TU Delft

Delft University of Technology

Document Version
Accepted author manuscript

Citation (APA)
Yu, J., Du Nguyen, H. A., Abu Lebdeh, M., Taouil, M., & Hamdioui, S. (2019). Time-division Multiplexing Automata Processor. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE) : Proceedings* (pp. 794-799). Article 8715140 (Proceedings of the 2019 Design, Automation and Test in Europe Conference and Exhibition, DATE 2019). IEEE. https://doi.org/10.23919/DATE.2019.8715140

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

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.

Key words— time-devision multiplexing, automata, parallel processing

1 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:

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

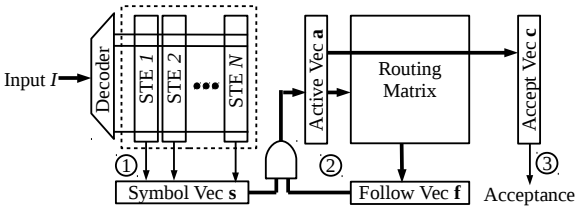


Figure 1: General architecture of Automata Processors [17].

- 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 2, we explain the basic principle of a popular automata accelerator type. Section 3 presents the TDM methodology and how it can be integrated in an RRAM-based automata accelerator. Section 4 evaluates the performance and the area overhead of the TDM RRAM-based automata accelerator. Section 5 contains a brief discussion. Finally, 6 concludes the paper.

2 Background

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

2.1 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.

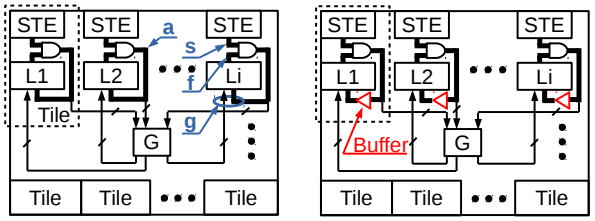
The generalized architecture of APs is shown in Fig. 1 [17]. An input symbol I is processed using three major steps:

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 **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 **a**) based on the transition function (stored in the routing matrix), and stores the result in the Follow Vector **f**; (2) the next active states (i.e., Active Vector) by bit-wise ANDing **s** and **f**.

③ **Output identification.** In order to decide whether the input sequence is accepted or not, the intersection of **a** and pre-configured Accept Vector **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.



(a) Original (b) Adapted for TDM

Figure 2: Adapting the hierarchical switching network for TDM.

2.2 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 2.1) 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.

3 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.

3.1 Methodology

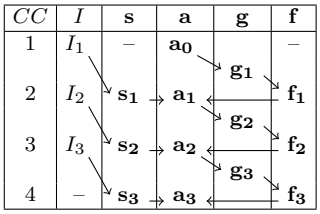
In this section, we first examine the data flow of an AP without pipelining and then apply TDM on it. We use Fig. 3a to explain Cache Automaton's working principle. Each row of the table represents a clock cycle (CC), while the columns contain the values of the input symbol I and key vectors introduced in Section 2 (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 $\mathbf{a_3}$.

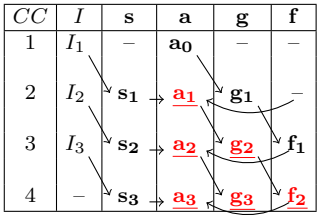
We can solve this problem by decreasing the input frequency as shown by Fig. 3c. If an input symbol is processed every two cycles instead of every cycle, then it can match the speed of the switching network. The dependency among all the values are the same as Fig. 3a. Therefore, the results are correct. 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.

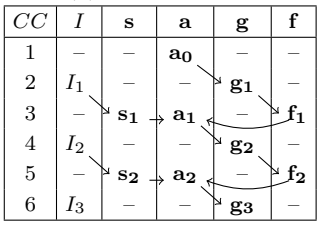
Fig. 3 clearly shows that the TDM methodology may improve the performance. Both Fig. 3a and Fig. 3d process one symbol every cycle; nevertheless, the clock frequency of the implementation in Fig. 3d can be much higher. The length of the clock period for Fig. 3d equals the latency of a single switching 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 ex-



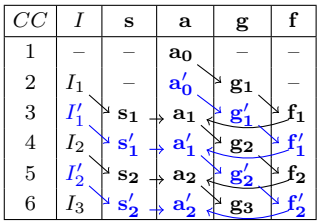
(a) No pipline



(b) Naive pipeline



(c) Slow pipeline



(d) TDM pipeline

Figure 3: Pipelining of global and local switches for APs.

ample, 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).

3.2 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

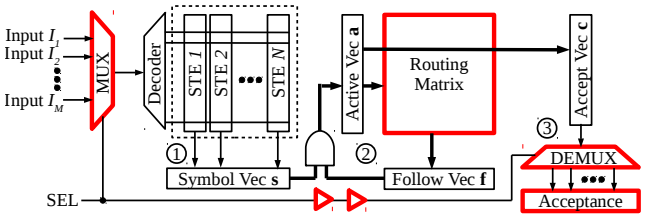


Figure 4: Architectural modifications required to support TDM in AP.

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 **a** and the local switches in order to balance all paths between Active Vector **a** and Follow Vector **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 **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.

3.3 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

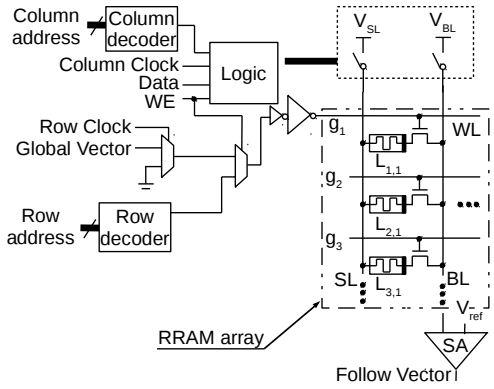


Figure 5: Local switch implementation.

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 **g** and the array's configuration **L**. During a read operation, the bit lines are first precharged to a high voltage. Subsequently, **g** is applied to the word lines; note that multiple word lines can be activated simultaneously. Each column computes the inner product of **g** and a column vector of **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 1. 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 3.1).

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.

Table 1: 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

4 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 2.1 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.

4.1 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 nm and 80 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 nm technology. To simplify and speed up the simulation, only one complete row and column of the STE arrays, global, and local switches are simulated. In

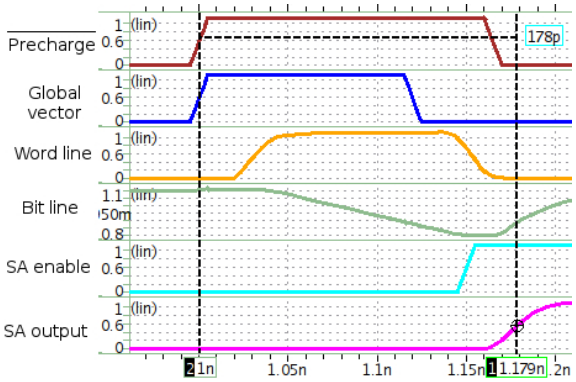


Figure 6: SPICE simulation result of the local switch.

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 µm and 1.5 mm, respectively, with a latency of 66 ps/mm. Therefore, the latency introduced by the global wire is 99 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 1 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.

4.2 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.

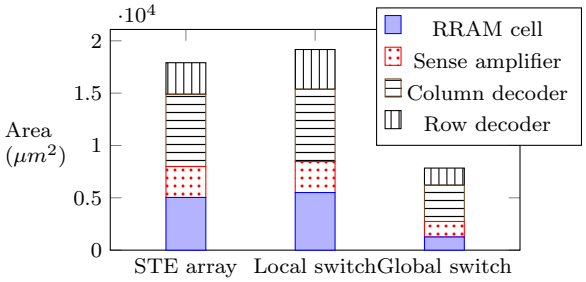


Figure 7: Area breakdown of STE array and switches.

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:

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×.

4.3 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:

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

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 2. The

Table 2: Component Area of TDM RRAM-AP

Component	Array size	Area (μm^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 %

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.

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 μm^2 considering 25 % routing overhead.

5 Discussion

5.1 TDM Methodology

Introducing TDM to APs increases their throughput significantly. The RRAM-AP design presented in Section 4.2 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 3. 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].

[1] In Cache Automaton, four STEs share an SA to save area. Therefore, the *input symbol matching* step (Step ① in Section 2.1) 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

Table 3: Comparison Between TDM RRAM-AP and The State-of-the-Art

Designs	Freq. (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

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.

5.2 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].

6 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] M. Roesch, "Snort - lightweight intrusion detection for networks," in *LISA '99*, (Berkeley, CA, USA), pp. 229–238, USENIX Association, 1999.

[2] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru, "High performance pattern matching using the automata processor," in *IPDPS'16*, pp. 1123–1132, IEEE, May 2016.

[3] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the micron automata processor," in *International Conference on Semantic Computing*, pp. 236–239, 2015.

[4] T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins, "Nondeterministic finite automata in hardware-the case of the levenshtein automaton," *ASBD'15*, 2015.

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

[6] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the automata processor," in *High Performance Computing*, pp. 200–218, Springer, 2016.

[7] M. H. Wang, G. Cancelo, C. Green, D. Guo, K. Wang, and T. Zmuda, "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] A. Subramaniyan and R. Das, "Parallel automata processor," in *ISCA '17*, (New York, NY, USA), pp. 600–612, ACM, 2017.

[9] H. Wang, S. Pu, G. Knezek, and J. Liu, "Min-max: A counter-based algorithm for regular expression matching," *TPDS*, vol. 24, pp. 92–103, Jan 2013.

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

[11] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *TPDS*, vol. 25, no. 12, pp. 3088–3098, 2014.

[12] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: The unified automata processor," in *MICRO-48 '15*, pp. 533–545, Dec 2015.

[13] J. Wadden, V. Dang, N. Brunelle, T. T. 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 *IISWC*, 2016.

[14] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch, "Hawk: Hardware support for unstructured log processing," in *ICDE'16*, pp. 469–480, IEEE, May 2016.

[15] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *MICRO-49 '16*, (New York, NY, USA), pp. 1–12, ACM, Oct 2016.

[16] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *MICRO-50 '17*, (New York, NY, USA), pp. 259–272, ACM, Oct 2017.

[17] J. Yu, H. A. D. Nguyen, L. Xie, M. Taouil, and S. Hamdioui, "Memristive devices for computation-in-memory," in *DATE'18*, pp. 1646–1651, IEEE, March 2018.

[18] W. Dong, D. Liu, S. Xu, B. Chen, and Y. Zhao, "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)*, pp. 851–854, Oct 2017.

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

[20] I. Agbo, M. Taouil, S. Hamdioui, S. Cosemans, P. Weckx, P. Raghavan, and F. Catthoor, "Comparative analysis of rd and atomistic trap-based bti models on sram sense amplifier," in *DTIS'15*, pp. 1–6, April 2015.

[21] Cadence Design Systems, "Tools." Accessed: 2018-11-27.

[22] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *TCAD*, vol. 31, July 2012.