers. The logic block of Fig. 5 is responsible for providing control signals and setting up the circuit in one of the two modes: configurable mode or operational mode; this depends on the input control signals shown in Table I. In the configurable mode, the write enable (WE) signal is 1, and the local switch is initialized (i.e., configured) based on the targeted automaton. During the configuration, either SET or RESET voltages (V_{SET} and V_{RESET}) are applied to the RRAM device by the column voltage drivers based on the values in Data signal. As word line (WL) is long (256-bit wide), we assume a single word is written in multiple cycles (e.g., 64 bits a time) by using a column select signal. In case the column select value is zero, the cells in those lines are kept floating during writing. In the operational mode, WE $= 0$ and the memory is used for reading; i.e., it generates the value of the next Follow Vector based on the Global Vector (see Section III-A).

TABLE I
COLUMN AND ROW VOLTAGES IN A LOCAL SWITCH

Inputs			Outputs		
WE	Column sel.	Data	V_{SL}	V_{BL}	V_{WL}
1	1	1	GND	V_{SET}	Row decoder output
		0	V_{RESET}	GND	
	0	–	Float	Float	
0	–	–	GND	V_{Read}	Global Vector

It is worth noting that the Accept Vector is implemented together with the local switches using an extra column in the array. As there are 64 tiles, the outputs of these columns in all the tiles together are used to generate the acceptance bit via a 64-to-1 OR operation. This operation is implemented using three levels of 4-input NOR, NAND, and OR gates.

IV. Evaluation

In this section, we first present the simulation setup. Subsequently, we present the performance results and area overhead. Note that the latency of each step listed in Section II-A can be divided into several parts:

① **Input symbol matching.** It equals to the latency of an STE array operation, which includes symbol decoding and the operation of RRAM array;

② **Active state processing.** It consists of global switching phase and local switching phase. The former includes the latency of an AND gate, signal transferring via a global wire, and a global switch. The later includes global wire transmission and a local switch;

③ **Output identification.** It consists of the latency of Accept Vector (= local switch) and 64-to-1 OR operations.

A. Simulation Setup

We conducted SPICE simulation to measure the latency of these operations mentioned above. We assume that each memory cell of the 1T1R array contains an Pd/Al_2O_3/HfO_2/NiO_x/Ni RRAM device [18], with a high and a low resistance of 10^9 and 10^3 Ω, respectively. Its top and bottom electrodes are connected to the bit line and the pass transistor and have a width of $40\,\mathrm{nm}$ and $80\,\mathrm{nm}$, respectively. The RRAM device is simulated using the ASU model [19] configured using the device characteristics of [18].

For the CMOS part of the AP implementation, we use TSMC $40\,\mathrm{nm}$ technology. To simplify and speed up the simulation, only one complete row and column of the STE arrays,

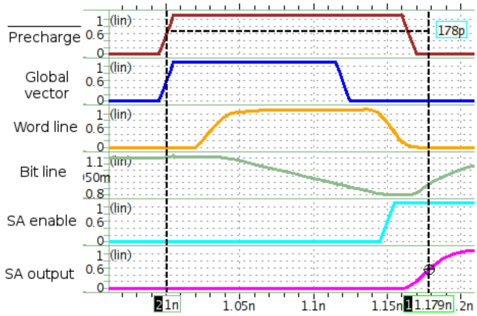


Fig. 6. SPICE simulation result of the local switch.

global, and local switches are simulated. In such columns, only one cell is configured to a low resistance. During the computation of an inner product, this configuration results in the highest discharge time [17] and therefore, it determines the minimum clock period. To guarantee a correct sense amplifier output, we need to make sure that the difference between the bit line and reference voltage V_{Ref} is larger than ΔV_{min}, which is the minimum voltage difference that the sense amplifier requires to operate correctly. When the RRAM cells in a column are all configured as logic 0, the voltage drop in the bit line is negligible due to the high resistance of the RRAM devices. As a result, we set $V_{dd} = 1.1\,\text{V}$, $V_{Ref} = 0.95\,\text{V}$, and $\Delta V_{min} = 150\,\text{mV}$. The sense amplifier design is adopted from [20]. With respect to the latency of global wires, we follow the assumption of [16]; i.e., their pitch and length are $1\,\mu\text{m}$ and $1.5\,\text{mm}$, respectively, with a latency of $66\,\text{ps/mm}$. Therefore, the latency introduced by the global wire is $99\,\text{ps}$.

We use Cadence Virtuoso [21] to place and route the sense amplifier, column and row drivers, and the buffer, and measure their area. The area of the other digital components, including the AND gate and the decoders, are acquired from Cadence Genus [21]. For example, we describe the behavior of the peripheral circuit in Table I using Verilog and subsequently synthesize it using Genus. Note that Genus reports only the total area of the cells. To be on the safe side, we add a 25 % extra overhead to account for routing.

B. Performance Results

Fig. 6 shows the simulation result of an operation in the local switch, i.e., the inner product between the Global Vector g and a configuration vector. The bit line is first precharged to V_{dd}, which is controlled by the active low signal $\overline{Precharge}$. Then, g is used to activate the word lines. As a result, the bit line starts to discharge as one cell has a low resistance path. After a while, the sense amplifier is enabled and it finally generates a positive output. The period between the rising edges of g and sense amplifier's output is the latency of the local switch; it is approximately equal to 178 ps.

Similarly, other simulation shows that the latency of an STE array, an AND gate, a global switch, and a 64-to-1 OR gate are 258 ps, 11 ps, 129 ps, and 32 ps, respectively. Therefore, the latency of each step can be decided:

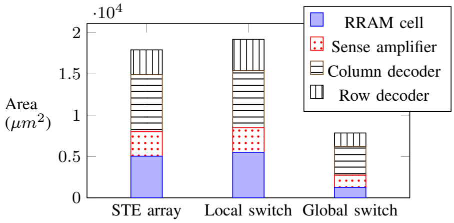


Fig. 7. Area breakdown of STE array and switches.

① 258 ps.
② Global switching phase: $11+99+129=239$ ps. Local switching phase: $99+178=277$ ps.
③ $178+32=210$ ps

The clock period of TDM RRAM-AP is determined by the pipeline stage with the highest latency. Without TDM, RRAM-AP's clock period is the sum of the latency of the global and local switching phases, i.e., $239+277=516$ ps. With TDM, the clock period equals the latency of the local switching phase, i.e., 277 ps. Therefore, the TDM methodology leads to a frequency and throughput improvement of 1.86×.

C. Overhead

Implementing TDM requires additional hardware. In this subsection, we evaluate this overhead. NVSim [22] estimates the area of a 1T1R cell using the following equation:

$$Area_{1T1R} = 3(W/L + 1)(F^2)$$

where $W/L = 3$ is the width-length ratio of the access/pass transistor and $F = 80\,\text{nm}$ the feature size.

The area breakdown of the area of STE, local switch and global switch is shown in Fig. 7. For each memory, the area of the RRAM cells, sense amplifiers, and the column and the row decoder are included. Note that the drivers and combinational logic are considered as part of the decoders. We first observe that the area of the STE array and local switch are similar, as they have approximately the same number of rows and columns. Second, the RRAM cells only contribute to a small proportion of the total area due to the small RRAM feature size. Third, the column decoder is relatively large as it also contains the control logic block shown in Fig. 5.

Based on the result of Fig. 2b, we can estimate the total area of our AP design; it is given in Table II. The first column lists the name of the components, and the second and third columns indicate the size and area of the component, respectively. The fourth and fifth columns present how many of them are used in our AP chip and their combined area. The relative area of the components with respect to the whole chip area is listed in the last column. The first row (MUX+) represents the multiplexer, demultiplexer, and the buffers between them. The buffers are inserted between the AND gates and the local switches (see Fig. 2b). The other rows contain the memories described above and the global wires.

TABLE II
Component Area of TDM RRAM-AP

Component	Array size	Area ($\mu\mathrm{m}^2$)	#	Total area (mm^2)	%
Global switch	128×128	7842	8	0.063	2.5 %
MUX+*	1×8	134.6	1	0.000	0.0 %
STE array	256×256	17907	64	1.146	45.4 %
Local switch	280×256	19168	64	1.227	48.6 %
Accept Vector	280×1	59.74	64	0.004	0.2 %
AND gate	1×256	271.0	64	0.017	0.7 %
Buffer*	1×256	1091	64	0.083	2.8 %
* Overhead introduced by TDM			Sum	2.527	100 %

The area overhead introduced by TDM includes the area of the MUX+ circuits and the buffers (denoted by a star (*) in the table), and does not exceed 2.8 %. The total area of our AP chip would be $3.16\,\mu\mathrm{m}^2$ considering 25 % routing overhead.

V. Discussion

A. TDM Methodology

Introducing TDM to APs increases their throughput significantly. The RRAM-AP design presented in Section IV-B has a shortest path of 277 ps. Assuming that the chip operates at a frequency of 3.0 GHz, its throughput will be 24.0 Gbps as each input symbol is 8 bit wide. This RRAM-AP design outperforms the state-of-the-art designs as indicated by Table III. Compared to Cache Automaton, a throughput increase of 53 % can be achieved at 26 % less area.

TDM can be applied to APs with any memory technology. In Cache Automaton, which is based on SRAM technology, the latency of the global and local switch phases equal 227 ps and 263 ps, respectively [16]. When TDM is applied to it, a similar frequency and throughput improvement of 1.86× can be expected due to a similar design[1].

The area overhead introduced by TDM is marginal. This is because that TDM only requires several minor modifications to the hardware, such as additional multiplexer and buffers. The majority of the design, such as the STEs, global, and local switches remains the same. Therefore, we expect that TDM's energy overhead is marginal as well.

B. Applicability

Many FSA applications require the processing of multiple input streams. Their throughput can be improved by using the TDM methodology. For instance, Snort is a network security application which matches data packages with particular patterns (called *rules*) to detect viruses and attacks [1]. The processing of multiple input sequences (i.e., data packages) is common when it is deployed to protect a local network. Similarly, Protomata analyzes protein samples against amino acid patterns called *motifs* [2]. Usually, there are many samples to be analyzed. Other examples include natural language processing [3], string matching [4], and path recognition [7]. This methodology can also be used in conjunction with Subramaniyanet's method to accelerate a single input stream [8].

TABLE III
Comparison Between TDM RRAM-AP and The State-of-the-Art

Designs	Frequency (GHz)	Throughput (Gbps)	Area (mm^2)
HARE (w=32) [15]	1.0	3.9	80
UAP [12]	1.2	5.3	5.67
Cache Automaton [16]	2.0	15.6	4.3
TDM RRAM-AP (this work)	3.0	24.0	3.16

VI. Conclusion

In this paper, we proposed a methodology of pipelining APs with TDM technique to improve their throughput. We developed an RRAM-based AP design to prove the concept. This prototype exhibits 1.86× performance improvement with 2.8% area overhead. The proposed methodology can be applied to all the AP designs and may benefit a wide variety of applications.

References

[1] Roesch, "Snort - lightweight intrusion detection for networks," in *LISA '99*. Berkeley, CA, USA: USENIX Association, 1999, pp. 229–238.

[2] Roy *et al.*, "High performance pattern matching using the automata processor," in *IPDPS'16*. IEEE, May 2016, pp. 1123–1132.

[3] Zhou *et al.*, "Brill tagging on the micron automata processor," in *International Conference on Semantic Computing*, 2015, pp. 236–239.

[4] Tracy *et al.*, "Nondeterministic finite automata in hardware-the case of the levenshtein automaton," *ASBD'15*, 2015.

[5] Roy *et al.*, "Finding motifs in biological sequences using the micron automata processor," in *IPDPS '14*. IEEE, 2014, pp. 415–424.

[6] Tracy *et al.*, "Towards machine learning on the automata processor," in *High Performance Computing*. Springer, 2016, pp. 200–218.

[7] Wang *et al.*, "Using the automata processor for fast pattern recognition in high energy physics experiments – a proof of concept," *Nucl Instrum Methods Phys Res A*, vol. 832, pp. 219 – 230, 2016.

[8] Subramaniyan *et al.*, "Parallel automata processor," in *ISCA '17*. New York, NY, USA: ACM, 2017, pp. 600–612.

[9] Wang *et al.*, "Min-max: A counter-based algorithm for regular expression matching," *TPDS*, vol. 24, pp. 92–103, Jan 2013.

[10] Yang *et al.*, "High-performance and compact architecture for regular expression matching on fpga," *TC*, vol. 61, pp. 1013–1025, July 2012.

[11] Dlugosch *et al.*, "An efficient and scalable semiconductor architecture for parallel automata processing," *TPDS*, vol. 25, pp. 3088–3098, 2014.

[12] Fang *et al.*, "Fast support for unstructured data processing: The unified automata processor," in *MICRO-48 '15*, Dec 2015, pp. 533–545.

[13] Wadden *et al.*, "Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *IISWC*, 2016.

[14] Tandon *et al.*, "Hawk: Hardware support for unstructured log processing," in *ICDE'16*. IEEE, May 2016, pp. 469–480.

[15] Gogte *et al.*, "Hare: Hardware accelerator for regular expressions," in *MICRO-49 '16*. New York, NY, USA: ACM, Oct 2016, pp. 1–12.

[16] Subramaniyan *et al.*, "Cache automaton," in *MICRO-50 '17*. New York, NY, USA: ACM, Oct 2017, pp. 259–272.

[17] Yu *et al.*, "Memristive devices for computation-in-memory," in *DATE'18*. IEEE, March 2018, pp. 1646–1651.

[18] Dong *et al.*, "Demonstrate high roff/ron ratio and forming-free rram for rfpga application based on switching layer engineering," in *IEEE 12th International Conference on ASIC (ASICON)*, Oct 2017, pp. 851–854.

[19] Chen *et al.*, "Compact modeling of rram devices and its applications in 1t1r and 1s1r array design," *TED*, vol. 62, pp. 4022–4028, Dec 2015.

[20] Agbo *et al.*, "Comparative analysis of rd and atomistic trap-based bti models on sram sense amplifier," in *DTIS'15*, April 2015, pp. 1–6.

[21] Cadence Design Systems, "Tools," accessed: 2018-11-27. [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools.html

[22] Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *TCAD*, vol. 31, July 2012.

[1] In Cache Automaton, four STEs share an SA to save area. Therefore, the *input symbol matching* step (Step ① in Section II-A) has a much longer latency than the local switching phase. Here, we assume no SA sharing and Step ①'s latency is smaller than the one of the local switching phase

C

Publications - Design Automation

The content of this chapter consists of the following research articles:

1. **J. Yu**, R. Nane, A. Haron, S. Hamdioui, H. Corporaal, K. L. M. Bertels, *Skeleton-based Design and Simulation Flow for Computation-in-Memory Architectures*, The 12th IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'16), Beijing, China, July 2016, pp. 165-170.

2. **J. Yu**, R. Nane, I. Ashraf, M. Taouil, S. Hamdioui, H. Corporaal, K. L. M. Bertels, *Skeleton-based Synthesis Flow for Computation-In-Memory Architectures*, IEEE Transactions on Emerging Topics in Computing (TETC), Volume 8, Issue 2, 2020, pp. 545-558.

3. **J. Yu**, M. Abu Lebdeh, H. A. Du Nguyen, M. Taouil, S. Hamdioui, *APmap: An Open-Source Compiler for Automata Processors*, submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), undergoing a minor reversion.

Skeleton-Based Design and Simulation Flow for Computation-In-Memory Architectures

Jintao Yu Razvan Nane Adib Haron Said Hamdioui Henk Corporaal* Koen Bertels
Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
*Eindhoven University of Technology, Postbus 513, 5600 MB Eindhoven, The Netherlands
{J.Yu-1, R.Nane, M.A.B.Haron, S.Hamdioui, K.L.M.Bertels}@tudelft.nl
H.Corporaal@tue.nl

ABSTRACT

Memristor-based Computation-in-Memory is one of the emerging architectures proposed to deal with Big Data problems. The design of such architectures requires a radically new automatic design flow because the memristor is a passive device that uses resistance to encode its logic value. This paper proposes a design flow for mapping parallel algorithms on the CIM architecture. Algorithms with similar data flow graphs can be mapped on the crossbar using the same template containing scheduling, placement, and routing information; this template is named *skeleton*. By configuring such a skeleton with different pre-designed circuits, we can build CIM implementations of the corresponding algorithms in that class. This approach does not only map an algorithm on a memristor crossbar, but also gives an estimation of its performance, area, and energy consumption. It also supports user-defined constraints and parallel SystemC simulation. Experimental results demonstrate the feasibility and the potential of the approach.

1. INTRODUCTION

Big Data Analytics is becoming increasingly difficult to solve using classical Von Neumann-based computer architectures because of limited bandwidth (due to memory-access bottlenecks), energy inefficiency and limited scalability (due to CMOS technology). Computation-in-Memory (CIM)-based [1, 2] architectures address the aforementioned problems by enabling in-memory computations using non-volatile memristor technology [3, 4]. They have huge potential and they could outperform the state-of-the-art with orders of magnitude [2, 5]. Exploring the potential of such architectures and appropriately evaluating their performance and scalability for larger applications require automatic flows and methods that efficiently map high-level algorithmic description to low-level memristor crossbar.

VLSI (Very-Large-Scale Integration) CAD (Computer Aided Design) flows for CMOS-based hardware solutions are not applicable to memristor-based CIM because of different signal propagation styles. In CMOS circuits, logic values are represented by the voltage; they propagate along wires implicitly and the propagation finishes within one clock cycle [6]. Because memristors are passive devices that use resistance to encode logic values, data has to be copied by controllers explicitly. This requires specific commands and several clock cycles; In addition, it depends not only on the relative positions of the source and sink [7], but also on their [1]orientations on the crossbar. In conventional VLSI CAD flows, placement and routing are performed based on the High-Level Synthesis (HLS) scheduling results [8]. However, in memristor-based CIM, placement and routing information is required before scheduling can be performed. As a consequence, a new methodology is required to appropriately design a memristor-based CIM architecture.

In this work, we propose a design and simulation flow that performs scheduling, placement, and routing simultaneously; the flow is based on the *algorithmic skeleton* [9] concept, or a *skeleton* in short. A skeleton provides an implementation template for a specific class of algorithms that have similar Data Flow Graph (DFG)s. It uses this knowledge for optimising communication and hides its implementation details from the user. Skeleton-based design flows have been used in parallel programming for supercomputers [10], GPU [11], grid structures [12], and hybrid architectures [13]. Benkrid et al. extended this concept into a *hardware skeleton* with placement information and applied it to FPGA (Field Programmable Gate Arrays)-based designing [14, 15]. Hardware skeletons do not contain routing information since it can be generated by FPGA back-end tools. However, the routing information is an essential part of mapping algorithms on the memristor crossbar. We further extend the hardware skeleton concept with routing information and refer it as [2]*CIM skeleton*. This skeleton can be configured with different predesigned circuits for implementing corresponding algorithms. Furthermore, complex algorithms can be implemented by composing simple skeletons. The main contributions of this paper are:

- Extending the hardware skeleton concept by appending routing information. The extended skeleton integrate information about scheduling, placement, and routing for a class of algorithms.
- Developing a design and simulation flow for memristor-based CIM architecture based on the extended skeleton. By using a few carefully designed skeletons, many

Nanoarch '16, July 18-20, 2016, Beijing, China
© 2016 ACM. ISBN 978-1-4503-4330-5/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2950067.2950071

[1] Direction of source/sink input/output ports, i.e., North, South, East, West.

[2] In the rest of the paper, *skeletons* refer to CIM skeletons.

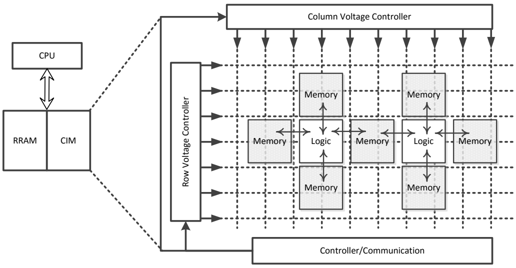


Figure 1: CIM/CPU Heterogeneous Computing and Memristor Crossbar Configuration

parallel algorithms can be implemented rapidly. Both a SystemC model and a layout are generated through the flow.

The rest of the paper is organized as follows. After introducing the CIM architecture in Section 2, Section 3 presents the design and simulation flow. Section 4 shows the experimental results for three study cases. Finally, Section 5 concludes the paper and discusses future research directions.

2. BACKGROUND

Figure 1 shows the heterogeneous CIM/CPU computing scenario. The CPU works with memristor-based memory, including Resistive Random Access Memory (RRAM) and CIM. Besides of storing data, CIM also performs computing as an accelerator of CPU.

2.1 CIM Architecture

CIM architecture [2] consists of a memristor layer and a CMOS layer. The former is a dense crossbar with a memristor at each cross point while the latter is used to implement the controller as shown by Figure 1. Any part of the crossbar can be configured as either memory or logic, depending on the commands of the controller. The communication between logic and memories within the crossbar can be in any direction, as shown in the figure. Due to the high density of memristor technology [16, 17], a CIM chip could contain as many as 10^6 functional units. As a result, manually exploring the design space is impossible. To transfer data between functional units, we have two options. One is through the CMOS layer, which has a bandwidth limit. The other one is on the memristor crossbar, which is named as the *copy* operation [7]. The latter one is preferred, because it has higher parallelism. The state of a memristor can be copied to another in one cycle if they share the column or the row. Otherwise, this operation will take a minimum of two cycles and temporary registers will be needed. In Figure 2, A and C represent source memristors on the output ports of two multipliers while B and D are destinations memristors on two input ports of an adder. Since A and B share the row, copying the data from A to B needs only one cycle. The pseudo command of the controller is:

$S_1:\ Move\ (25,25)\ to\ (35,25)\ \{A \rightarrow B\}.$

The controller addresses the memristors with their coordinates. Different from this case, C and D are not in the same column or row. Thus, we need to divide the communication into two steps:

$S_1:\ Move\ (25,10)\ to\ (40,10)\ \{C \rightarrow E\}$

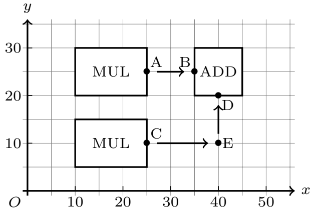
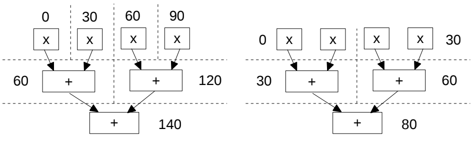


Figure 2: CIM Crossbar Communication



(a) One Multiplier (b) Two Multipliers

Figure 3: ASAP Scheduling of Inner Product

$S_2:\ Move\ (40,10)\ to\ (40,20)\ \{E \rightarrow D\}.$

2.2 Requirements for a New Flow

The data transfer mechanism on memristor crossbar has a significant influence on the design flow, especially for scheduling. Scheduling is a process that assigns time steps to operations. If a design cannot meet the constraint, the compiler will allocate more resources and try to schedule again. Figure 3 illustrates the scheduling results of the vector inner product function: $\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$. Here, $\vec{a} = (a_1, a_2, ..., a_n)$, $\vec{b} = (b_1, b_2, ..., b_n)^\intercal$, and the vector size $n = 4$. For explanation purposes, we set a latency constraint of 100 cycles for this function. We assume the multiplication's latency is 30 cycles and addition is 20. Now, if three adders and only one multiplier are allocated, the lowest latency of the function is 160 cycles. It can be achieved by using ASAP (As-Soon-As-Possible) scheduling algorithm. The start cycle of each operation is marked in Figure 3a. This scheduling cannot meet the constraint, so the compiler will allocate more resources and schedule again. Figure 3b shows the scheduling results with two multipliers and three adders. The overall latency is 100, which meets the constraint. From this case, we learn that scheduling validates resource allocation. Because the placement and routing are based on the allocated resources, they can only be performed after the scheduling process.

The scheduling process for memristor-based CIM also depends on routing results. Let A and B be two operations and B have a data dependency on A. For conventional CMOS technology, the data transfer from A to B is done once operation A finishes in no more than one cycle. Therefore, we can schedule B to the next time step after A without considering the routing. In CIM architecture, the data transfer needs one or more cycles, depending on the routing between them. As a result, the scheduler cannot decide the numbers of time steps between A and B without routing information. In conclusion, the scheduling, placement and routing depend on each other in CIM architecture. Hence, conventional design flows cannot be applied directly to CIM designing.

It is possible to adapt the conventional flow to CIM by iteratively executing it. However, even to get a suboptimal result, a long execution time is needed. The idea is to

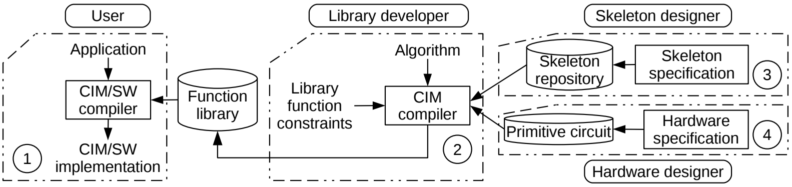


Figure 4: CIM Compilation Tool-chain

make assumptions on data transfers during the scheduling phase. Later, we convert these assumptions into constraints on placement and routing. If these cannot be met, then we need to increase the assumed costs and restart from scheduling. Placement and routing are time-consuming processes, so this iteration leads to a long time. Alternatively, we can make a trade-off between the quality of the solution and the execution time. In this case, the result is suboptimal.

Different from this approach, we solve the scheduling, placement and routing all together for a class of problems with a particular structure. We obtain the optimal solution without the need to iterate. This methodology is introduced next.

3. SKELETON-BASED DESIGN AND SIMULATION FLOW

Figure 4 shows the overview of the complete CIM/SW compilation tool-chain, which consists of four components. At the highest level (Box 1 in the figure), the user programs an application in a high-level language, such as C, with annotations of using CIM library functions. These functions are the most time-consuming algorithms in the application, which are designed by library developers (Box 2). Their work is based on the support of skeleton designers and hardware designers, who provide the specification of fundamental skeletons (Box 3) and primitive circuits (Box 4) respectively.

In this paper, we address only the flow for the library developer, i.e., the CIM compiler. It translates algorithms into CIM implementations, including configuration files for the memristor crossbar and the controller circuits. In current research phase, we generate SystemC models for simulation, together with files that indicate the function-level layout. Section 3.1 introduces the compiler's working basis, i.e., primitive circuits and skeletons. Section 3.2 uses three examples to show the system generation process. Section 3.3 presents the support for parallel simulation.

3.1 Primitive Circuits and Skeletons

Primitive circuits form the basic structures with which we build higher-level functional blocks. They are crossbar memristor implementations of widely used operators, such as Boolean logic gates [18], adders [19], multipliers. We use SystemC models to represent them. Along with these models, the hardware designer also needs to provide the attributes of the primitive circuits, which are latency, energy, and area (i.e., width and height within the crossbar). They are used by the CIM compiler to calculate the attributes of generated functions. When a circuit is idle, it consumes no energy, due to the zero-leakage property of memristors [20]. We regard the boundary of a circuit as a rectangle. Its width and height are given by the number of memristors used in each side. We assume CIM works at a fixed frequency, and the attributes are evaluated at this frequency.

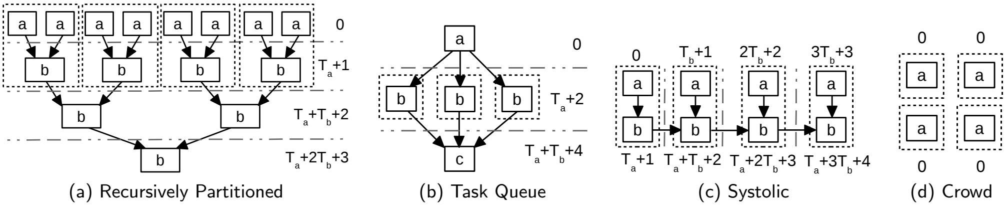
In this work, a skeleton consists of several nodes. It specifies the parallelism, communication, and synchronization of these nodes, without defining their functionality. These nodes can be configured as either primitive circuits or skeletons. When a node is configured as a primitive circuit, its functionality is decided. Configuring a node as a skeleton is called *skeleton nesting*. By using nesting repetitively, the function designer can build large and complex skeletons.

In Figure 4, the skeletons stored in the repository are provided by skeleton designers. They are called *fundamental skeletons*. The skeleton designer first needs to decide the set of fundamental skeletons, according to their expressiveness, reuse-ability, and designing difficulty. Then, he defines the scheduling, placement, and routing algorithms for each skeleton. When a skeleton is used to create library functions, the library designer does not need to care about its implementation details. We choose the fundamental skeleton set following the classification proposed by Campbell [21] and for which the DFGs are shown in Figure 5; the nodes with the same letters are configured with the same primitive circuit or skeleton. These fundamental skeletons are:

- **Recursively partitioned**. Problems are partitioned into a small size, and they are solved separately. After that, the solutions are collected in a recursive style.
- **Task queue**. These problems are solved by repeated concurrent execution of many instances of a task.
- **Systolic**. It consists of nodes that have data flowing between them and that may operate concurrently in a pipelined fashion.
- **Crowd**. Similar to the Systolic skeleton except for that there is no data flow between the concurrently operating nodes. A one-dimensional *Crowd* skeleton is an array of nodes while a two-dimensional one is a matrix.

To represent the layout, every skeleton is extended with a coordinate system. The placement of its nodes is performed under this system. A circuit rectangle may have eight different orientations, which is sufficient to be represented by an angle (0, 90, 180, and 270), and whether it reflects over the x-axis [22]. We use the coordinate of the bottom left corner as the position of the node. When skeletons are nested, their layout coordinate systems are also nested. Ports are also regarded as rectangles. Their placement is described under the coordinate system of a primitive circuit or a skeleton.

A skeleton is associated with a placement and routing algorithm. In the *Recursively partitioned* skeleton, nodes are arranged following the H-tree [23] pattern to minimize the communication cost. In this pattern, an output port is linked with a direct path to an input port if there is a data flow in between. Since all the corresponding bits share same rows or columns, their communication costs are just one cycle. Figure 6 shows a layout of a three-level *Recursively*



(a) Recursively Partitioned (b) Task Queue (c) Systolic (d) Crowd

Figure 5: DFGs, Scheduling, and Parallel Simulation Support of Fundamental Skeletons

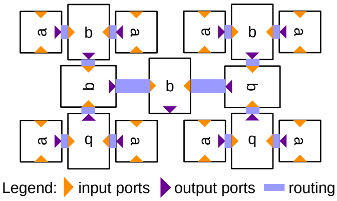


Figure 6: Three-level H-tree Layout

partitioned skeleton as an example. In other skeletons, the nodes are placed next to each other in a matrix style.

Since a skeleton also defines the placement and routing algorithms, it knows the communication cost between operations during the scheduling phase. For instance, all the communication costs are only one cycle in *Recursively partitioned* skeletons while some of them are two cycles in *Task queue* skeletons. In Figure 5, the starting moments of the nodes' execution are marked besides the skeletons. T_x represents the latency of node x in these expressions.

3.2 System Generation

In functional programming languages, skeletons are higher-order functions. They take functions as parameters. Via these parameters, a skeleton's nodes are configured as primitive circuits or skeletons. If a skeleton is scalable, it also has parameters for configuring the size.

Suppose we want to implement the matrix multiplication algorithm:

$$AB = (\vec{a_1}^{\mathsf{T}} \quad \vec{a_2}^{\mathsf{T}} \quad \cdots \quad \vec{a_n}^{\mathsf{T}})^{\mathsf{T}} (\vec{b_1} \quad \vec{b_2} \quad \cdots \quad \vec{b_n})$$

$$= \begin{pmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} & \cdots & \vec{a_1} \cdot \vec{b_n} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} & \cdots & \vec{a_2} \cdot \vec{b_n} \\ \vdots & \vdots & \ddots & \vdots \\ \vec{a_n} \cdot \vec{b_1} & \vec{a_n} \cdot \vec{b_2} & \cdots & \vec{a_n} \cdot \vec{b_n} \end{pmatrix},$$

where $\vec{a_i}$ is a row vector of matrix A and $\vec{b_i}$ is a column vector of B. This is a complex algorithm that does not fit any fundamental skeleton. However, we can see that it contains repetitive patterns. Each element of the result matrix is an inner product of two vectors. Thus, we can divide it into two levels. The top level is a two-dimensional *Crowd* skeleton, because there are no data flows between these elements. The lower level is the vector inner product function. This function suits a *Recursively partitioned* skeleton, with "a" and "b" nodes in Figure 5 configured as multipliers and adders. They are predefined operators that can be found in the primitive circuit library.

To implement the matrix multiplication, we need to build the system bottom-up. First, we declare instances of the adder and the multiplier. Subsequently, we instantiate a *Recursively partitioned* skeleton based on these primitive circuit library elements. Constraints can be applied to the skeleton if necessary. Finally, we build a two-dimensional *Crowd* skeleton on top of the inner product and generate SystemC codes. We assume both matrices are 16 × 16, so the vector size of the inner product is also 16. Figure 7 represents the generated system. The symbols "×" and "+" stand for multipliers and adders while dashes between them are communication paths. Each subsystem, as shown in the dashed box, has a detailed layout following the H-tree pattern. If an application cannot fit any existing skeleton, it is necessary to develop a new one. In this case, the skeleton repository should be extended.

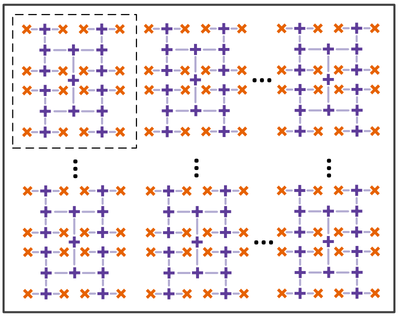


Figure 7: Multiplication of Two 16×16 Matrices

In a similar way, we can calculate all the results of discrete convolution on range $[-M,\ M]$:

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m] \cdot g[m],\ n \in [-M,\ M],$$

where f and g are two functions, and n is a variable. we can use the *Systolic* skeleton as the low level, and instantiate a one-dimensional *Crowd* skeleton on top of that. An instance of the *Systolic* skeleton produces the result of one input value, so all the results can be acquired with multiple instances simultaneously.

3.3 Parallel Simulation Support

The standard SystemC implementation [24] does not support parallelism. It limits the performance and scale of the simulation since the resources on a single machine are finite. Therefore, we add parallel simulation support in code generation for acceleration and for enlarging the system scale.

Different skeletons require different support strategies. Figure 5 illustrates one possible parallelization method for each skeleton. The parts marked with dotted boxes can be

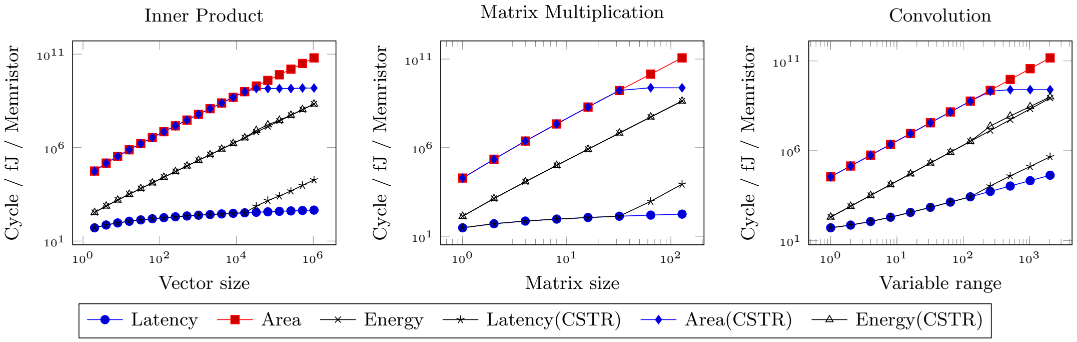


Figure 8: Experimental Results

simulated in parallel by different threads (possibly on different machines). The sizes of these boxes are decided by the number of available threads. The communication between these threads is implemented with MPI (Message Passing Interface).

4. EXPERIMENTAL RESULTS

To validate the design flow, we first demonstrate the approach for vector inner product. Thereafter, we analyze its scalability while considering not only the inner product but also matrix multiplication and convolution. Finally, the ability of the proposed approach to perform large simulation will be shown. It is worth noting that these experiments are performed on a high-performance computer with 20 Intel Xeon E5-2670 HT cores, running at 2.50 GHz.

4.1 Placement and Routing Results

We use the graphic output of inner product to show the placement and routing results. We configured the *Recursively partitioned* skeleton with multipliers and a subsystem, which is the combination of an adder and two registers. Registers are needed to change the orientations of the input ports so that the adders and the multipliers can be arranged in a H-tree style. The attributes of these primitive circuits are listed in Table 1 [2, 18, 19]; they are synthetic data used only for illustration purpose. Here, the latency is the number of clock cycles (CC) between the inputs and the corresponding output. The width and height are expressed in the number of memristors. The energy is valued for producing one (set of) result(s) in terms of femtojoule (fJ). Figure 9 shows the graphic output generated by our flow when the vector size is 16. Adders and multipliers are marked with “A” and “M” while registers are squares without labels. The input ports (orange triangles) are aligned with the output ports (violet triangles), and the circuit is mapped according to the H-tree pattern. The graphical output allows us to verify that the placement algorithm defined by the skeleton works correctly.

4.2 System Scaling

We varied the system sizes to evaluate the scaling capabilities without putting any constraint, and generated three cases: inner product, matrix multiplication, and convolution; we assume the matrices to be square $N \times N$. The results are shown in Figure 8 for Latency, Area, and Energy.

Table 1: Primitive Circuit Attributes

Module	Latency/CC	Width	Height	Energy/fJ
Adder	20	80	100	67
Multiplier	30	120	160	134
Register	1	33	33	1

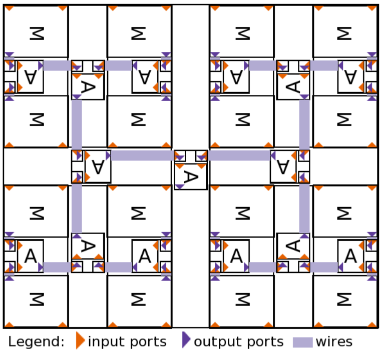


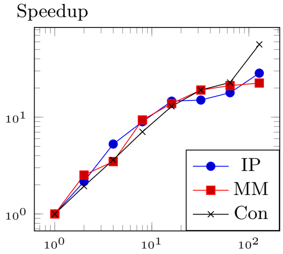
Figure 9: Graphic Output of Inner Product

The area is defined based on the total number of memristors used by the implementation. The time complexities of the inner product and matrix multiplication are $O(\log N)$ while that of the convolution is $O(N)$. They are confirmed by the experimental results.

Next, we put area constraints to investigate the flow’s ability to handle them; the constraint specifies that the width and height should not exceed 50,000 memristors each. We have to point out that this is only a hypothetical setting. In reality, a CIM chip could be much bigger than this. These experiment results are also shown in Figure 8; they are marked with CSTR in the legend. Comparing these with those for which no constraints was assumed, we can see that the trends of latency and area change. The area stops growing, which shows that the constraint is applied. However, the latency grows in a polynomial manner due to hardware reuse.

4.3 Parallel Simulation

We enabled the parallel simulation support to examine its effect. First, we simulated the baselines which are based on



Alg	Size	Base/s
IP	2^{18}	590.1
MM	64	275.4
Con	256	718.4

Figure 10: Speedup of Parallel Simulation

sequential simulations. The results are shown on the right side of Figure 10; it lists the simulation sizes and the corresponding simulation time. The abbreviations IP, MM, and Con stand for Inner Product, Matrix Multiplication, and Convolution respectively. Thereafter, we fixed the system size and changed the number of MPI nodes. For each configuration, we performed the simulation ten times and calculated the average execution time after removing the maximum and minimum values. Figure 10 shows the speedup of each configuration over the sequential simulation as the baseline. The output data of all the parallel simulations are verified and found to match those of the sequential one. When MPI nodes are less than 16, the speedups are almost the same as the thread number. This result shows a good scalability.

5. CONCLUSION AND FUTURE WORK

In this work, we explained why a skeleton-based flow is required for CIM, and we presented a high-level description of this flow with a focus on the collaboration between different designers. We extend hardware skeletons with routing information. An extended skeleton provides scheduling, placement, and routing algorithms for a class of problems that have similar structures. With composition operations, complex skeletons can be built from simple ones.

In future work, we will address each of the specific toolchain compilers in detail.

6. REFERENCES

[1] Y. Pershin and M. Ventra, “Memcomputing: A computing paradigm to store and process information on the same physical platform,” in *IWCE*, 2014, pp. 1–2.

[2] S. Hamdioui, L. Xie, H. A. D. Nguyen *et al.*, “Memristor based computation-in-memory architecture for data-intensive applications,” in *DATE*. EDA Consortium, 2015, pp. 1718–1725.

[3] L. O. Chua, “Memristor-the missing circuit element,” *Circuit Theory, IEEE Transactions on*, vol. 18, no. 5, pp. 507–519, 1971.

[4] D. B. Strukov, G. S. Snider, D. R. Stewart *et al.*, “The missing memristor found,” *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[5] H. A. D. Nguyen, L. Xie, M. Taouil *et al.*, “Computation-in-memory based parallel adder,” in *NANOARCH*, July 2015, pp. 57–62.

[6] C. L. Seitz, “System timing,” *Introduction to VLSI systems*, pp. 218–262, 1980.

[7] L. Xie, H. A. D. Nguyen, M. Taouil *et al.*, “Interconnect networks for memristor crossbar,” in *NANOARCH*, July 2015, pp. 124–129.

[8] S. H. Gerez, *Algorithms for VLSI design automation*. Wiley New York, 1999, vol. 8.

[9] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[10] M. Zandifar, M. Abdul Jabbar, A. Majidi *et al.*, “Composing algorithmic skeletons to express high-performance scientific applications,” in ser. ICS ’15. New York, NY, USA: ACM, 2015, pp. 415–424.

[11] C. Nugteren and H. Corporaal, “Bones: An automatic skeleton-based c-to-cuda compiler for gpus,” *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 35:1–35:25, Dec. 2014.

[12] Y. Wang and Z. Li, “Gridfor: A domain specific language for parallel grid-based applications,” *International Journal of Parallel Programming*, pp. 1–22, 2015.

[13] M. Goli and H. Gonzalez-Velez, “Heterogeneous algorithmic skeletons for fast flow with seamless coordination over hybrid architectures,” in *PDP*, Feb 2013, pp. 148–156.

[14] K. Benkrid, D. Crookes, J. Smith, and A. Benkrid, “High level programming for fpga based image and video processing using hardware skeletons,” in *FCCM ’01*, March 2001, pp. 219–226.

[15] K. Benkrid and D. Crookes, “From application descriptions to hardware in seconds: a logic-based approach to bridging the gap,” *VLSI*, vol. 12, no. 4, pp. 420–436, April 2004.

[16] K. Eshraghian, K. R. Cho, O. Kavehei *et al.*, “Memristor mos content addressable memory (mcam): Hybrid architecture for future high performance search engines,” *VLSI*, vol. 19, no. 8, pp. 1407–1417, Aug 2011.

[17] E. Linn, R. Rosezin, S. Tappertzhofen *et al.*, “Beyond von neumann - logic operations in passive crossbar arrays alongside memory operations,” *Nanotechnology*, vol. 23, no. 30, p. 305205, 2012.

[18] L. Xie, H. A. D. Nguyen, M. Taouil *et al.*, “Fast boolean logic mapped on memristor crossbar,” in *ICCD*, Oct 2015, pp. 335–342.

[19] S. Kvatinsky, G. Satat, N. Wald *et al.*, “Memristor-based material implication (imply) logic: Design principles and methodologies,” *VLSI*, vol. 22, no. 10, pp. 2054–2066, Oct 2014.

[20] H. Lee, Y. Chen, P. Chen *et al.*, “Low-power and nanosecond switching in robust hafnium oxide resistive memory with a thin ti cap,” *EDL*, vol. 31, no. 1, pp. 44–46, Jan 2010.

[21] D. K. Campbell, “Clumps: a candidate model of efficient, general purpose parallel computation,” Ph.D. dissertation, University of Exeter, 1994.

[22] M. A. Riepe and K. A. Sakallah, “Transistor level micro-placement and routing for two-dimensional digital VLSI cell synthesis,” in ser. ISPD ’99, New York, NY, USA: ACM, 1999, pp. 74–81.

[23] A. L. Fisher and H. Kung, “Synchronizing large systolic arrays,” in *1982 Technical Symposium East*, 1982, pp. 44–52.

[24] Accellera Systems Initiative, SystemC. [Online]. Available: http://accellera.org/

Skeleton-based Synthesis Flow for Computation-In-Memory Architectures

Jintao Yu, *Student Member, IEEE,* Răzvan Nane, *Member, IEEE,* Imran Ashraf, *Member, IEEE,* Mottaqiallah Taouil, *Member, IEEE,* Said Hamdioui, *Senior Member, IEEE,* Henk Corporaal, *Member, IEEE,* and Koen Bertels, *Member, IEEE*

***Abstract*—Memristor-based Computation-in-Memory (CIM) is one of the emerging architectures for next-generation Big Data problems. Its design requires a radically new synthesis flow as the memristor is a passive device that uses resistances to encode its logic values. This article proposes a synthesis flow for mapping parallel applications on memristor-based CIM architecture. First, it employs solution templates that contain scheduling, placement, and routing information to map multiple algorithms with similar data flow graphs to the memristor crossbar; this template is named *skeleton*. Complex algorithms that do not fit a single skeleton can be solved by nested skeletons. Therefore, this approach can be applied to a wide range of applications while using a limited number of skeletons only. Secondly, it further improves the design when spatial and temporal patterns exist in input data. To accelerate simulation of generated SystemC models, we integrate MPI in skeletons. The synthesis flow and its additional features are verified with multiple applications, and the results are compared against a multicore platform. These experiments demonstrate the feasibility and the potential of this approach.**

***Index Terms*—Memristor, algorithmic skeleton, SystemC.**

I. Introduction

Big-Data analytics is becoming increasingly difficult to solve using CMOS-based Von Neumann computer architecture [1]. The reasons include, but are not limited to, the access bottleneck between the processor and memory, energy inefficiency [2], and the limited scalability of CMOS technology [3]. Memristor-based [4], [5] Computation-in-Memory (CIM) architectures [6]–[9] address the aforementioned problems by enabling in-memory processing using emerging non-volatile technologies. Manually designed case studies revealed their enormous potential by outperforming the state-of-the-art with orders of magnitude [10]–[13]. Exploring the potential of such architectures and appropriately evaluating their performance and scalability for larger applications require automatic flows and methods that efficiently map high-level algorithmic description to low-level memristor crossbar configuration.

Existing Computer-Aided Design (CAD) flows for CMOS-based VLSI (Very-Large-Scale Integration) are not applicable to memristor-based CIM because of different signal propagation styles. In CMOS circuits, logic values are represented by their voltage. The voltage change of a source automatically propagates to the sink along a dedicated wire within one clock cycle [14]. However, in a memristor crossbar, logic values can only propagate to other positions with the help of controllers, because they are encoded by the memristors' resistance. The controller transfers the data in one or multiple steps, each of which is conducted along a vertical or horizontal nanowire shared by many memristors. Therefore, the number of steps equals to the number of turnings in the path between the source and the sink [15]. In addition to computation, memristor-based CIM needs extra clock cycles for communication, and the communication latency is determined by the routing result. In conventional VLSI CAD flows, placement and routing are performed based on the High-Level Synthesis (HLS) scheduling results [16]. However, it is not applicable for memristor-based CIM since the routing result is required to schedule communications. As a consequence, a new methodology is needed to eliminate the cyclic dependency among scheduling, placement, and routing.

In this work, we propose a synthesis flow that simultaneously performs scheduling, placement, and routing. This is inspired by the *skeleton* concept used in parallel computing domain [17]–[21]. A skeleton is a scheduling template for a specific class of algorithms that share a similar Data Flow Graph (DFG) in the sense of data dependency. A scheduling template handles parallelism, synchronization, and communication among threads, regardless of their functionality. It can be optimised according to the characteristic of DFG structures, thus achieving better performance than generic scheduling algorithms. FPGA developers extended this concept into a *hardware skeleton* with placement information [22], [23]. Routing is not included in hardware skeletons since it is generated by FPGA back-end tools. Nevertheless, the routing information is essential for mapping algorithms to memristor crossbar. Hence, we further extend the hardware skeleton concept with routing information and refer it as [1]*CIM skeleton* [24]. The skeletons can be configured for different predesigned circuits and implement their corresponding algorithms. Furthermore, complex algorithms can be implemented via skeleton nesting. This article is built on our preliminary work, where the main focus was laid on the general idea of applying skeletons to CIM architecture design. Compared to the preliminary work,

J. Yu, R. Nane, I. Ashraf, M.Taouil, S. Hamdioui, and K. Bertels are with the Department of Quantum Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: j.yu-1, r.nane, i.ashraf, m.taouil, s.hamdioui, k.l.m.bertels@tudelft.nl).

H. Corporaal is with Department of Electrical Engineering, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands (e-mail: h.corporaal@tue.nl).

Manuscript received 29 December 2016; revised 31 May 2017; accepted 30 July 2017. Date of publication 20 October 2017.

[1] *Skeletons* refer to CIM skeletons in the rest of the paper.

we have made the following contributions:

- We developed memristor-based computational units including a 32-bit adder and a 16-bit multiplier. The adder outperforms state-of-the-art memristor adder designs in terms of delay. Currently, no binary multipliers have been proposed for the memristor crossbar.
- We specified a methodology that allows us to integrate multiple computational units in the crossbar while maintaining parallel computing.
- We considered data input and output processes and identified possible patterns in the input/output data.
- We provided new test cases. All the skeletons are covered by the updated test cases.

The rest of the paper is organized as follows. First, Section II presents our memristor-based designs and systems. Subsequently, Section III presents the skeleton-based synthesis flow. Section IV explains implementation details of its key parts. Experimental results of three case studies are shown in Section V. Finally, Section VI concludes the paper and discusses future research directions.

II. Hardware Platform

Section II-A presents the primitive circuits; they are a 32-bit adder and a 16-bit multiplier. Subsequently, Section II-B presents the hardware organization at the system level.

A. Primitive Circuits in CIM

In memristor-based CIM architectures, a *primitive circuit* or a *circuit* in short, is a memristor circuit that performs a computational operation in the crossbar, such as addition and multiplication. These circuits can be implemented in various manners. Design styles where the applied voltages are data-independent may use shared controllers. Examples are material implication logic [25], Boolean logic [26], [27], majority logic [28], and MAGIC (Memristor-Aided loGIC) [29]. Other designs cannot have a shared controller, such as CRS [30] as their control signals are data-dependent. In this work, we design the primitive circuits based on MAGIC due to its simplicity. In principle, any of the above logic schemes that support a shared controller can be used.

CIM regards memristors as digital devices that have two stable states. In MAGIC, logic '1' is represented by low resistance (ON) and '0' by high resistance (OFF) [31]. The operations that switch a memristor to the ON/OFF states are respectively called SET/RESET. They can be achieved by applying positive or negative voltages that are larger than the SET/RESET threshold voltages of the memristor [29].

In MAGIC, memristors are placed on a 2-dimensional grid, where each memristor is connected to a horizontal and a vertical nanowire [29] (see Fig. 1). Appropriate voltages are applied to nanowires by the CMOS controlled voltage drivers. The voltage drivers consist of a set of voltage sources and switches as shown in Fig. 2; the three switches select the required supply voltage. MAGIC uses up to nine voltage levels, including the three levels shown in the figure. Here, V_0 is the execution voltage, which is used together with ground (GND) to execute a logic operation. V_{VS} is the isolation voltage and is applied to the columns where the memristor states must keep their values. Examples of the other voltages are SET, RESET and READ voltages. The number of required voltage levels is much more than for typical RRAMs, which typically require four voltage levels [32].

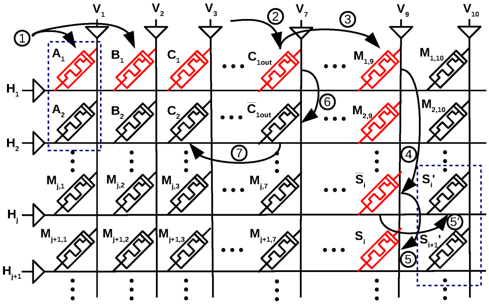


Fig. 1. A 32-bit adder implemented with MAGIC logic.

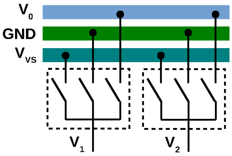


Fig. 2. Voltage controller implementation in CIM.

MAGIC supports only one logic operation in the crossbar, an n-input NOR operation [29]. The input and output values of the NOR operation are stored in memristors that share the same row or column. Note that when $n = 1$, the NOR operation reduces to a NOT operation. As NOR is functionally complete, we can use it to build various circuits. Talati *et al.* presented a 1-bit full adder [29] using the red memristors shown in Fig. 1. First, the inputs are copied to the adder (step ①). Then, several NOR operations are conducted in the first row and the carry bit is obtained at the seventh column step ②). Subsequently, step ③ and step ④ are used as intermediate computation steps. Finally, the sum bit S is obtained from its complement $\overline{S}$ (step ⑤). We designed an n-bit adder based on this adder. The n-bit adder is also shown in Fig. 1. Two main changes have been made. The first one is that we inserted two steps (⑥ and ⑦) between step ② and ③ to move the carry output bit to the carry input bit of the next 1-bit adder. The second change is to conduct step ⑤ horizontally (as shown by ⑤’) instead of vertically, to allow parallel operations. We have listed the detailed control steps in the supplementary material of this article. Talati *et al.* also presented an n-bit adder in which multiple 1-bit adders are linked together, with an overall latency of $10n + 3$ cycles [29]. We have improved it to $8n + 8$ by leveraging the data parallelism as indicated by the dashed box in Fig. 1. Our design is also faster than the MAGIC-based design provided in [33].

We have also implemented a 16-bit multiplier, inspired by

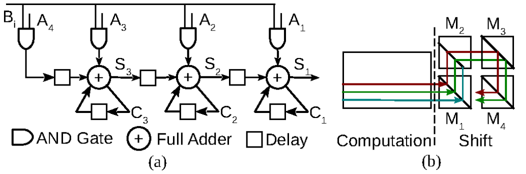


Fig. 3. CIM multiplier design: (a) diagram of a 4-bit CSAS multiplier; (b) implementation overview in CIM architecture.

TABLE I
ATTRIBUTES OF PRIMITIVE CIRCUITS USED IN THE ARTICLE

Circuits	Latency (CC)	Width	Height	Energy (pJ)
Adder	264	10	96	265.19
Multiplier	499	81	64	3084.53
3-input XOR	32	12	32	169.28

the Carry-Save Add-Shift (CSAS) algorithm [34]. Fig. 3(a) shows a 4-bit CSAS multiplier, which contains four AND gates and three 1-bit full adders. For more details regarding this algorithm, we refer the reader to Gnanasekaran's article [34]. CSAS algorithm suits a memristor implementation due to the parallel AND and add operations. However, the shift operation (required to shift the sum S_i) might be a bottleneck. When no dedicated hardware is provided for this shift operation, its time complexity equals $O(N)$, with N representing the amount of bits to shift [35]. We try to accelerate the shift operation in CIM using additional hardware as shown by Fig. 3(b). The left part (donated by *Computation*) contains all AND gates, 1-bit full adders, and intermediate results. The right part (donated by *Shift*) contains four *mirrors* (M_1 to M_4); a mirror is a small square crossbar that only contains memristors on diagonal positions used to link horizontal and vertical nanowires [15]. Mirror M_2 is a special mirror in which its memristors are located on a line parallel to the diagonal. Therefore, this mirror shifts a signal by one position as shown by the red and green lines. More details, including control steps, can be found in the supplementary material.

The latency, area, and energy consumption of the primitive circuits are listed in Table I. The latency, expressed in clock cycles, is directly obtained from the number of cycles the controller needs (see also the supplementary material). The area, expressed in number of required memristors, is determined from the number of rows (Height) and columns (Width). The energy, expressed in picojoules, is calculated from the number of operations per cycle and the data width. We assume one memristor to be written for each bit during each operation. Note that the energy consumption is ideally input-depended. The cost to write a memristor (SET/RESET) lies in the range of 0.1 fJ [36] to 230 fJ [37]. We assume the worst case of 230 fJ. The static power consumption is ignored here and will be part of the future work.

B. Circuits Organization in CIM

Fig. 5 shows one of the possible CIM's working scenarios [9], [24]. It consists of a CIM accelerater and a Resistive Random Access Memory (RRAM) used as the main memory. It exchanges data with CPU and storage in the same way as conventional technologies. Since the memristor crossbar is not continuous, we need to add additional nanowires to transfer main inputs and outputs to/from the internal circuits. RRAM and CIM both have memristor and CMOS layers. Their controllers are both implemented in CMOS. CIM is connected to the RRAM via nanowires in a dedicated layer as shown in the part of Fig. 5(b). Each nanowire creates a connection between RRAM and one or more primitive circuits as shown in the front view. A nanowire transfers at most a single data bit during each clock cycle.

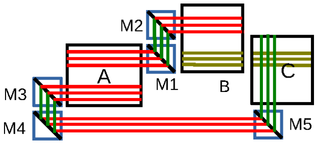


Fig. 4. The linkage of three primitive circuits.

CIM's memristor layer consists of primitive circuits that satisfy two conditions. First, these circuits must operate in parallel to achieve better performance, and second, they should not conflict with each other during operation. Therefore, we avoid placing input/output ports on the same rows or columns and link them by using two or more mirrors. Fig. 4 shows for example three primitive circuits A, B, and C; A exchanges data with B and C. The horizontal and vertical lines related to communication are colored red and green respectively. In CIM, we assume that all data words are 32-bit wide, and that all the bits are transferred in parallel. When A transfers a word to B, it is first copied to $M1$, then to $M2$, and finally to B. The nanowires between $M1$ and B are disconnected. This created isolated islands of primitive circuits except where I/O takes place. The isolated nanowires in B are colored brown. Breaking the nanowires allows A and B to operate independently since they do not share nanowires or memristors. Communication is conducted when A and B are not operating. Similarly, the nanowires between B and C are also broken. For brevity, nanowires not related to previous discussion are not shown. The negative affect of this solution is that it decreases the density of the crossbar and, more importantly, increases the manufacturing complexity.

The latency of the communication described above equals to the number of mirrors plus one. Therefore, transferring data from A to C needs one more cycles for A to transfer data to C than from A to B in Fig. 4. This feature has a significant influence on the design automation, which will be analyzed in Section III-A.

We have synthesized the CMOS controller for the 32-bit adder with Cadence's RTL Compiler and NanGate's 15 nm library [38]; the reported area is 30 μm^2. However, International Technology Roadmap for Semiconductors 2.0 (ITRS 2.0) predicts that the density of memristor crossbar will reach 2.38×10^{11} $\mathrm{bit/cm^2}$ in 2020 [39]. With this density, a crossbar of 96 rows and 10 columns (i.e. the size of the adder) is only 0.23 μm^2. This means that the CMOS controller is 130x larger than the memristor crossbar. If we assign a

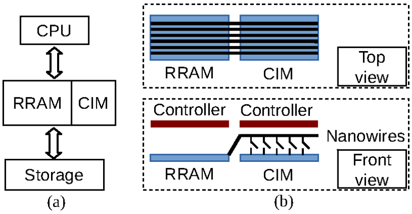


Fig. 5. CIM/CPU heterogeneous computing: (a) Overall structure; (b) Communication between RRAM and CIM.

dedicated a controller to every primitive circuit, then CMOS controller's area dominates the chip area and we cannot exploit the high density of the crossbar. Therefore, the controller must be shared between the logic circuits. Furthermore, this is possible due to the nature of memristor logic as discussed in Section II-A. Sharing the same controller requires the circuits to operate synchronously. In Section V, we will show that the controller can handle 10^4 to 10^5 circuits simultaneously. As a result, the crossbar area becomes dominant. Due to the small area of the controller, it will be ignored in the rest of this paper.

RTL Compiler reports the power of the 32-bit adder to be $13.59\,\mu\mathrm{W}$ with a frequency of $1\,\mathrm{GHz}$. It leads to an energy consumption of $3.588\,\mathrm{pJ}$ for a time period of $264\,\mathrm{ns}$ as the latency of the adder is 264 cycles. It is less than 2% compared to the energy consumption in the memristor crossbar. When the controller is shared with many primitive circuits, the percentage will be even smaller. Therefore, we will also omit the energy consumption of CMOS layer in the rest of the article.

III. Skeleton-based Synthesis Flow

Section III-A motivates the reason why a radically new design flow is required. Subsequently, we introduce the skeleton-based synthesis flow in Section III-B, III-C, and III-D.

A. Requirement for a New Flow

The communication characteristics of the memristor crossbar make the scheduling routing dependent, as the communication latency is determined by the routing. Fig. 6(a) shows the HLS flow used for CMOS circuit design. It consists of sequential processes; they are resource allocation, scheduling, placement, and routing. Only when a process fails meeting the performance or resource constraints, it goes back to the previous process. It is worth noting that scheduling is conducted before routing; hence, this flow is not applicable to CIM.

We can try to adapt the regular HLS flow to CIM, but these variants all lead to unsatisfactory situations. Since routing information is not available at scheduling phase, we can assume all communication has a maximum latency, like six or eight cycles. Based on this assumption, the operators can be scheduled. Then, after the routing phase, the communication latency is updated. Scheduling is conducted again to get a more accurate design. This adapted flow is shown in Fig. 6(b).

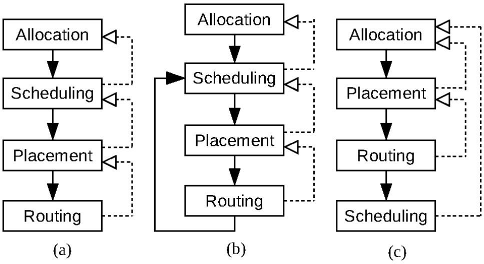


Fig. 6. Regular HLS flow and its variants for CIM: (a) HLS flow for CMOS; (b) Flow with communication latency assumption; (c) Flow with an adjusted sequence.

Placement and routing are time-consuming processes, so these iterations are extremely time consuming. If we make a trade-off between the quality of the solution and the execution time, then the performance of the generated design will be only suboptimal.

Fig. 6(c) shows an alternative variant, i.e. conducting placement and routing before scheduling. In this scenario, the scheduler has accurate information on communication latency. The drawback of this flow is that we need to go through all the processes before knowing whether the latency constraint is met. Similarly to Fig. 6(b), the long execution time of placement and routing will impair either the productivity or the quality of the design.

The fundamental problem of these adaptions is that they cannot eliminate the cyclic dependence among scheduling, placement, and routing. In a regular HLS flow, placement and routing should be conducted based on the scheduling result. However, in CIM architecture, the communication mechanism makes scheduling depend on the routing result. Therefore, a radically new approach is required. Different from these adapted flows, we solve the scheduling, placement, and routing altogether using CIM skeletons. The optimal solution is guaranteed without the need of iteration. This methodology is introduced in the next section.

B. Hardware/Software Partitioning

Fig. 7 shows the overview of the complete CIM synthesis flow, which consists of four components. At the application level (Box 1), the user partitions the original program into software and hardware, taking the hint given by the profiling tool. The hardware part needs to be rewritten to fit predefined skeletons. A skeleton contains scheduling, placement, and routing algorithms for a specific type of DFG structure (Box 2). The compilation at the kernel level (Box 3) is to instantiate skeletons with *Primitive Circuits*, which are predefined function units like adders and multipliers. The design of the circuit level (Box 4) has been presented in Section II-A In the flowing subsections, we will elaborate each of the rest boxes.

Before the compilation at the kernel level, we need to identify the favorable algorithms for hardware implementation. The best candidates should meet the following criteria:

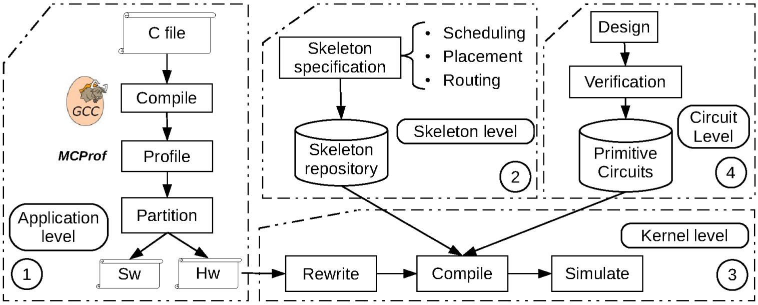


Fig. 7. Synthesis flow for memristor-based CIM architecture.

```
#define InputSize 8192
#define TAPS 64
int pre[TAPS], coe[TAPS], out;
void FIR(int in){
  int temp = 0;
  for(int j=TAPS-1; j>0; j-=1)
    pre[j] = pre[j-1];
  pre[0] = in;
  for(int j=0; j<TAPS; j++)
    temp+=pre[TAPS-j-1]*coe[j];
    out = temp;
}
void main(){
  int i, total = 0;
  for(i=1; i<=InputSize; i++){
    FIR(i); total += out;
  }
  printf("total:%d\n", total);
}
```

Fig. 8. FIR filter source codes.

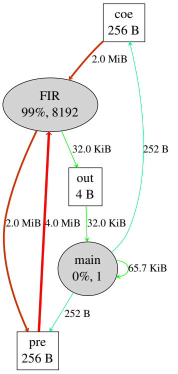


Fig. 9. MCProf profiling result.

- They form a large percentage of the execution time. According to Amdahl's law [40], accelerating such kernels can generate a recognizable overall speedup.
- They are coarse-grained, which means they do not change a large quantity of data with other parts of the application.
- They have inherent massive parallelism so that they have the potential to be accelerated.
- Their structures are easy to be implemented by hardware.

In order to highlight the computing and memory intensity parts of an application and to obtain the communication among these parts, we utilize *MCProf* [41], [42]. MCProf is a runtime memory and communication profiler based on Intel *Pin* dynamic binary instrumentation framework [43]. MCProf takes the binary of an application as input to generate profiling results in various formats. Based on the information generated by MCProf, developers can partition the application into software and hardware parts, as shown in Box 1 in Fig. 7. Later, the hardware part enters the kernel-level design flow which will be explained in Section III-C.

To illustrate the utilization of MCProf to extract the required information from an application, let us consider the C program of a Finite Impulse Response (FIR) filter modified based on LegUp's [44] testbench as shown in Fig. 8. The initialization of the coefficient array *coe* is omitted for concision. MCProf generates the output shown in Fig. 9. The functions are represented by ovals, which contains the name of the function, its percentage execution contribution, and the total number of calls; e.g. *FIR* function consumes 99% of the overall execution time and is called 8192 times. The rectangles represent objects, such as the *pre* and *coe* arrays in this case. The arrows represent the communication with the data amounts marked near the lines. Dense communication is indicated by red color (bold lines), and the rest is green. Clearly, the *FIR* function consumes most of the execution time, and most of the communication is between it and arrays *pre* and *coe*. If we implement the *FIR* function in main memory using the CIM concept, then the data transfer between the processor and the memory will be several orders of magnitudes smaller than the original version. In this example, the profiling is performed at the function level. By using markers, it is also possible to obtain profiling information at lower granularity levels, such as the loop level.

C. CIM Skeletons

In this skeleton-based synthesis flow, targeting problems are mapped to the crossbar by instantiating predefined solution templates with primitive circuits. Each skeleton can map a specific class of problems that share a similar DFG structure. In this paper, we generally follow the classification defined by Campbell [45] and define four structures as shown in Fig. 10. We chose this classification because it contains a relatively small number of classes while covering a broad range of problems. Each box in Fig. 10 represents an operation or a DFG consisting of multiple operations, and boxes with the same labels represent the same operation(s). The arrows between them indicate data dependency. The four structures are:

- **Recursively partitioned**. Problems are partitioned into a small size, and they are solved separately. After that, the solutions are collected in a recursive style.
- **Farm**. The same function is applied potentially in parallel to a list of independent jobs. The results are combined by a controlling process.

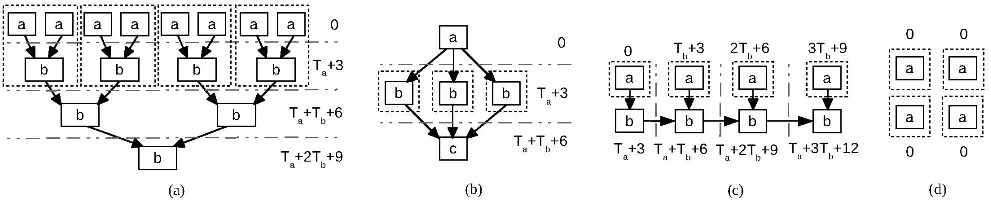


Fig. 10. DFGs, scheduling, and parallel simulation support of fundamental skeletons: (a) Recursively Partitioned; (b) Farm; (c) Systolic; (d) Crowd.

- **Systolic**. It consists of processes that have data flowing between them, and that may operate concurrently in a pipelined fashion.
- **Crowd**. Similar to the Systolic skeleton except for that there is no data flow between the concurrently operating processes.

Note that we are not using Campbell's *Task queue* skeleton, as it is a generalized version of *Farm* as which not suited for hardware implementation. It is also worth noting that the sizes of these structures are unfixed. For example, a *Recursively Partitioned* skeleton also suits problems with more layers as long as the solutions are collected recursively. Other classifications, e.g. the one used in STAPL framework [18], can also be adopted in the synthesis flow.

For each problem class, we specify the scheduling, placement, and routing algorithms, and store them in a repository as shown in Box 2 of Fig. 7. With respect to the placement, primitive circuits and the hardware design they constitute are represented by their bounding rectangles. These rectangles are not allowed to overlap each other. We take *Recursively Partitioned* skeleton as an example of the solution templates. The placement algorithm specified in this skeleton places boxes *a* and *b* following a binary-tree patternas shown in Fig. 11(a). All the intermediate data are transferred via two mirrors, which are minimum number required (see Section II-B). Since the communication cost is known as three cycles, the problem can be scheduled as the expressions shown in Fig. 10(a). The expressions are the starting moments of corresponding operations, in which T_x represents the latency of box x, e.g., T_a means a's latency. The dash-dot lines divide the DFG into several regions. Boxes in each of them execute in parallel. For other skeletons shown in Fig. 10, the scheduling results are also marked in a similar way.

The skeleton can break the cyclic dependence of scheduling, placement, and routing that we discussed in Section III-A. The reason is that these algorithms are defined altogether instead of separately. Limiting the problems' DFG structures facilities the development of these algorithms. For instance, the binary-tree placement algorithm improves the performance for the *Recursively Partitioned* skeleton, but it cannot be applied to other problems.

D. Skeleton Instantiation

A skeleton generates a hardware design after instantiated with primitive circuits or other hardware designs. In the latter case, we can solve complex problems that do not fit any fundamental skeleton. One advantage of the skeleton-based flow is that the users do not need to take care of implementation details. Instead, they just need to analysis the DFG and apply the right skeleton.

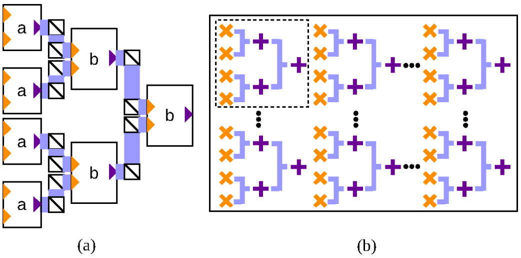


Fig. 11. Skeleton layout and nesting: (a) Binary-tree layout pattern; (b) 4×4 matrix multiply.

Suppose we intent to map the matrix multiply algorithm on CIM:

$$
\begin{aligned}
AB &= \left(\vec{a_1}^{\intercal} \quad \vec{a_2}^{\intercal} \quad \cdots \quad \vec{a_n}^{\intercal}\right)^{\intercal} \left(\vec{b_1} \quad \vec{b_2} \quad \cdots \quad \vec{b_n}\right) \\
&= \begin{pmatrix}
\vec{a_1}\cdot\vec{b_1} & \vec{a_1}\cdot\vec{b_2} & \cdots & \vec{a_1}\cdot\vec{b_n} \\
\vec{a_2}\cdot\vec{b_1} & \vec{a_2}\cdot\vec{b_2} & \cdots & \vec{a_2}\cdot\vec{b_n} \\
\vdots & \vdots & \ddots & \vdots \\
\vec{a_n}\cdot\vec{b_1} & \vec{a_n}\cdot\vec{b_2} & \cdots & \vec{a_n}\cdot\vec{b_n}
\end{pmatrix},
\end{aligned}
\tag{1}
$$

where $\vec{a_i}$ is a row vector of matrix A, and $\vec{b_i}$ is a column vector of B. It is a complex algorithm that does not fit any skeleton. However, we can see that it contains repetitive patterns. Each element of the result matrix is an inner product of two vectors. Thus, we can divide it into two levels. The top level is a *Crowd* skeleton because there are no data flows between these elements. The lower level is the vector inner product function. This function suits a *Recursively partitioned* skeleton, with "a" and "b" boxes in Fig. 10 replaced as multipliers and adders.

To implement the matrix multiply, we need to build the system bottom-up. First, we instantiate a *Recursively partitioned* skeleton with the multiplier and the adder. After that, we instantiate the *Crowd* skeleton with the inner product just generated. We assume both matrices are 4×4, so the vector size of the inner product is also 4. Fig. 11(b) represents the generated system. The symbols "×" and "+" stand for multipliers and adders while dashes between them are communication paths. Each subsystem, as shown in the dashed box, has a detailed layout following the binary-tree pattern. If

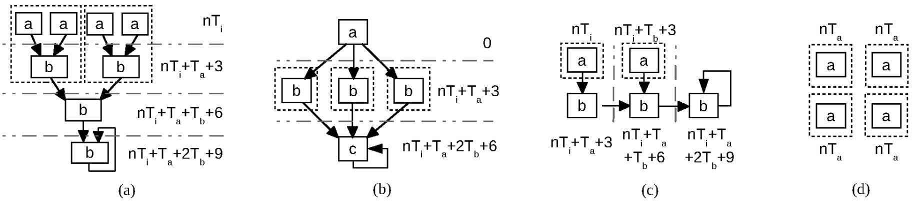


Fig. 12. DFGs, scheduling, and parallel simulation support of fundamental skeletons with hardware reuse: (a) Recursively Partitioned; (b) Farm; (c) Systolic; (d) Crowd.

an application cannot fit any existing skeleton, it is necessary to develop a new one. In this case, the skeleton repository should be extended.

In a similar way, we can implement the *FIR* function shown in Fig. 8. The *for* loop at line 9 and 10 suits the *Systolic* skeleton, where “a” and “b” boxes are instantiated with multipliers and adders. To further accelerate the program, we instantiate the *Crowd* skeleton with the generated *FIR* kernel to enable the high-level parallelism represented by the *for* loop in the main function (line 15 and 16).

IV. Constraints and Optimizations

Next, we introduce more implementation details of CIM’s synthesis flow. First, we present the methods that satisfy the area constraints in Section IV-A. Then in Section IV-B, we analyze data transfer patterns and use them to decrease the bandwidth and optimize the design. Thereafter, we describe in Section IV-C the tool’s parallel SystemC simulation feature used to deal with large designs.

A. Area Constraint and Hardware Reuse

Our skeleton-based flow supports user-defined area constraint, which represents the chip size or the area reserved for a hardware design. When the design area exceeds the constraint, we need to allocate less hardware and reuse it over time. We first adjust the DFGs to preserve the functionality. Then, the scheduling, placement, and routing algorithms are modified accordingly.

The modified DFGs are shown in Fig. 12. Boxes in these DFGs execute n times instead of just once in Fig. 10. Loop-backs are introduced to accumulate the result generated in different iterations. Comparing Fig. 10(a) and Fig. 12(a) as examples, we can find the box b at the lowest level both accepts two inputs. In the former DFG, these two inputs come from two sub-DFGs at higher levels simultaneously. In the latter one, they are from the same sub-DFG sequentially. The result would be the same as long as b is correctly initialized. For instance, if b is an adder, its initial output should be set to zero.

The mapping and routing algorithms for these skeletons are similar to the *flattened* designs, i.e. the skeletons without hardware reuse. A *demultiplexer*, or a *demux* in short, is introduced into each box that has a loop-back routing. The demux can route the output signal to loop-back during computing, and send it to the output port of the whole design at the final stage. The scheduling results are also indicated in Fig. 12; T_i in these expressions means the largest latency among all the boxes. It is usually called *initiation interval*, which is the number of cycles that must elapse between two sets of inputs.

```
   Data: Area constraint $A_c$, problem size $S_p$
   Result: A valid hardware design
 1 Calculate flattened design's area $A_f$;
 2 if $A_f < A_c$ then return The flattened design ;
 3 $N_u \leftarrow S_p$,$N_l \leftarrow 1$ ;     // Upper & lower bounds
 4 while $N_l + 1 < N_u$ do        // Search space > 1
 5 |   $N_c \leftarrow (N_u + N_l)/2$;       // Set reuse times
 6 |   Build a design for problem size $Ceil(S_p/N_c)$;
 7 |   Calculate current design's area $A_r$;
 8 |   if $A_r < A_c$ then     // Shrink search space
 9 |   |   $N_u \leftarrow N_c$;
10 |   else
11 |   |   $N_l \leftarrow N_c$;
12 |   end
13 end
14 return The design for problem size $Ceil(S_p/N_u)$;
```

Fig. 13. Build designs under area constraint.

Fig. 13 shows the procedure of constructing designs with hardware reuse. First, we build a flattened design without hardware reuse (line 1). If the area meets the constraint, this design will be returned immediately (line 2). Otherwise, hardware reuse is required. In this case, we use binary search to decide how many times the hardware needs to be reused (line 3 to line 13). The initial search space is between one and the problem size S_p (line 3), and the exit condition is that the search space has shrunk to one (line 4). When the hardware is reused for N_c times, each time the hardware only needs to process S_p/N_c inputs. We build a new hardware design for this problem size (line 6) and calculate its area (line 7). Then we update the upper or lower bounds depending on whether the area meets the constraint (line 8 to line 12). The final design is for problem size S_p/N_u, which has lowest latency and meets the area constraint.

B. Data Transfer and Bandwidth Constraint

After building the hardware for computation following previous sections, we need to consider their input/output data transfer, which also has an important impact on the overall performance. This section focuses on the communication between

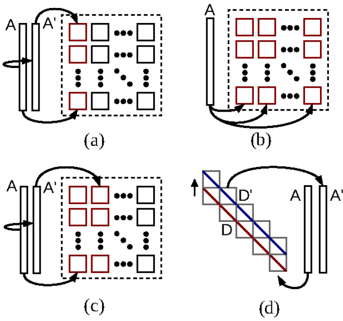


Fig. 14. Data transfer for spatial and temporal patterns and shift operation: (a) Shifts; (b) Duplication; (c) Offset Value; (d) Shift operation implementation.

RRAM and CIM (See Fig. 5). In this paper, we assume the data has been stored in the RRAM. The communication between RRAM and other components, such as CPU and storage, is beyond the scope.

Before presenting our data transfer solution, let us examine the patterns in the input/output data. In the *FIR* function shown in Fig. 8, lines 9 and 10 specify the computation while lines 6 to 8 describe the input data transfer. The computation part has two input arrays named *pre* and *coe*. In each execution, *pre* is shifted from the previous iteration (lines 6 to 8). Similar patterns are common in other programs. By leveraging these patterns, the data can be transferred more efficiently. These patterns can be either temporal or spatial. The temporal patterns we recognized include:

- **Constants**. The data does not change in all/some iterations, like the *coe* array in *FIR* function. (The initialization of *coe* array is omitted as shown in Fig. 8.)
- **Shifts**. The data should be shifted before it is applied to different iterations. This is the case of *pre* array in *FIR* function.
- **None**. No aforementioned temporal relations among the data.

The spatial patterns are:

- **Duplication**. The same data is used in different parts of the design. E.g., $\vec{a_1}$ is the input array for all the inner product in the first row of the matrix multiply as shown in Formula 1.
- **Offset Value**. The original data and its shifted versions are applied to different parts of the design. In Section III-D, we introduced that the *FIR* function can be implemented with the *Crowd* and the *Systolic* skeletons. It means duplicated hardware work in parallel. In this case, the *pre* array has an *Offset Value* pattern.
- **None**. No aforementioned spatial relations among the data.

Next, we will show the way to deal with the above patterns in case of CIM design. Fig. 14 shows the data transfer procedures for different patterns. The dotted box represents CIM, and the boxes inside it are logic units. The rectangle on the left symbolizes the input data arrays stored in RRAM. We will not show the solution for the *Constants* pattern since the data does not change. Other solutions are listed below.

- **Shifts**. As shown in Fig. 14(a), first the original data A is transferred to CIM. Then, it is shifted in RRAM. After one iteration of execution, the new data A' is transferred to CIM for the next iteration.
- **None**. Data is transferred from RRAM to CIM column by column.
- **Duplication**. Data is simultaneously transferred to multiple columns as shown by Fig. 14(b), following the broadcast method proposed by Xie [15]. It is faster and more energy efficient compared with column-by-column data transfer.
- **Offset Value**. Similar to the solution for the *Shifts* pattern except that the shifted data A' is now transferred to other parts of the design within the same iteration as A, as illustrated by Fig. 14(c).

The solutions for *Offset Value* and *Shifts* patterns both require the shift operation. This is conducted by using two groups of mirrors following the steps shown in Fig. 14(d). First, the data A is copied to mirrors D. Then, all the bits are shifted to mirrors D' in parallel. Finally, the data is copied back as A', which is the shifted version of A.

We use matrix multiply as an example to show the usage of data patterns. Fig. 15 specifies the matrix multiply with three skeletons. First, we set the area constraint (line 1), which represents a crossbar with $10^5 \times 10^5$ memristors. Then we define primitive circuits including the multiplier (line 2) and the adder (line 3). After that, three skeletons are instantiated: one *Recursively Partitioned* skeleton and two *Crowd* skeletons (lines 4 to 6). This instantiation is based on primitive circuits (such as *mul* and *add*), matrix parameters (such as m and n), as well as other skeletons; e.g., the instantiation of *row* makes use of *inner* (line 5), which is a *Recursively Partitioned* skeleton. Note that *Crowd* skeleton make use of two constants, HORZ and VERT, to specify the direction of duplicating circuits. HORZ in line 5 indicates *inner* is duplicated and placed in a horizontal direction (i.e., forming a row of *inner*). On the other hand, VERT in line 6 indicates that the former row of *inner* is duplicated and placed vertically, resulting in a matrix of *inner*. The parameters in parenthesises indicate the data patterns. Matrix multiply has two *duplication* (DUPL) patterns for rows and columns.

The communication bandwidth between RRAM and CIM

```
Data: Input matrices dimensions: $m$, $n$, and $k$
Result: Hardware design of matrix multiply $A_{m\times n} \times B_{n\times k}$
1 SetAreaCon(1e5, 1e5);
2 Multiplier mul(a, b);
3 Adder add;
4 Recur_ske inner⟨mul, add, $n$⟩(NONE, NONE);
5 Crowd_ske row⟨inner, $m$, HORZ⟩(DUPL, NONE);
6 Crowd_ske mm⟨row, $k$, VERT⟩(NONE, DUPL);
7 return mm.GenSystem();
```

Fig. 15. Pseudo codes of specifying matrix multiply in CIM compiling flow.

is also a significant constraint on the hardware design. The product of the bandwidth and initiation interval indicates the maximum data amount that can be transferred between two pipelining stages. If the computation kernel expects more data, it will stall. We can limit the size of hardware design to avoid such stall, so that the same performance can be achieved with a smaller area. The *Duplication* pattern reliefs the bandwidth constraint because it requires fewer data transfers from RRAM to CIM. If no bandwidth constraint is specified by the user, the theoretical maximum bandwidth is used. The theoretical bandwidth is N bits per Clock Cycle (CC), where N denotes the number of nanowires across the interface.

C. Parallel Simulation Support

Our compiler integrates parallel SystemC simulation support into skeletons' specification for acceleration and enabling large simulation scale. At the current development phase, the compiler generates SystemC files for behavior verification. However, the standard SystemC implementation [46] does not support parallelism, which limits the performance and scale of the simulation. Therefore, we replace some channels with Message Passing Interface (MPI)-based communication. Subsequently, we can distribute the simulation to multiple machines.

Fig. 10 and Fig. 12 illustrate our parallel simulation support for each skeleton. The parts surrounded with dotted boxes are simulated in parallel by different threads (possibly on different machines). The number and sizes of these boxes are decided by the number of available threads, which is set by the user.

V. Experimental Results

We conducted three sets of experiments to validate the design flow. Section V-A uses the inner product, FIR filter, and Tiny Encryption Algorithm (TEA) as case studies to show the source codes and graphic outputs. After that, we analyze the scalability of the flow in Section V-B while considering FIR filter. Parallel SystemC simulation results will be presented in Section V-C. Finally, we discuss the strength and limitations of the proposed synthesis flow in Section V-D.

A. Case Studies

We use the inner product of vector size four (see Fig. 16(a)) and FIR filter with tap size three (see Fig. 16(b)) as two examples to show the generated graphic layout of the skeleton-based synthesis flow. The large and small rectangles represent multipliers and adders, respectively. Within them, the light yellow and dark blue triangles denote the input and output ports, and the light blue fields represent the area dedicated for wiring. The figures clearly show the usage of the binary tree and systolic patterns in these figures.

Next, we use a more complex case study, i.e. the TEA, to show how the skeleton-based design methods can be used to implement real-life applications. TEA is a simple block cipher that uses a 128-bit key to encrypt 64-bit data blocks [47]. Fig. 17 shows its C implementation. The function accesses the plaintext and the key with pointers (line 1), and the ciphertext

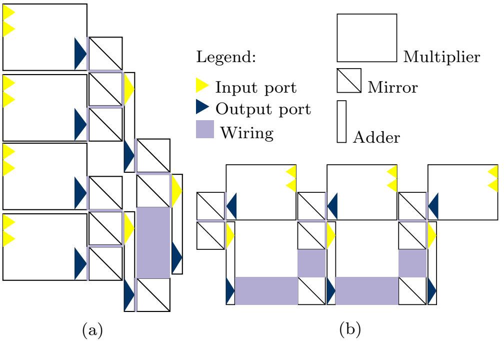


Fig. 16. Generated graphic output of study cases: (a) Inner product; (b) FIR filter.

```
1  void encrypt (unsigned* v, unsigned* k) {
2    unsigned v0=v[0], v1=v[1], sum=0, i;
3    unsigned delta=0x9e3779b9; //key schedule const
4    for (i=0; i < 32; i++) {
5      sum += delta;
6      v0 += ((v1<<4)+k[0])^(v1+sum)^((v1>>5)+k[1]);
7      v1 += ((v0<<4)+k[2])^(v0+sum)^((v0>>5)+k[3]);
8    }
9    v[0]=v0; v[1]=v1;
10 }
```

Fig. 17. Tiny Encryption Algorithm source codes.

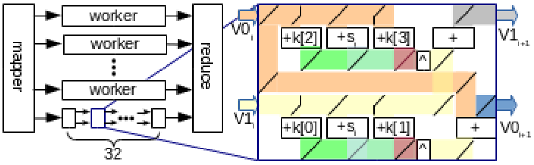


Fig. 18. Tiny Encryption Algorithm's hardware implementation with the *Farm* and *Systolic* skeletons.

is also returned via a pointer (line 9). A 32-bit constant ($0x9e3779b9$, line 3) is used to prevent simple attacks based on the symmetry of the rounds. The encryption process consists mainly of a loop of 32 iterations (line 4 to 8). Each iteration contains shift, addition, and XOR operations (line 6 and 7).

We manually designed a hardware unit as shown in the right part of Fig. 18 to implement one iteration of TEA. This unit has eight adders and two 3-input XOR operators, represented by rectangles with "+" and "^" symbols, respectively. As shown in the source code (line 6 and 7 in Fig. 17), most adders have one constant input. These constants are also provided in Fig. 18. "S_i" means variable sum's value in the ith iteration, which is available during compilation. The mirrors are represented by black slashes. They link the horizontal and vertical nanowires, which are illustrated by colored stripes. Different colors are used to indicate different data. Polylines ("╯" and "╭") represent special mirrors whose memristors are located in a line parallel to the diagonal. These special mirrors are used to implement shift operations. Next, this

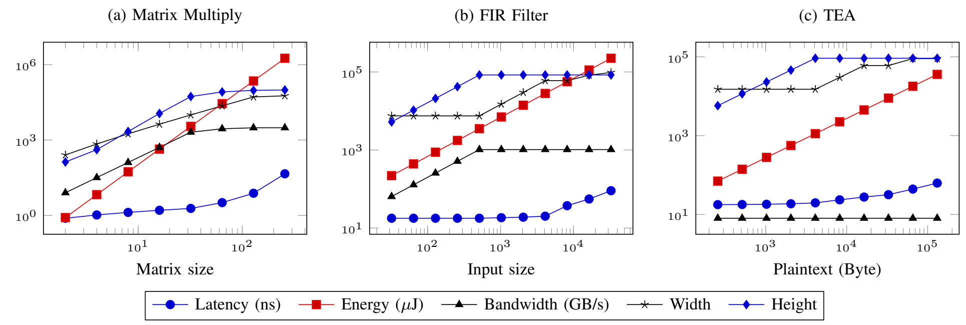


Fig. 19. Latency, energy, bandwidth, and area of scaling applications in CIM.

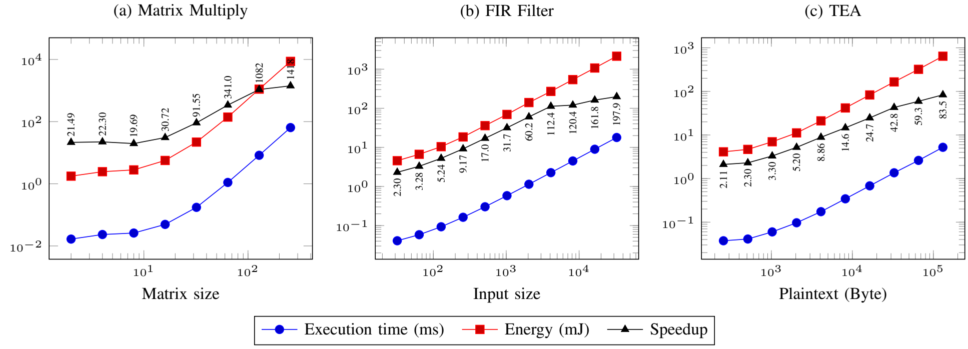


Fig. 20. Latency and energy of scaling applications on the multicore platform.

unit is duplicated using the *Systolic* skeleton as shown in the bottom left part of Fig. 18. The resulting circuit (*worker*) implements all the 32 iterations, thus representing an entire TEA function. This circuit is further duplicated by the *Farm* skeleton to speedup encrypting different parts of the plain text in parallel. The *Farm* skeleton uses two helper circuits (*mapper* and *reduce*) to split the plaintext and merge the ciphertext as shown in the left part of Fig. 18.

B. System Scaling

In this section, we compare CIM's performance against a multicore system to show the quality of our compiler's generation. The targeted multicore system is Intel Xeon X7460 processor that consists of six cores on a die of $503\,\text{mm}^2$, running at $2.66\,\text{GHz}$ each [48]. We assume the CIM chip to be only 10% of the area of Xeon X7460, and only 10% of the CIM chip is used for computation (the rest is used as RRAM, see Fig. 5). Then, the computation part contains about 10^{10} memristors according to the density predicted by ITRS [39]. Therefore, we add an area constraint $10^5 \times 10^5$ to the synthesis flow.

We varied the problem sizes to evaluate the scaling capabilities with area constraint and generated three cases: matrix multiply, FIR filter, and TEA. We assume the matrices to be square $n \times n$. In the FIR application, the taps number is fixed as 64, and input size changes; see Fig. 8. The problem size of TEA can be valued by the plaintext size. We assume CIM's clock frequency is $1\,\text{GHz}$, considering the memristor switching time is in the range of a hundred picoseconds [49]. The performance and the cost of generated designs are shown in Fig. 19. In all three cases, the latency increases faster when the area limit is reached. This indicates that the hardware is reused to meet the area constraint. Whether the hardware is reused or not, the energy consumption increases almost at the same rate as it is determined by the total number of operations. For matrix multiply and FIR, there is a positive correlation between the bandwidth and the crossbar height, since the data in different rows can be transferred in parallel (see Section II-B). On the other hand, TEA's bandwidth remains constant, because it uses a *mapper* circuit to split sequential inputs to the *worker* threads (see Fig. 18). In all three cases, the width and the height do not increase when they approach 10^5, due to the area constraint we set.

To show the quality of the synthesized designs, we evaluated the execution time and energy consumption of these applications on a multicore platform and compared the execution time against CIM. This evaluation is conducted with Sniper [50], and the energy consumption is reported by McPAT [51]. We employed a simulator instead of using real hardware because it benefits the reproducibility. The targeted hardware platform

TABLE II
Comparison Among Design Methodologies

	Application		Design quality		Development	
Design methodologies	Type	Size	Latency	Area	Efforts	Execution time
Manual	Regular	Large	Low	Large	Much	-
Skeleton-based (this work)	Half-regular	Large	Low	Large	Medium	Short
Fully automated	Irregular	Medium	Slightly high	Small	Little	Long

TABLE III
Baselines of Parallel Simulation

Applications	Size	Base (s)
Matrix multiply	64	1686
FIR filter	512	1036
TEA	4096	1859

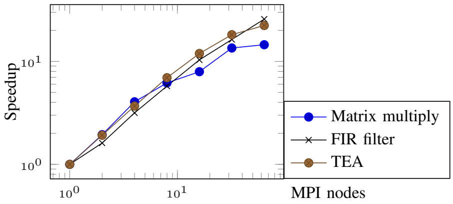


Fig. 21. Parallel simulation speedup.

is an Intel Xeon X7460 processor, which consists of six cores, each running at 2.66 GHz. These cores have 64 kB L1 cache each and share a 16 MB L3 cache. Every two cores share an L2 cache of 3 MB. The experimental results, including the speedup of CIM over the multicore platform, are shown in Fig. 20. The values of the speedup are marked beside the line. The maximum speedup for matrix multiply, FIR filter, and TEA is 1418x, 197.9x, and 83.5x, respectively. The energy consumption of multicore is about one order of magnitude larger than CIM for all the three cases.

C. Parallel Simulation

We enabled parallel simulation support to be able to simulate large designs. The parallel experiments are performed on a high-performance computer with 20 Intel Xeon E5-2670 HT cores, running at 2.50 GHz each. First, we obtained the baseline execution time which are based on sequential simulations. Table III shows the sizes of simulated applications and the corresponding simulation time. After that, we fixed the system size and changed the number of MPI nodes and generated eight configurations. For each of these configurations, we performed the simulation ten times and calculated the average execution time after removing the maximum and minimum values. Fig. 21 shows the speedup of each configuration over the sequential simulation as the baseline. The output data of all the parallel simulations are verified and found to match those of the sequential one. When MPI nodes are less than 16, the speedups are almost the same as the thread number. When the nodes number increases beyond 16, the speedup tends to saturation. It is understandable since the cores in hardware are limited. This result shows a good scalability.

D. Discussion

We compared the skeleton-based design flow, the manual design flow, and a potential fully automated flow in Table II to identify their characteristics. The fully automated flow is similar to existing VLSI design flows that can map any application to the hardware without using predefined solutions. Such a flow is currently not available due to design constraints of memristor-based CIM architectures that have been discussed in Section II-B and III-A. In addition, existing research on manual designs, such as [10], [52], have different assumptions on primitive circuits, hardware platforms, and applications as compared to this work. Therefore, the comparison is qualitative instead of quantitative. We first compare the supported applications of these three design methodologies. Manual designs can only handle regular applications such as parallel addition [52] and matrix multiply [10] due to design complexity. Skeleton-based flow requires the application to be regular at the top level while it has no restriction for the computational kernel at low level, as demonstrated in the TEA case study. The fully automated flow is the most flexible one with regard to the application type. However, the application size supported by the automated flow is not as large as the skeleton-based flow because the former has to explore the compute design space. Secondly, with respect to the quality of the generated designs, automated design flow cannot achieve optimal communication cost as discussed in Section III-A. However, since communication latency (typically 2-6 cycles) is relatively small compared to computation latency (tens to hundreds of cycles), the difference in performance between optimal design and suboptimal one would be minor. From an area point of view, the manual and skeleton-based flows have large white space in the designs, and hence require a larger design area than the automated flow. Finally, comparing their design efforts, the automated flow would be the easiest one to use. For the skeleton-based flow, the user is required to identify the patterns in the application; hence, it needs more effort. A skeleton-based synthesis tool executes quickly because it does not require design space exploration.

Despite the huge potential of this synthesis flow, memristor-based CIM architectures are still in their infancy stage and facing many challenges. The implementation of primitive circuits affects the performance of memristor-based computation, including CIM. The latency of the multiplier (499 CC) and the adder (264 CC) that we proposed in the experiments are still much greater than those operators implemented with CMOS technology. Therefore, even greater performance improvement can be achieved if these primitive circuits are improved. On the other hand, only few arithmetic operators have been implemented in memristor crossbars. It limits the number of algorithms that we can map to CIM. Since memristor-based computation is emerging, future research will produce more and better circuits designs, and they will also benefit CIM.

Endurance is another concern [6], [53]. It restricts the potentials of memristor-based computation and CIM. Currently, the largest number of allowed write/erase operations on a memristor is only around 10^{12} [54], [55], but this number is believed to be able to reach 10^{15} [56] in the future. Nevertheless, CIM is an accelerator that typically has a lower workload profile than CPUs. For instance, CIM can be used in specific applications that are mostly in power-off state, such as wearable devices, smart IDs, and sensors [57], [58]. These applications have a lower endurance requirement.

VI. Conclusion and Future Work

Memristor-based CIM architecture requires a radically new development flow due to the characteristics of the memristor crossbar. We built a desirable synthesis flow for CIM based on an extension of algorithmic skeletons. In this flow, scheduling, placement, and routing algorithms are specified according to problems' DFG structures. We also investigated data patterns existing in stream applications and combined them with skeletons. To accelerate SystemC simulation, we integrated it with MPI. This work is verified using three applications, and their latency is compared against a multicore system. Primary results show the feasibility and potential of the proposed approach.

In future work, we will further investigate the communication between the RRAM and other components, such as the storage and the CPU. We are also developing a new Domain-Specific Language (DSL) to better describe CIM skeletons, especially with data patterns.

References

[1] M. Saecker and V. Markl, *Big Data Analytics on Modern Hardware Architectures: A Technology Survey*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36318-4_6

[2] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra *et al.*, "Top ten exascale research challenges," DOE ASCAC, Tech. Rep., 2014.

[3] T. Skotnicki, J. A. Hutchby, T.-J. King, H. S. P. Wong, and F. Boeuf, "The end of cmos scaling: toward the introduction of new materials and structural changes to improve mosfet performance," *IEEE Circuits and Devices Magazine*, vol. 21, no. 1, pp. 16–26, Jan 2005.

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

[5] 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.

[6] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar, "Resistive gp-simd processing-in-memory," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 57:1–57:22, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2845084

[7] 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*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 173:1–173:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898064

[8] P. E. Gaillardon, L. Amar, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. D. Micheli, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 427–432. [Online]. Available: http://ieeexplore.ieee.org/document/7459349/

[9] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, "Memristor based computation-in-memory architecture for data-intensive applications," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1718–1725. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2757210

[10] A. Haron, J. Yu, R. Nane, M. Taouil, S. Hamdioui, and K. Bertels, "Parallel matrix multiplication on memristor-based computation-in-memory architecture," in *2016 International Conference on High Performance Computing Simulation (HPCS)*. IEEE, July 2016, pp. 759–766.

[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, Oct 2014.

[12] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[13] L. Xie, H. A. D. Nguyen, J. Yu, A. Kaichouhi, M. Taouil, M. Al-Failakawi, and S. Hamdioui, "Scouting logic: A novel memristor-based logic design for resistive computing," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2017, pp. 176–181.

[14] C. L. Seitz, *Introduction to VLSI systems*. Reading, MA: Addison-Wesley, 1980, ch. System timing, pp. 218–262.

[15] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, "Interconnect networks for memristor crossbar," in *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*. IEEE, July 2015, pp. 124–129.

[16] S. H. Gerez, *Algorithms for VLSI design automation*. New York: Wiley, 1999, vol. 8.

[17] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[18] M. Zandifar, M. Abdul Jabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger, "Composing algorithmic skeletons to express high-performance scientific applications," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 415–424.

[19] C. Nugteren and H. Corporaal, "Bones: An automatic skeleton-based c-to-cuda compiler for gpus," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 35:1–35:25, Dec. 2014.

[20] Y. Wang and Z. Li, "Gridfor: A domain specific language for parallel grid-based applications," *International Journal of Parallel Programming*, vol. 44, no. 3, pp. 427–448, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0348-z

[21] M. Goli and H. Gonzalez-Velez, "Heterogeneous algorithmic skeletons for fast flow with seamless coordination over hybrid architectures," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, Feb 2013, pp. 148–156.

[22] K. Benkrid, D. Crookes, J. Smith, and A. Benkrid, "High level programming for fpga based image and video processing using hardware skeletons," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*. IEEE, March 2001, pp. 219–226.

[23] K. Benkrid and D. Crookes, "From application descriptions to hardware in seconds: a logic-based approach to bridging the gap," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 4, pp. 420–436, April 2004.

[24] J. Yu, R. Nane, A. Haron, S. Hamdioui, H. Corporaal, and K. Bertels, "Skeleton-based design and simulation flow for computation-in-memory architectures," in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, July 2016, pp. 165–170.

[25] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[26] J. Borghetti, Z. Li, J. Straznicky, X. Li, D. A. Ohlberg, W. Wu, D. R. Stewart, and R. S. Williams, "A hybrid nanomemristor/transistor logic circuit capable of self-programming," *Proceedings of the National Academy of Sciences*, vol. 106, no. 6, pp. 1699–1703, 2009.

[27] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, "Fast boolean logic mapped on memristor crossbar," in *Computer Design (ICCD), 2015 33rd IEEE International Conference on*. IEEE, Oct 2015, pp. 335–342.

[28] G. Rose, J. Rajendran, H. Manem, R. Karri, and R. Pino, "Leveraging memristive systems in the construction of digital logic circuits," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 2033–2049, June 2012.

[29] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, July 2016.

[30] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, and R. Waser, "Beyond von neumann - logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, no. 30, p. 305205, 2012.

[31] 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, Nov 2014.

[32] P. J. Kuekes, D. R. Stewart, and R. S. Williams, "The crossbar latch: Logic value storage, restoration, and inversion in crossbar circuits," *Journal of Applied Physics*, vol. 97, no. 3, p. 034301, 2005. [Online]. Available: http://dx.doi.org/10.1063/1.1823026

[33] P. L. Thangkhiew, R. Gharpinde, P. V. Chowdhary, K. Datta, and I. Sengupta, "Area efficient implementation of ripple carry adder using memristor crossbar arrays," in *2016 11th International Design Test Symposium (IDT)*, Dec 2016, pp. 142–147.

[34] R. Gnanasekaran, "A fast serial-parallel binary multiplier," *IEEE Trans. Comput.*, vol. 34, no. 8, pp. 741–744, Aug. 1985. [Online]. Available: http://dx.doi.org/10.1109/TC.1985.1676620

[35] E. Lehtonen, J. H. Poikonen, and M. Laiho, *Memristive Stateful Logic*. Cham: Springer International Publishing, 2014, pp. 603–623. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-02630-5_27

[36] C.-L. Tsai, F. Xiong, E. Pop, and M. Shim, "Resistive random access memory enabled by carbon nanotube crossbar electrodes," *Acs Nano*, vol. 7, no. 6, pp. 5360–5366, 2013.

[37] S. Lee, J. Sohn, Z. Jiang, H.-Y. Chen, and H.-S. P. Wong, "Metal oxide-resistive memory using graphene-edge electrodes," *Nature communications*, vol. 6, no. 8407, pp. 1–7, 2015.

[38] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD '15. New York, NY, USA: ACM, 2015, pp. 171–178. [Online]. Available: http://doi.acm.org/10.1145/2717764.2717783

[39] International RoadMap Committee, "International technology roadmap for semiconductors 2.0," Tech. Rep., 2015. [Online]. Available: www.itrs2.net/

[40] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[41] I. Ashraf, V. Sima, and K. Bertels, "Intra-application data-communication characterization," in *Proc. 1st International Workshop on Communication Architectures at Extreme Scale*, Frankfurt, Germany, July 2015, pp. 1–11.

[42] I. Ashraf, "Communication driven mapping of applications on multicore platforms," Ph.D. dissertation, Delft University of Technology, Delft, Netherlands, April 2016.

[43] C. Luk and et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI '05*. New York, NY, USA: ACM, 2005, pp. 190–200.

[44] A. Canis, J. Choi, B. Fort, B. Syrowik, R. L. Lian, Y. T. Chen, H. Hsiao, J. Goeders, S. Brown, and J. Anderson, *LegUp High-Level Synthesis*. Cham: Springer International Publishing, 2016, pp. 175–190. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26408-0_10

[45] D. K. Campbell, "Clumps: a candidate model of efficient, general purpose parallel computation," Ph.D. dissertation, University of Exeter, 1994.

[46] Accellera Systems Initiative, "Systemc," 2016. [Online]. Available: http://accellera.org/downloads/standards/systemc

[47] D. J. Wheeler and R. M. Needham, *TEA, a tiny encryption algorithm*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 363–366. [Online]. Available: http://dx.doi.org/10.1007/3-540-60590-8_29

[48] Intel, "Xeon processor x7460," 2008. [Online]. Available: http://ark.intel.com/products/36947/Intel-Xeon-Processor-X7460-16M-Cache-2_66-GHz-1066-MHz-FSB

[49] A. C. Torrezan, J. P. Strachan, G. Medeiros-Ribeiro, and R. S. Williams, "Sub-nanosecond switching of a tantalum oxide memristor," *Nanotechnology*, vol. 22, no. 48, p. 485203, 2011. [Online]. Available: http://stacks.iop.org/0957-4484/22/i=48/a=485203

[50] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629677

[51] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669172

[52] H. A. D. Nguyen, L. Xie, M. Taouil, R. Nane, S. Hamdioui, and K. Bertels, "Computation-in-memory based parallel adder," in *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*. IEEE, July 2015, pp. 57–62.

[53] P. Pouyan, E. Amat, and A. Rubio, "Memristive crossbar memory lifetime evaluation and reconfiguration strategies," *IEEE Transactions on Emerging Topics in Computing*, vol. PP, no. 99, pp. 1–1, 2016.

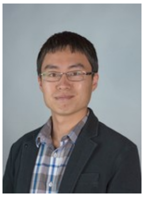
[54] F. Miao, J. Yang, J. Strachan, W. Yi, G. Ribeiro, and R. Williams, "Memristor with channel region in thermal equilibrium with containing region," Mar. 1 2016, uS Patent 9,276,204. [Online]. Available: https://www.google.com/patents/US9276204

[55] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo *et al.*, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta2o5-x/tao2- x bilayer structures," *Nature materials*, vol. 10, no. 8, pp. 625–630, 2011.

[56] K. Eshraghian, K. R. Cho, O. Kavehei, S. K. Kang, D. Abbott, and S. M. S. Kang, "Memristor mos content addressable memory (mcam): Hybrid architecture for future high performance search engines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1407–1417, Aug 2011.

[57] X. Li, K. Ma, S. George, J. Sampson, and V. Narayanan, "Enabling internet-of-things: Opportunities brought by emerging devices, circuits, and architectures," in *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Sept 2016, pp. 1–6.

[58] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs, L. Xie, N. Wald, S. Joshi, H. M. Elsayed, H. Corporaal, and K. Bertels, "Memristor for computing: Myth or reality?" in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 722–731.



Jintao Yu (S'17) received the M.Sc degree in China in 2013. He is currently a Postdoctoral Researcher at Delft University of Technology. He is currently a pursuing the Ph.D. degree with the Computer Engineering Laboratory, Delft University of Technology. His current research interests include memristor-based computing systems and domain-specific languages.

Razvan Nane (S'11—M'14) received his Ph.D. in Computer Engineering from Delft University of Technology, The Netherlands in 2014. He is currently a Postdoctoral Researcher at Delft University of Technology. He is the main developer of the DWARV C-to-VHDL hardware compiler. His current research interests are high-level synthesis for reconfigurable architectures, hardware/software co-design methods for heterogeneous systems, and compilation and simulation techniques for emerging memristor-based in-memory computing high-performance architectures.

Imran Ashraf (S'10—M'15) is a Postdoctoral researcher at Delft University of Technology. He received his Ph.D. in Computer Engineering from Delft University of Technology, The Netherlands in 2016. His research interests were advanced profiling, code parallelization, communication driven mapping of applications on multicore platforms. Currently, Imran is working as Post-Doctoral Researcher at Quantum Computing Lab, QuTech, TU Delft. His recent research also focuses on compilation techniques for quantum computing.

Mottaqiallah Taouil (S'10—M'15) received the M.Sc. and Ph.D. degrees (both with Hons.) in computer engineering from the Delft University of Technology, Delft, The Netherlands. He is currently a Post-Doctoral Researcher with the Dependable Nano-Computing Group, Delft University of Technology. His current research interests include reconfigurable computing, embedded systems, very large scale integration design and test, built-in-self-test, and 3-D stacked integrated circuits, architectures, design for testability, yield analysis, and memory test structures.

Said Hamdioui (M'99—SM'11) is currently a Chair Professor on Dependable and Emerging Computer Technologies and head of the Computer Engineering Laboratory of the Delft University of Technology (TUDelft), the Netherlands. Prior to joining TUDelft, Hamdioui worked for Intel Corporation (Califorina, USA), Philips Semiconductors R&D (Crolles, France) and for Philips/ NXP Semiconductors (Nijmegen, The Netherlands). His research focuses on two domains: Dependable CMOS nanocomputing (including Reliability, Testability, Hardware Security) and emerging technologies and computing paradigms (including 3D stacked ICs, memristors for logic and storage, in-memory-computing). He owns one patent and has published one book and co-authored over 170 conference and journal papers. He delivered dozens of keynote speeches, distinguished lectures, and invited presentations and tutorial at major international forums/conferences/schools and at leading semiconductor companies. Hamdioui is a Senior member of the IEEE, Associate Editor of IEEE Transactions on VLSI Systems (TVLSI), and he serves on the editorial board of IEEE Design & Test, and of the Journal of Electronic Testing: Theory and Applications (JETTA). He is also member of AENEAS/ENIAC Scientific Committee Council (AENEAS = Association for European NanoElectronics Activities).

Henk Corporaal received the Ph.D. degree in electrical engineering, in the area of computer architecture, from the Delft University of Technology, The Netherlands. Currently, he is a Professor of embedded system architectures with the Eindhoven University of Technology, The Netherlands. He has co-authored over 300 journal and conference papers in the (multi)processor architecture and embedded system design area. Furthermore, he invented a new class of very long instruction word architectures, the Transport Triggered Architectures, which is used in several commercial products and by many research groups. His current research interests include single and multiprocessor architectures, deep learning, and the predictable design of soft and hard real-time embedded systems.

Koen Bertels (M'05) received the Ph.D. degree in computer information systems from the University of Antwerp, Antwerp, Belgium. He is currently a Professor and the Head of the Computer Engineering Laboratory, Delft University of Technology, Delft, The Netherlands, where he is researching on quantum computing as a Principle Investigator with the Qutech Research Center. He has co-authored more than 30 journal papers and 150 conference papers. His current research interests include heterogeneous multicore computing, investigating topics ranging from compiler technology, runtime support, and architecture. Prof. Bertels has been the General and Program Chair for various conferences such as FPL, RAW, and ARC.

APmap: An Open-Source Compiler for Automata Processors

Jintao Yu, *Student Member, IEEE,* Muath Abu Lebdeh, *Student Member, IEEE,* Hoang Anh Du Nguyen, Mottaqiallah Taouil, *Member, IEEE,* and Said Hamdioui, *Senior Member, IEEE*

***Abstract*—A novel type of hardware accelerators called *automata processors* (APs) have been proposed to accelerate finite-state automata. The bone structure of an AP is a hierarchical routing matrix that connects many memory arrays. With this structure, an AP can process an input symbol every clock cycle, and hence achieve much higher performance compared to conventional architectures. However, the design automation for the APs is not well researched. This paper proposes a fully automated tool named *APmap* for mapping the automata to APs that use a two-level routing matrix. APmap first partitions a large automaton into small graphs and then maps them. Multiple transformations are applied to the automaton by APmap to meet hardware constraints. The experiments on a standard benchmark suite show that our approach leads to around 19% less storage utilization compared to state of the art.**

***Index Terms*—Automata Processor, design automation, mapping, graph partitioning.**

I. Introduction

FINITE-state automata (FSA) are widely used in domains such as network security [1], bioinformatics [2], and artificial intelligence [3]. Some innovative hardware designs re-purpose memory array for accelerating FSA execution, e.g., Micron Automata Processor (MAP) [4], Cache Automaton [5], and RRAM-AP [6]. These accelerators store many states in memory arrays and distribute each input symbol to all the states simultaneously. Based on the input symbol, a state activates other states via a hierarchical routing matrix. These actions are repeated every clock cycle, and hence these accelerators achieve much high throughput [4], [5], [7]. We refer to these accelerators as *automata processors* (APs). The routing matrix mimics the transition function of FSA. It is implemented with memory arrays that are connected with rich wiring. The routing matrix is configured for specific FSA by writing configurable bits to the memory array. For example, the routing matrix of Cache Automaton connects 32k states and contains 10M configurable bits [5]. Therefore, design automation is required for mapping FSA to the APs.

Currently, there are no open-source design tools available for the APs. The authors of Cache Automaton described their methodology of mapping FSA to the hardware [5]. However, not all the details are explained, and their tool is not publicly available. Following their methodology, some FSA cannot be mapped directly due to the constraints on the routing matrix. As a result, these FSA have to be transformed into other equivalent forms [5]. This step is iterative and requires experience. As for other related works, Micron provides a commercial software development kit (SDK) for MAP. Since this SDK is closed-source, it cannot be adapted for other architectures such as Cache Automaton. The compiler of RAPID can generate mapping by duplicating an initial result, e.g., produced by MAP SDK [8]. Therefore, it cannot be used alone. Wadden *et al.* have developed an open-source tool named *ATR* to estimate the resource needed for mapping an application to MAP [9]. This tool is based on *VPR*, a routing tool that targets a 2D-mesh structure such as FPGAs. This structure is different from the hierarchical routing matrix of APs, and hence ATR cannot produce accurate results. While open-source tools, such as REAPR [10] and Grapefruit [11], have been proposed to map applications to FPGAs, a similar one that targets APs is still needed.

This paper addresses the above issues and presents *APmap*[1] (Automata Processor mapping tool), an open-source compiler for APs that are based on a two-level routing matrix, such as Cache Automaton and RRAM-AP. Note that APmap cannot be applied to MAP due to its algorithm limitation. APmap uses multiple strategies to change given FSA to equivalent forms so that they can meet the constraints of the routing matrix. Therefore, the compilation process does not require any user involvement. The main contributions of this paper are:

- A methodology to automatically map automata to APs that are based on a two-level routing matrix. The methodology optimizes the *storage utilization*;
- An *open-source* tool APmap based on the proposed methodology. This tool can be adapted to various designs by altering its parameters;
- An evaluation of APmap and comparison with state of the art.

The rest of the paper is organized as follows. In Section II, we explain the working principle and the routing matrix of APs. Section III presents the methodologies of APmap. Next, Section IV evaluates APmap's performance using ANMLzoo. After a brief discussion in Section V, Section VI concludes the paper.

II. Background

A. Automata Processors

The APs share a generalized architecture as shown in Fig. 1 [6]. In every clock cycle, an input symbol I is processed using three major steps:

The authors are with the Laboratory of Computer Engineering, Delft University of Technology, Delft, the Netherlands. E-mail: {J.Yu-1, M.F.M.AbuLebdeh, H.A.DuNguyen, M.Taouil, S.Hamdioui}@tudelft.nl.

[1] APmap can be downloaded at https://github.com/yjt98765/apmap

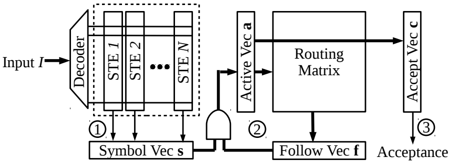


Fig. 1. General architecture of Automata Processors [6].

① **Input symbol matching.** All the states that have incoming transitions occurring on I are identified in this step. Each state is presented by column vectors called state transition element (STE) that are pre-configured based on the targeted automaton. The decoder activates one of the word lines according to the input symbol I. If an STE has an incoming transition occurring on I, its output is logic 1; otherwise, the output is logic 0. The outputs of all STEs are mapped to a vector called Symbol Vector $\mathbf{s}$.

② **Active state processing.** It generates all the possible states that can be reached from the currently active states (stored in Active Vector $\mathbf{a}$) based on the transition function (stored in the routing matrix), and stores the result in the Follow Vector $\mathbf{f}$. This step also generates the next active states by bit-wise ANDing $\mathbf{s}$ and $\mathbf{f}$.

③ **Output identification.** Accept Vector $\mathbf{c}$ is pre-configured based on the automaton's accepting states C. This step checks the intersection of $\mathbf{a}$ and $\mathbf{c}$ to decide whether the input sequence is accepted.

Multiple components, including STEs, the routing matrix, and Accept Vector $\mathbf{c}$, need to be configured based on the targeted FSA. The configuration will be generated by APmap.

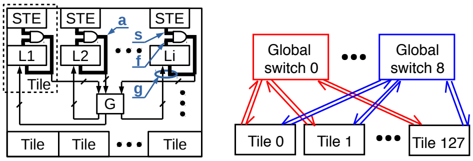
B. Routing Matrix

The routing matrix implements the transition function of an automaton. Its input and output are Active Vector $\mathbf{a}$ and Follow Vector $\mathbf{f}$, respectively. The lengths of these two vectors are both N, i.e., the state number of the automaton. Each member in the vectors is a Boolean value, corresponding to an automaton state. The routing matrix of the existing APs all consist of multiple components that are linked in a hierarchical style. The routing matrix of MAP contains four levels; Cache Automaton and RRAM-AP contain two, i.e., global and local switches. In Cache Automaton and RRAM-AP, the global switches are located at the center of the chip while the local switches are distributed. 256 STEs, a local switch, 256 AND gates, and a decoder are grouped as a *tile*, as shown in Fig. 2a. The input symbol I is sent to all the tiles in parallel.

APmap targets the routing matrix of the space-optimized design of Cache Automaton, which consists of 128 tiles, eight 1-way global switches, and a 4-way global switch. The connection between the tiles and global switches is shown in Fig. 2b. Each tile has two input wires from and two output wires to every 1-way global switch. Each tile also has eight input wires from and eight output wires to the 4-way global switch. In total, a tile has 24 input and 24 output wires.

III. APMAP METHODOLOGIES

We suggest several other tools to be used together with APmap to develop applications targeting APs. The application can be coded in RAPID, a high-level programming language designed for pattern-recognition processors such as APs [8]. RAPID's compiler generates Automata Network Markup Language (ANML), an XML-based format for describing automata [12], files as output. ANML can be parsed by *VASim* [13], a tool that simulates the execution of a homogeneous automaton. It also supports some important automata transformations, such as prefix merging. We modified VASim to preprocess the automata and generate the file formats used by APmap. These files describe the automata as a collection of connected components (CCs), i.e., non-overlapping subsets of the original automata. Finally, APmap produces the configuration files for APs.



(a) AP chip structure (b) Routing matrix

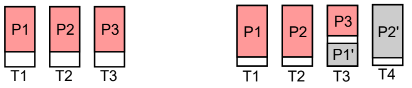
Fig. 2. Detailed structure of Cache Automaton and RRAM-AP.

APmap first sorts the CCs by their state numbers and then maps them one by one. In each iteration, APmap picks the largest unmapped CC and maps it to one or multiple tiles. In some cases, not all the space of these tiles are occupied by this CC. Therefore, APmap tries to find some small CCs to fill in the remaining space. This process repeats until all the CCs are mapped.

When a CC contains more than 256 states, it has to be mapped to multiple tiles. First, this CC is partitioned into several parts, which will be presented in detail in Section III-A. To increase the chance of mapping success, the partitioning process produces multiple solutions. Then, APmap examines the partitioning results with the hardware constraints on a tile. If there are conflicts, an extra process is applied to resolve the conflicts, which will be presented in Section III-C. Next, APmap tries to generate the configuration for the global switches and the tiles, which will be presented in Section III-B. If the configuration of global switches cannot be generated, APmap selects the next partitioning solution and repeats the previous processes. If none of these partitioning solutions leads to a valid configuration, the mapping flow fails.

A. CC Partitioning

APmap partitions a CC with the help of *METIS* [14], a widely-used graph partitioning tool. METIS divides an undirected graph into k non-overlapping parts while trying to cut the least number of edges. An edge is *cut* in a division means that the two nodes linked by the edge are assigned to different parts. First, we transform the CC to an undirected graph to make it acceptable for METIS. Self-loops are removed in this transformation. Then, we invoke METIS with two types of input parameters: the number of parts and the size constraints on those parts. The size of a part means the number of nodes contained in that part, and a size constraint is the desired size of a part. METIS produces the same number of parts as required; however, the size constraints are not guaranteed to



(a) Balanced partitioning (b) Unbalanced partitioning

Fig. 3. Different styles of partitioning a graph.

be satisfied. Therefore, we need to check the size of each part after the partitioning. If any part contains more than 256 nodes, we need to modify parameters and invoke METIS again.

This partitioning process is iterated with different parameters to lower the *storage utilization*, or *utilization* for short, which is referred to as the number of tiles that the automaton is mapped to. It is influenced by the partitioning result from three aspects. Firstly, each partitioned part will be mapped to different tiles; therefore, the number of parts is the most important factor for the final utilization. Secondly, the cutting edges will be mapped to *global wires*, i.e., the wires connecting tiles and global switches. The wire resource is limited. Therefore, a partitioning result that contains many cutting edges may lead to overhead for resolving the constraint conflict. Thirdly, the balancing of partitioned parts may also affect utilization. Fig. 3 shows two possible partitioning styles of a CC. It is partitioned into three parts, i.e., $P1$, $P2$, and $P3$, represented by gray rectangles. The size of the rectangle indicates the size of the part, and the total size of the three parts are equal in the two partitioning styles. These parts are mapped to three tiles, i.e., $T1$, $T2$, and $T3$, represented by transparent rectangles. In the balanced partitioning, as shown in Fig. 3a, all the parts have similar sizes. In the unbalanced partitioning, as shown in Fig. 3b, the sizes of the first two parts are close to the size of a tile, while the third part is relatively small. This is an important feature as the whitespace in $T3$ can be used for mapping a small CC. Alternatively, it can also be used to map a part of another large CC (indicated by $P1'$ and $P2'$ in Fig. 3b). On the contrary, the whitespace in Fig. 3a is only enough for fitting tiny CCs, which are rare in automata benchmarks. We prefer the unbalanced style during partitioning since it provides more optimizing opportunities.

B. Mapping Method

This section focuses on the configuration of the routing matrix, i.e., the global and local switches. The configuration of STEs, i.e., their associated input symbols and whether they are the start or final states, is generated by VAsim. APmap simply copies this information to the final configuration file.

The transitions among the states in different tiles are mapped to multiple global and local switches in two steps: first to configure global switches and then the local switch part. Fig. 4 uses an example to illustrate these steps. Assume the four states in the automaton are partitioned into three parts, i.e., $\{S2\}$, $\{S2$ and $S4\}$, and $\{S3\}$, and these parts will be mapped to Tile 1, 2, and 3, respectively. For both global and local switches, the input signals connect to the rows, and the columns generate the output signal. Dots indicate that the column is connected to the row, i.e., when the row is activated, the column also activates. First, APmap selects a global switch, e.g., Global switch 1, that has at least one *free* wire with Tile 1. Here, free means that it has not been assigned for mapping other transitions. Next, APmap checks the connections between Global switch 1 and Local switch 2 and 3. Similarly, it requires at least one free wire. If the above requirements are all satisfied, then the global switch part is successfully mapped. $S1$ will be placed at the slot that connects Global switch 1. Note that only 24 slots can output its signal to global switches, and these slots are illustrated by the blue region. Similarly, only 24 slots receive signals from the global switches, and they are colored yellow. All the wires assigned in this step are colored red in Fig. 4. If any condition is not satisfied, then APmap selects the next global switch and checks again. If none of the global switches meet the requirement, the mapping process fails.

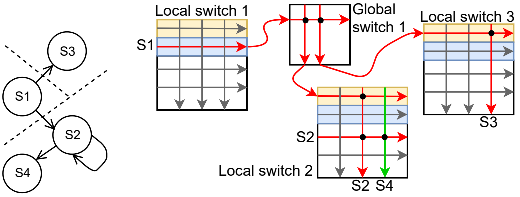


Fig. 4. Mapping an example automaton to AP. The automaton is partitioned into three parts, which are mapped to Tile 1 to 3, respectively. The input signal of the global switches origin from the blue region of the local switches while the output signals enter the yellow region of the local switches. The black dots indicated that the row can activate the corresponding column.

After all the transitions being mapped to global switches, detailed mapping to local switches consists of two parts. The first part is to map the transitions within a tile, e.g., two dots are placed in the row of $S2$ in Fig. 4, implementing the transitions from $S2$ to $S2$ and $S4$. The second part is to map the signals that come from global switches, i.e., configuring the dots on the red region in Local switch 2 and 3 in Fig. 4. They can be conducted similarly.

C. Meeting Constraints

Constraint violation is a result of hardware resource limits and unsatisfactory partitioning. When a large CC is partitioned into several parts, the number of transitions among these parts is unconstrained. Although METIS tries to minimize the total transition numbers, it is possible that the transition number exceeds the number of wires that connect a tile with global switches. In this case, we need to resolve this conflict before mapping it to the tile.

In this section, we first present the methods for resolving output constraint conflicts and then the input. As introduced in Section II-B, a tile has only 24 output wires that connect to global switches. Any state that transits to states in other tiles, referred to as an *outgoing* state, has to be mapped in those 24 slots. Assume that a partitioned part contains more outgoing states than the constraint, including two states, $S2$ and $S3$. We duplicate this part and keep only half of them as outgoing states in each copy. Fig. 6 shows a possible mapping result where these two parts are mapped to Tile 2 and 2', respectively. $S2$ is regarded as an outgoing state in Local switch 2 but not in Local switch 2'. $S3$ is the opposite. In this

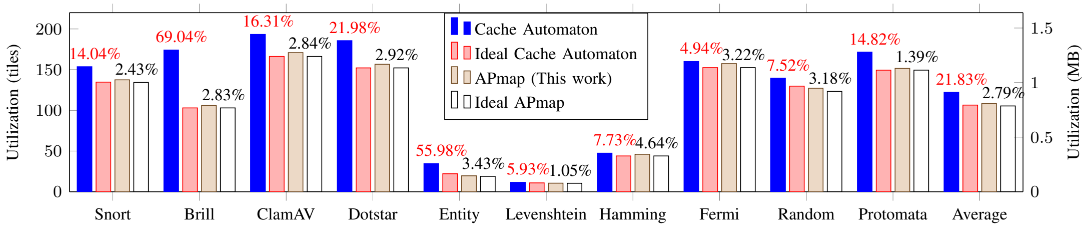


Fig. 5. Utilization comparison between APmap and Cache Automaton. The numbers above the bars illustrate the percentages that the actual utilization exceeds the ideal one.

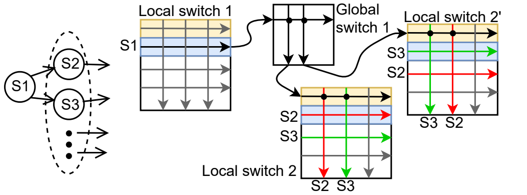


Fig. 6. Example of resolving an output constraint conflict. Assume a part contains more outgoing states than the constraint. To resolve this conflict, Local switch 2, including input signals (e.g., coming from $S1$), is duplicated as Local switch 2'. The outgoing states (e.g., $S2$ and $S3$) are split between these two tiles.

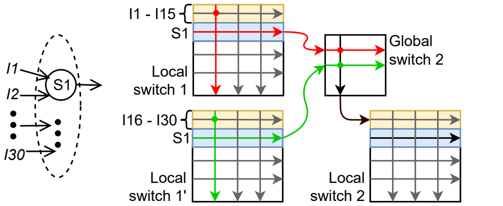


Fig. 7. Example of resolving an input constraint conflict. Assume a part requires 30 incoming signals (i.e., $I1$ to $I30$), which exceeds the constraint. To solve this conflict, Local switch 1 is duplicated as Local switch 1'. The incoming signals are split between these two switches. The outgoing states (e.g., $S1$) in these two switches activate other states together.

way, they can both activate other states. Note that all the input signals are also duplicated. Assume that $S1$, which is mapped to Tile 1, activates $S2$ and $S3$. After the duplication, the global switch maps the output of $S1$ to both Local switch 2 and 2'. Therefore, the execution process of the CC is unchanged.

APmap resolves input constraint conflicts by duplicating as well. However, comparing with output constraint resolving, one additional configuration is required. Assume a part contains 30 input signals named from $I1$ to $I30$, which exceeds the 24-input constraint. It is duplicated and assigned to Tile 1 and 1', respectively. The 30 inputs are also divided into two groups and assigned to those tiles. For the outgoing states in this part (e.g., $S1$), the duplicates activate the states in other parts together as shown in Fig. 7. This configuration guarantees the correctness of automata execution. Assume $S1$ can be activated by (one of) the input signals $I1$ to $I30$. After the duplication, the $S1$ in either Local switch 2 or 3 is (or both of them are) activated, and it (or they) will further activate other states through the red or the green paths in the figure.

In general, if a part contains N outgoing states or incoming signals, it will be duplicated $\lceil \frac{N}{24} \rceil - 1$ times, and these states or signals are distributed equally in these duplicates. If a part has both output and input constraint conflicts, it is duplicated for resolving the output constraint conflict first, and then the input constraint conflict in each duplicate is resolved individually.

IV. Evaluation

A. Evaluation Methodology

We adopt ANMLzoo as the benchmark suite in the evaluation since it is widely used for evaluating APs, especially Cache Automaton [5]. Two benchmarks in this suite, i.e., BlockRings and CoreRings, contain large CCs that exceed the capacity of an RRAM-AP or Cache Automaton chip. Therefore, they are excluded from this evaluation.

We use the latest commit of VASim to parse and optimize ANMLzoo benchmarks. Only the optimized automata will be used in the evaluation because their CC sizes are larger and hence more challenging for mapping tools. The state numbers after optimization are slightly different from that in Cache Automaton's paper [5], probably because of the usage of different VASim versions. The two notable exceptions are PowerEN and SPM. The state numbers are so different in the two works that they are not representing the same benchmark. Therefore, we will exclude them from the evaluation.

In the first experiment, we evaluate the performance of APmap. The hardware target for the mapping is two AP chips with a full routing matrix. No wires are connecting these chips. In the second experiment, we explore the hardware design space by removing the 4-way global switch and altering the number of 1-way global switches.

B. Experimental Results

When mapping the benchmarks to the full routing matrix, the utilization of the results is shown in Fig. 5. The utilization of Cache Automaton's mapping tool [5] and the *ideal* utilization for these two cases are also shown as a comparison. The results in [5] are reported using MBs (see the right Y-axis) and it is interchangeable with the number of tiles. A tile contains 256 STEs whose size is 32 bytes each, and hence a tile occupies 8 kB. Ideal utilization is referred to the minimum amount of memory required for mapping an automaton in theory, i.e., the product of the state number and the STE size. The ideal utilization may never be achieved due to the input and output constraints and the imperfect partitioning. However, it can be used to measure the ability of the mapping tools. The *overhead*, the percentage by which the actual utilization exceeds the ideal, of both tools is illustrated above the bars.

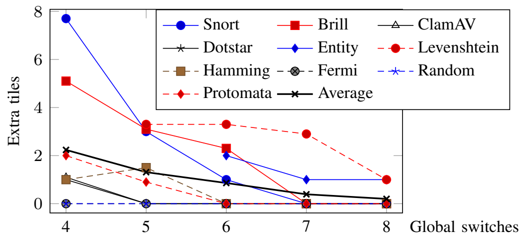


Fig. 8. Extra tiles needed when mapping the benchmarks to routing matrix with less global switches. The comparison baseline is the case with the full routing matrix, i.e., Fig. 5.

As the overhead is calculated using separate bases, it is a fair comparison for these two tools. For all the benchmarks, APmap achieves a lower overhead than Cache Automaton. In addition, APmap's overhead is always below 5%, with an average of 2.79%. On the Cache Automaton side, however, the average overhead is 21.83%, and the highest is more than 50% (Brill and Entity). No tiles are duplicated in this evaluation as all the input/output constraints are satisfied.

Fig. 8 illustrates the extra tiles used in routing matrix design space exploration. The comparison baseline is the utilization of mapping the benchmarks to the full routing matrix, i.e., the result shown in Fig. 5. The values plotted in Fig. 8 is the difference between the actual utilization and the baseline. These differences are mainly introduced in the tile duplication process of input/output constraint resolution. The trends showed that with less global switches, more input/output constraints are violated. As a result, more tiles are duplicated, which leads to a larger utilization overhead. Note that in some cases, e.g., mapping Levenshtein to 4 global switches, APmap failed, and hence no value is plotted in the corresponding columns.

V. Discussion

This section highlights two main advantages brought with APmap: allowing more applications deployed in a chip and assisting hardware design space exploration.

Deploy more applications. Imagine an application is mapped to Tile 1 and 2, which are connected by Global switch 1. Now, we map another application to Tile 3 and 4, which are connected by Global switch 1 as well. As indicated by different colors, the transitions in an application do not affect the other. Note the connections between Global switch 1 and Local switch 3 or 4 exist before the mapping of the second application. Therefore, the new mapping does not occupy any resources that have been used in the previous mapping. As proof, APmap successfully mapped Snort and Brill together to two AP chips, with the utilization of 243.4 tiles (95.1% of all the tiles in two chips). This is impossible for Cache Automaton as the sum of their reported utilization exceeds the capacity of two chips.

Assist hardware design. APmap provides methodologies to solve hardware constraints on the number of input and output signals. As shown in Fig. 8, APmap can map all the benchmarks to a routing matrix with only six 1-way global switches and no 4-way global switch. It allows the hardware to be more compact, and hence achieving better performance. Especially, the 4-way global switch is much slower than the 1-way global switch and STEs [5], which affects the throughput of the whole chip [7]. Therefore, APmap can become a crucial member in a hardware/software co-design toolchain.

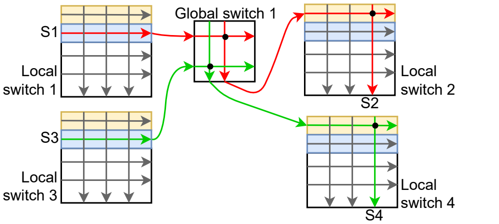


Fig. 9. Example of two applications sharing a global switch. There is no interference between the two groups of paths as indicated by the red and green colors.

VI. Conclusion

In this article, we proposed an open-source tool named APmap for mapping automata to AP chips. It employs multiple optimizations to automate the mapping process and decrease storage utilization. An evaluation with ANMLzoo benchmark suite shows that APmap achieves low overhead and significantly outperforms state of the art.

References

[1] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration*, ser. LISA '99. USENIX Association, 1999, pp. 229–238.

[2] I. Roy *et al.*, "High performance pattern matching using the automata processor," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 1123–1132.

[3] K. Zhou *et al.*, "Brill tagging on the micron automata processor," in *International Conference on Semantic Computing*, 2015, pp. 236–239.

[4] P. Dlugosch *et al.*, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 3088–3098, Dec 2014.

[5] A. Subramaniyan *et al.*, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, Oct. 2017, pp. 259–272.

[6] J. Yu *et al.*, "Memristive devices for computation-in-memory," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, March 2018, pp. 1646–1651.

[7] ——, "Time-division multiplexing automata processor," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019.

[8] K. Angstadt *et al.*, "Rapid programming of pattern-recognition processors," ser. ASPLOS'16. ACM, 2016, pp. 593–605.

[9] J. Wadden *et al.*, "Automata-to-routing: An open-source toolchain for design-space exploration of spatial automata processing architectures," in *FCCM'17*, April 2017.

[10] T. Xie *et al.*, "Reapr: Reconfigurable engine for automata processing," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8.

[11] R. Rahimi *et al.*, "Grapefruit: An open-source, full-stack, and customizable automata processing on fpgas," in *FCCM'20*, 2020, pp. 138–147.

[12] H. B. Noyes, "Microns automata processor architecture: Reconfigurable and massively parallel automata processing," in *Proc. of Fifth Int'l Symp. on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2014.

[13] J. Wadden *et al.*, "VASim: An open virtual automata simulator for automata processing application and architecture research," Technical Report CS2016-03, University of Virginia, Tech. Rep., 2016.

[14] G. Karypis *et al.*, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1998.

Curriculum Vitæ

Jintao Yu was born on August 20, 1987, in Kedong, Heilongjiang, China. He obtained a BSc. degree in Mechanical Engineering and Automation from Tsinghua University with an Excellent Graduate award in 2010. Thereafter, he received an MSc. degree in Computer Science and Technology from PLA Information Engineering University in 2013. His master's research was on high-level synthesis for FPGAs. Then, in 2015, He joined the Computer Engineering group at the Faculty of Electrical Engineering, Mathematics, Computer Science at Delft University of Technology to pursue the Ph.D. degree under the supervision of Prof. dr. ir. Said Hamdioui. His research interests include resistive computing, automata processing, and domain-specific languages.

List of Publications

International Journals

3. **J. Yu**, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, S. Hamdioui, *APmap: An Open-source Compiler for Cache Automaton*, submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), undergoing a minor reversion.

2. **J. Yu**, R. Nane, I. Ashraf, M. Taouil, S. Hamdioui, H. Corporaal, K. L. M. Bertels, *Skeleton-based Synthesis Flow for Computation-In-Memory Architectures*, IEEE Transactions on Emerging Topics in Computing (TETC), Volume 8, Issue 2, 2020, pp. 545-558.

1. H. A. Du Nguyen, **J. Yu**, M. Abu Lebdeh, M. Taouil, S. Hamdioui, F. Catthoor, *A classification of Memory-centric Computing*, ACM Journal on Emerging Technologies in Computing (JETC), Volume 16, Issue 2, 2020.

International Symposiums and Conferences

12. **J. Yu**, M. Abu Lebdeh, H. A. Du Nguyen, M. Taouil, S. Hamdioui, *The Power of Computation-In-Memory Based on Emerging NVM*, The 25th Asia and South Pacific Design Automation Conference (ASP-DAC'20), Beijing, China, January 2020, pp. 1-8.

11. **J. Yu**, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, S. Hamdioui, *Enhanced Scouting Logic: A Robust Memristive Logic Design Scheme*, The 15th IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'19), Qingdao, China, July 2019, pp. 1-6.

10. **J. Yu**, H. A. Du Nguyen, M. Abu Lebdeh, M. Taouil, S. Hamdioui, *Time-division Multiplexing Automata Processor*, The 22nd Design, Automation & Test in Europe Conference & Exhibition (DATE'19), Florence, Italy, March 2019, pp. 794-799.

9. **J. Yu**, H. A. Du Nguyen, L. Xie, M. Taouil, S. Hamdioui, *Memristive Devices for Computation-in-memory*, The 21st Design, Automation & Test in Europe Conference & Exhibition (DATE'18), March 2018, pp. 1646-1651.

8. **J. Yu**, R. Nane, A. Haron, S. Hamdioui, H. Corporaal, K. L. M. Bertels, *Skeleton-based Design and Simulation Flow for Computation-in-Memory Architectures*, The 12th IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'16), Beijing, China, July 2016, pp. 165-170. (**Best Student Paper Award**)

7. H. A. Du Nguyen, **J. Yu**, M. Abu Lebdeh, M. Taouil, S. Hamdioui, *A Computation-In-Memory Accelerator Based on Resistive Devices*, The 5th International Symposium on Memory Systems (MEMSYS'19), Washington DC, USA, September 2019, pp. 1-14 (to appear).

6. H. A. Du Nguyen, **J. Yu**, L. Xie, M. Taouil, S. Hamdioui, D. Fey, *Memristive Devices for Computing: Beyond CMOS and Beyond von Neumann*, The 25th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'17), Abu Dhabi, UAE, October 2017, pp. 1-10.

5. A. Haron, **J. Yu**, R. Nane, M. Taouil, S. Hamdioui, K. L. M. Bertels, *Parallel Matrix Multiplication on Memristor-based Computation-in-Memory Architecture*, The 14th International Conference on High Performance Computing & Simulation (HPCS'16), Innsbruck, Austria, July 2016, pp. 759-766.

4. L. Xie, H. A. Du Nguyen, **J. Yu**, A. Kaichouhi, M. Taouil, M. Alfailakawi, S. Hamdioui, *Scouting Logic: A Novel Memristor-based Logic Design for Resistive Computing*, IEEE Computer Society Annual Symposium on VLSI (ISVLSI'17), Bochum, Germany, July 2017, pp. 151-156.

3. H. A. Du Nguyen, L. Xie, **J. Yu**, M. Taouil, S. Hamdioui, K.L.M. Bertels, *Interconnect Networks for Resistive Computing Architectures*, The 12th IEEE International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS'17), Palma de Mallorca, Spain, April 2017, pp. 1-6.

2. L. Xie, H. A. Du Nguyen, **J. Yu**, M. Taouil, S. Hamdioui, *On the Robustness of Memristor Based Logic Gates*, The 20th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'17), Dresden, Germany, April 2017, pp. 158-163.

1. A. Banagozar, K. Vadivel, S. Stuijk, H. Corporaal, S. Wong, M. Abu Lebdeh, **J. Yu**, S. Hamdioui, *CIM-SIM: Computation In Memory SIMuIator*, The 22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES'19), St. Goar, Germany, May 2019, pp. 1-4.

Workshops & Posters

5. **J. Yu**, L. Xie, H. A. Du Nguyen, M. Taouil, S. Hamdioui, *Memristor-based Automata Processor*, The 5th Workshop on Memristor Technology, Design, Automation and Computing (MDAC'18) in conjunction with HiPEAC, Manchester, United Kingdom, January 2018.

4. **J. Yu**, R. Nane, S. Hamdioui, K. L. M. Bertels, *Data Patterns for Skeleton-base Programming Flows*, The 4th Workshop on Memristor Technology, Design, Automation and Computing (MDAC'17) in conjunction with HiPEAC, Stockholm, Sweden, January 2017.

3. **J. Yu**, H. A. Du Nguyen, L. Xie, M. Taouil, R. Nane, S. Hamdioui, K.L.M. Bertels, *Memristor-based Computation-in-Memory Synthesis Framework*, ICT.OPEN, Amersfoort, Netherlands, March 2017.

2. **J. Yu**, R. Nane, A. Haron, S. Hamdioui, H. Corporaal, K. L. M. Bertels, *Hardware Reuse for Skeleton-based Implementation of Computation-In-Memory Architecture*, The 1st International Workshop on In-Memory and In-Storage Computing with Emerging Technologies (IMISCET'16) in conjunction with PACT, Haifa Israel, September 2016.

1. L. Xie, H. A. Du Nguyen, **J. Yu**, M. Taouil, S. Hamdioui, *FPGA Implementations Based on Memristor Logic Circuits*, ICT.OPEN, Amersfoort, Netherlands, March 2017.