eAP: A Scalable and Efficient in-Memory Accelerator for Automata Processing

Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron
University of Virginia
<elaheh,rahimi,vv8dn,mircea,skadron>@virginia.edu

ABSTRACT

Accelerating finite automata processing benefits regular-expression workloads and a wide range of other applications that do not map obviously to regular expressions, including pattern mining, bioinformatics, and machine learning. Existing in-memory automata processing accelerators suffer from inefficient routing architectures. They are either incapable of efficiently place-and-route a highly connected automaton or require an excessive amount of hardware resources.

In this paper, first, we propose a compact, low-overhead, and yet flexible interconnect architecture that efficiently implements routing for next-state activation, and can be applied to the existing in-memory automata processing architectures. Then, we present eAP (embedded Automata Processor), a high-throughput and scalable in-memory automata processing accelerator. Performance benefits of eAP are achieved by (1) exploiting subarray-level parallelism in memory, (2) a compact memory-based interconnect architecture, (3) an optimal pipeline for state matching and state transition, and (4) efficiently mapping to appropriate memory technologies.

Overall, eAP achieves 5.1× and 207× better throughput per unit area compared to Cache Automaton and Micron's Automata Processor, respectively, as well as lower power consumption and better scaling.

CCS CONCEPTS

• **Computer systems organization** → **Multiple instructions, single data**; • **Hardware** → Emerging architectures; • **Theory of computation** → Formal languages and automata theory.

KEYWORDS

processing-in-memory, embedded DRAM, automata processing, reconfigurable computing, interconnect

ACM Reference Format:
Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A Scalable and Efficient in-Memory Accelerator for Automata Processing. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358324

1 INTRODUCTION

Data collection and the need for real-time analytics are rising rapidly [13, 15]. Pattern matching is an important operation used in many big-data applications such as network security, data mining, and genomics. These patterns are often complex, needing to support a variety of inexact matches. One leading methodology for inexact pattern matching is therefore to use regular expressions or equivalent finite automata to identify these complex patterns. The consequent demand for accelerated pattern matching has motivated many recent studies to utilize automata processing hardware accelerators for deep packet inspection [27], natural language processing [34, 49], bioinformatics [7, 33], pattern mining [36, 44, 45], machine learning [8, 40], and even particle physics [46]. Since 2014, more than 2000 papers and patents have been published on automata processing and its applications.

Researchers are increasingly exploiting in-memory accelerators as performance growth in conventional processors is slowing down. Finite automata processing on CPUs and GPUs exhibit irregular memory access patterns (which disable prediction and data forwarding techniques [26]) and unpredictable memory bandwidth [42, 43]. Therefore, von Neumann architectures struggle to meet today's big-data and streaming line-rate pattern processing requirements.

The Micron Automata Processor (AP) [14] and Cache Automaton (CA) [37] propose in-memory hardware accelerators for automata processing. They both allow native execution of non-deterministic finite automata (NFAs), an efficient computational model for regular expressions, by providing a reconfigurable substrate to lay out the rules in hardware. This allows a large number of patterns to be executed in parallel, up to the hardware capacity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-52, October 12–16, 2019, Columbus, OH, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6938-1/19/10...$15.00
https://doi.org/10.1145/3352460.3358324

For NFA processing in memory-centric architecture models, each input requires two processing phases: *state matching*, where the input symbol is decoded and the states whose rules match the input symbol are detected through reading a row of memory, and *state transition*, where successor states are activated by propagating signals through the interconnect. The interconnect design of existing automata processing accelerators are either incapable of efficient place-and-route of a highly-connected automaton or over-provision hardware resources for interconnect, at the expense of resources for state-matching [35]. However, real-world benchmarks are quite large in terms of the number of states, too big to fit in a single hardware unit, and thus, usually need multiple rounds of reconfiguration and re-processing of the data. This incurs significant performance penalties and makes state-matching resources a scarce resource.

The AP repurposes DRAM arrays for the state-matching and proposes a hierarchical interconnect design. Our study on a diverse set of 21 automata benchmarks reveals that congestion in the AP routing matrix cripples efficient state utilization, especially for the difficult-to-route automata. This means that only 13% of the state matching resources are utilized in Levenstein automata, and the remaining 87% cannot be used because there are not enough routing resources left. Moreover, although the density of DRAM memory is high, an AP chip can only store 1.5MB of data (state matching rules), whereas a conventional DRAM of an equal area can store 25MB of data [19, 37]. This implies that a majority of the area is likely spent for the interconnect and hiding DRAM latency.

Recently, Subramaniyan et al. [37] proposed an in-SRAM automata processing accelerator, Cache Automaton (CA), by repurposing last-level cache for the state-matching and using 8T SRAM cells for the interconnect. To address routing congestion in the AP, CA proposes to use a full-crossbar (FCB) topology for the interconnect to support full connectivity in an automaton, meaning there can be an edge between every two states. This implies a full-crossbar of size 256 needs 256^2 switches. This means that more than 50% of the hardware resources in CA are spent for interconnect! However, our study of 21 automata benchmarks reveals that on average, only 0.53% (maximum 1.15%) of the switches are utilized. Therefore, full crossbars are extremely inefficient and costly for automata processing applications. This expensive interconnect has an opportunity cost in terms of using that area for state matching.

To address the interconnect inefficiencies in the existing in-memory automata processing architectures, this paper presents a *reduced-crossbar (RCB)* design, a low-overhead and yet flexible interconnect architecture that efficiently implements state-transition. RCB design is inspired by intrinsic properties of real-world automata connectivity patterns. RCB requires at least 7× fewer switches compared to the FCB design used in CA. This, in turn, reduces the wire length, which results in shorter latency and lower power consumption. In addition, the area efficiency of RCB provides an opportunity to design a denser state matching resources, which can accommodate more states and results in fewer rounds of reconfiguration and re-processing of data.

Across 21 application from ANMLZoo[42] and Regex [5] benchmark suites, 17 of them can entirely map to RCB design and no FCB is required. To provide a general interconnect solution for every connectivity topology, we design a reconfigurable memory array for state-matching, in which blocks can be repurposed as an FCB to provide full connectivity when needed (at the expense of some state capacity). In addition, to support an automaton with a larger number of states, we design global switches that provide inter-block connectivity between RCBs and FCBs blocks.

To efficiently allow many-to-many transitions in an automaton, the underlying memory technology for eAP should be able to support logical OR functionality within memory rows in a subarray. This requires memory cells (a) to provide non-destructive read, and (b) to drive output to a "stable" state (logical OR in this case) when multiple bitcells drive a common bitline. 8T SRAM cells [9] and gain-cell embedded DRAM (GC-eDRAM) [10, 11] are examples of feasible memories to implement eAP. Note that conventional DRAM and Reduced-Latency DRAM [32] cannot be used for this purpose. They have destructive reads and the value of the simultaneously-activated rows cannot be recovered in the write-back phase.

CA design uses 8T SRAM cells. In this paper, we evaluate eAP on both 8T and 2T1D (2 transistors 1 diode) memory cells. The 2T1D cell is a GC-eDRAM designed and fabricated by [48]. 2T1D uses fewer transistors than an 8T cell and thus, incurs lower area overhead, which results in higher state density and therefore better throughput (due to the reduced rounds of reconfiguration and re-streaming of input). The scalability of gain-cells has been studied in FinFET technology [4, 6, 22], which show gain-cells have the potential to scale to smaller technology nodes in FinFETs.

Interestingly, the wired-OR capability of 8T or 2T1D memory arrays can also be utilized for in-situ computation of other important kernels in neural networks and graph processing. For example, recent studies explore the potential of processing binary neural network computations using 8T SRAM cells and its alternatives [3, 29].

In summary, this paper makes the following contributions:

- We propose a compact and low-overhead interconnect architecture that efficiently implements the state transition stage in automata processing.

- We present eAP (embedded Automata Processor), a high-throughput and scalable in-memory automata processing accelerator. Performance benefits of eAP are achieved by (1) exploiting subarray-level parallelism in memory, (2) designing an optimal pipeline for state matching and state transition, (3) our compact interconnect architecture, and (4) choice of memory technology.
- We evaluate eAP on both 8T and 2T1D memory arrays. Overall, eAP_2T1D achieves 1.7×, 5.1×, and 210× better throughput per unit area over eAP_8T, CA, and the AP, respectively, all in 28nm technology.
- We present a new place-and-route algorithm that is 1-2 orders of magnitude faster than the AP compiler, and provide an open-source cycle-accurate automata simulator to perform software optimization on the automata and map them to the proposed architecture.

2 BACKGROUND AND RELATED WORK

The storage cost of NFAs is $O(n)$ (n is the number of nodes in the NFA), while DFAs (deterministic finite automata) sometimes suffer exponential blowup in state count ($O(2^n)$) compared to equivalent NFAs. This is a side effect of the rule that a DFA can only have one active-state at a time, while an NFA can have many concurrently active-states. Hardware accelerators for automata processing are based on NFAs, both to exploit parallel state-matching and transitions, and the benefit of the NFA's more compact representation.

2.1 Non-Deterministic Finite Automata

An NFA is represented by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols, Δ is a transition function, q_0 are initial states, and F is a set of final states. The transition function determines the next states using the currently active states and the input symbol just read. If an input symbol causes the automata to enter into an accept state, the current position of the input is reported.

We use the homogeneous automaton representation in our model (same as ANML in [14]). In a homogeneous automaton, all transitions entering a state must happen on the same input symbol [17]. This provides a nice property that aligns well with a hardware implementation that finds matching states in one clock cycle and allows a label-independent interconnect. Following [14], we call this element that both represents a state and performs input-symbol matching in homogeneous automata a *State Transition Element* (STE).

Figure 1 (a) shows an example on classic NFA and its equivalent homogeneous representation. Both automata in this example accept the language $(A|C)^*(C|T)(G)^+$. The alphabets are $\{A, T, C, G\}$. In the classic representation, the start state is q_0 and accepting state is q_3. In the homogeneous one, we label each STE from STE_0 to STE_3, so starting states are STE_0, $STE1$, and $STE2$, and the accepting state is STE_3. In all the architectures analyzed in this paper, any states can be starting states without incurring any extra overhead.

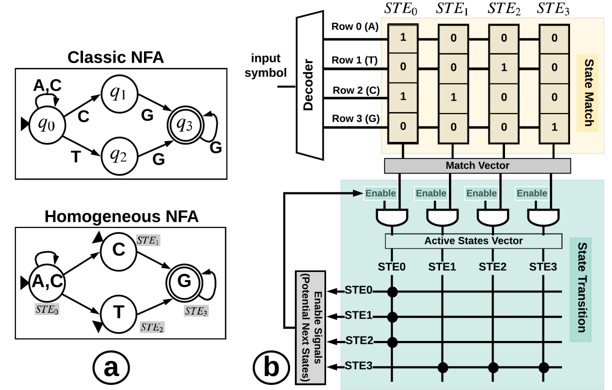

Figure 1: (a) Different NFA representation, (b) A simplified in-memory automata processing model.

2.2 In-memory Automata Processing

The Automata Processor (AP) and Cache Automaton (CA) are two reconfigurable in-memory solutions, both directly implementing NFAs in memory. The following example shows a simplified two-level pipeline of automata processing used in memory-centric models such as the AP and CA.

Working example: In Figure 1 (b), memory columns are configured based on the homogeneous example in Figure 1 (a) for STE_0-STE_3. Generally, automata processing involves two steps for each input symbol, *state match* and *state transition*. In the state match phase, the input symbol is decoded and the set of states whose rule or label matches that input symbol are detected through reading a row of memory (*match vector*). Then, the set of potentially matching states is combined with the *active state vector*, which indicates the set of states that are currently active and allowed to initiate state transitions; i.e., these two vectors are ANDed. In the state-transition phase, the potential next-cycle active states are determined for the currently active states (*active state vector*) by propagating signals through the interconnect to update the active state vector for the next input symbol operation.

In the example, there are four memory rows, and each is mapped to one label (i.e., A, T, C, and G). Each state in the NFA example is mapped to one memory column, with '1' in the rows matching the label(s) assigned to those STEs. STE_0 matching symbols are A and C (Figure 1 (a)), and the corresponding positions have '1', i.e., in the first and third rows (Figure 1 (b)). Assume STE_0 is a current active state. The potential next cycle active states (or enable signals) are the states connected to STE_0, which are STE_0, STE_1, and

STE_2 (the enable signals for STE_0, STE_1, and STE_2 are '1'). Specifically, if the input symbol is 'C,' then Row2 is read into the *match vector*. Bitwise AND on the *match vector* and *potential next states* (enable signal) determines STE_0 and STE_1 as the current active states.

Automata Processor: The AP can process each input symbol in 7.5ns, which is roughly equal to one DRAM row-cycle time. However, conventional DRAM row-cycle latency ($T_{RAS} + T_{RP}$) in this generation is 50ns. This observation shows that the majority of the AP chip area is spent on the routing matrix and on hiding DRAM latency, which we surmise is done by duplicating resources in the critical path.

Cache Automaton: This automata-processing architecture re-purposes a portion of last-level cache (LLC) and proposes an in-cache automata processing accelerator targeting NFAs. The state-matching phase is based on the AP model. The crossbar interconnect uses 8T SRAM memory arrays, and a 2-level hierarchical switch topology with local switches is proposed to provide intra-partition connectivity and global switches providing sparse inter-partition connectivity. CA uses a full-crossbar topology for the interconnect to support full connectivity in an automaton, and as we mentioned earlier, this is excessive for real applications need.

FPGA: REAPR [47] is an FPGA-based implementation of an automata processing engine, and takes advantage of the one-to-one mapping between the spatial distribution of automata states and hardware resources such as LUTs and block RAM. REAPR can achieve 2× to 4× higher clock speeds (250-500 MHz) than the AP, but lower than the estimated clock speed for CA and eAP. Large FPGA chips have approximately 2× more STE capacity than a single AP chip, but 3-6× less capacity than CA when utilizing 10-20MB of LLC and 10-20× less capacity than eAP when utilizing 128MB of 2T1D embedded RAM. Moreover, the power consumption of FPGA-based engines is higher compared to the AP, CA, and eAP. The recent FPGA-based automata processing solutions fail to map complex-to-route automata to the routing resources due to their logical interconnect complexity [24].

2.3 ASIC Implementations

Several ASIC implementations have been proposed [16, 18, 30, 38] to accelerate pattern matching and automata processing. The Unified Automata Processor (UAP) [16] and HARE[18] have demonstrated line-rate automata processing and regular matching expression on network intrusion detection and log processing benchmarks. HARE uses an array of parallel RISC processors to emulate the AHO-Corasick DFA representation of regular expression rule-sets. UAP can support many automata models using state transition packing and multi-stream processing at low area and power costs.

In general, while ASICs provide high line-rates, they are limited by the number of parallel matches, state transitions, and automata shape. HARE implements DFA and has limitations on the regex shape, and also incurs high area and power costs when processing more than 16 patterns. UAP's line-rate drops for large NFAs with many parallel active states.

2.4 The Importance of Capacity

In CA, the authors use the ANMLZoo benchmarks to calculate cache utilization and report 1.2MB of cache usage on average. The automata provided in ANMLZoo benchmark suite [42] represent just a small portion of the actual application (normalized to fill one AP chip). However, real applications are much larger, with many independent automata comprising the various patterns that make up the full application, which requires orders of magnitude more states than reported in Table 1 in [42].

We illustrate this issue using sequential pattern mining (SPM) [45] benchmark, used in ANMLZoo. SPM is an iterative algorithm where in each iteration of the algorithm, a set of sequence candidates (automata) are checked against the input stream. A relatively small but realistic dataset in SPM requires about 300× more state capacity compared to SPM benchmark in ANMLZoo in order to run the whole application. This means that in order to execute one iteration of the algorithm on a parallel automata accelerator such as one AP chip (48K state capacity), we need to reconfigure the hardware 300 times, each with a subset of the overall problem, and each time stream the whole input string. This incurs a large overhead from repeatedly re-streaming the input, as well as reconfiguration time.

To reduce the specialized hardware resources per NFA and increase the total capacity, Liu et al. [28] demonstrated that not all the states in an NFA are enabled during execution (cold states), and thus, do not need to be configured on the AP. Our proposed technique, both the compact interconnect architecture and utilizing 2T1D cells, are complementary to their technique and improves the efficiency of whatever hardware resources are allocated for automata processing.

3 INTERCONNECT ARCHITECTURE

In this section, we first describe a simple implementation of interconnect using memory subarrays (FCB) and then, we present an efficiently compact and reconfigurable interconnect design (RCB) and its feasibility in hardware. Then, we discuss the potential memory switch cells that can be used.

3.1 Reduced Crossbar Interconnect

The interconnect should provide functionality for every STE to wake up all their successors in one step. This process should be done in one cycle since the triggered successors are needed to process the next symbol in the next cycle. This

implies an interconnect that is statically programmed and can ensure that all required paths are routable, non-blocking, and contention-free. More conventional interconnects require many steps to process all the activations for each symbol. For example, buses can carry many bits simultaneously but cannot support a large number of clients. Ring, mesh, and hypercube are multihop, and contention is a problem.

Full Crossbar Interconnect (FCB) is a straightforward interconnect topology for connecting STEs in an automaton, where every state is connected to every other state (including itself) at the cost of $O(N^2)$ (N is the number of states). To model transitions from multiple STEs to one STE, the output should be connected to multiple inputs. This is equal to logical OR of active inputs (e.g., in Figure 1 (a), STE_3 will be a potential next state if either of STE_1, STE_2, or STE_3 are active). Therefore, there is no need for dynamic arbitration. Cache Automaton (CA) uses the FCB interconnect topology for both local and global switches.

NFAs for real-world applications are typically composed of many independent patterns, which manifest as separate *connected components (CCs)* with no transitions between them. Each CC usually has a few hundred states. All the CCs can thus be executed in parallel, independently of each other. Thus, a crossbar switch can be utilized by packing CCs as densely as possible using a greedy approach [37].

However, our study confirms that using a FCB is very inefficient in routing resources [35]. Assume the FCB switch block's size is 256×256. In a greedy approach, CCs are first sorted based on the number of states and then, are assigned to the interconnect resources. Assume there are three CCs of size 100, 100, and 140. Figure 2 shows mapping of CCs to the FCB switch blocks. Switches in gray areas are configured for the corresponding CC. White areas (70% of total area), are unused switches. Within each CC, transitions are sparse, meaning very few switches in the gray areas are used.

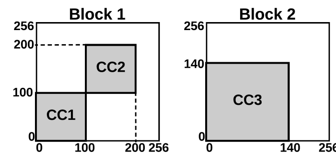

Figure 2: Full-crossbar utilization

We observed that in our 19 real-world and synthetic benchmarks, on average, fewer than 0.48% (maximum 1.1% in Levenshtein) of switch cells (256^2 cells) are utilized in the FCB interconnect solution. This shows that FCB model is extremely inefficient for automata processing applications and forces larger area overhead, power consumption, and delay in the state-transitions phase.

To motivate our efficient and compact interconnect, we visualize the connectivity matrix for the automaton in each benchmark with an image. We first label each node in an automaton with a unique index using breadth-first search (BFS) numeric labeling since BFS assigns adjacent indices to the connected nodes. To draw the image, we model an edge (transition) between two nodes (with indices i and j) in an automaton with a black pixel at coordinate (i,j).

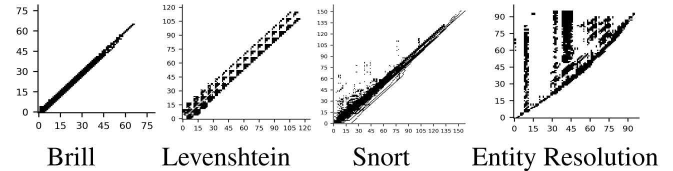

Figure 3: Union heatmap of switches with BFS labeling

In Figure 3, each graph shows the union overall connectivity images for CCs in one benchmark. We chose union to make sure that we have considered every possible transition, even for rare connection patterns. Except Snort and Entity Resolution, the rest of the benchmarks represent a nice diagonal connectivity property. The union and average images for the rest of benchmarks are here[1].

This diagonal connectivity pattern motivates a more compact interconnect, and comes from two properties: first, the power of numeric BFS labeling, which tries to label a child node closely to its parent(s); second, CCs are mostly tree-shape graphs with short cycles and the nodes have a small out-degree. Motivated by these observations, we propose a *reduced* crossbar interconnect (RCB), which has switch patterns similar to what we observed in the union images. RCB have a smaller area overhead, lower power consumption, and smaller delay compared to FCB. Moreover, it can be applied to CA or AP without reducing their computation power.

Feasibility support for RCB Design: To save area via an RCB design, we compact the memory array, preserving input and output signals similar to an FCB, but with fewer switches. This might complicate the layout process because wiring congestion may happen while compacting the array. Automated layout generation tools sometimes are not clever enough to provide the best compacting scheme even for regular patterns like RCB. Therefore, we propose a simple scheme to compact a FCB array to a smaller RCB array.

Simply flipping the diagonal-shape interconnect to a horizontal or vertical side forces the wire congestion in one dimension and it does not utilize the other available dimension to contribute in signal routing. However, squeezing the diagonal-shape to a square shape would significantly compact the subarray and at the same time, spread the burden of signal routing in both dimensions.

[1] github.com/elaheh-sadredini/MICRO52/tree/master/Heatmap

Figure 4 shows a toy example for an FCB subarray of size (9,9) with diagonal width of 3. In each square, the first index shows the row-index and the second one shows the column index. For example, a switch in the location (4,3) shows that the input signal comes from an STE labeled 4 (in BFS), and it is connected to an STE labeled 3. The left block shows the initial naive mapping of diagonal memory cells, while all the white regions are the wasted areas (or switches). The right block shows how moving nearby memory cells close to the lower left side can reduce 9×9 array to 7×6.

Our calculation shows that an FCB of size 256×256 and diagonal width 21 can be reduced to a RCB of size 96×96, which results in approximately 7× switch saving. RCB supports 256 inputs and outputs. Our placement guarantees that each row and column has a maximum of 3 inputs and outputs. For example, in row 4 of RCB, there are two input signals (word-line signal), 2 and 9, and in column-3, there are two output signals (bitline signal), 3 and 6.

From our experiments, we found that the diagonal width of 21 is a safe margin to accommodate all the transitions (except in Entity Resolution and Snort). It should be noted that in the routing subarrays, there is no need for decoding the input because the "active state vectors" (or an array of registers) are directly connected to the wordlines. Therefore, RCB does not incur any extra area overhead for extra decoders. Moreover, RCB has smaller bit-lines due to area compression, which potentially leads to a shorter memory access cycle.

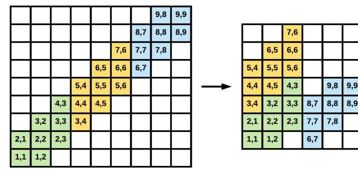

Figure 4: FCB to RCB compression

3.2 Mapping to Memory Technologies

As discussed earlier, to implement the proposed interconnect in memory, the underlying memory technology should be able to support logical-OR functionality among memory rows in a subarray. This requires memory cells (1) to provide non-destructive read (it means data is maintained after read operations and write-back is not necessary), and (2) to drive output to a "stable" state (logical OR in this case) when multiple bitcells drive a common bitline.

Clearly, conventional DRAM and Reduced-latency DRAM (RLDRAM) [32] cannot be adopted, because they have destructive reads and wired-OR destroys the value stored in every node participating in the OR operation. Furthermore, 6T SRAM is not also able to perform wired-OR, because if two cells with different values drive the same bitline, the resulting value would be unstable or undefined. On the other hand, 8T SRAM cells [9] and gain-cell embedded DRAM (GC-eDRAM) [10, 11] appear to be the most suitable memory technologies to implement eAP.

Gain Cell embedded DRAMs (GC-eDRAMs) are comprised of 2-3 logic transistors and optionally an additional MOSCAP or diode [39]. Recent adoption of GC-eDRAMs as on-die caches [10, 11, 31] provides realization for in-eDRAM acceleration of applications. Three-transistor (3T) [11] and two-transistor (2T) [12, 48] GC-eDRAMs are particularly beneficial for providing (1) a fast read-cycle time, and (2) non-destructive read, by splitting read and write paths to the cell. The latter property is especially useful for the interconnect design, where wired-OR functionality is needed.

In this paper, we adopt two memory cell technologies as the reconfigurable switches to evaluate our architecture: (1) 8T SRAM cells, as used in CA [37], and (2) the 2T1D (2 transistors 1 diode) GC-eDRAM cell [48]. Compared to 8T SRAM, 2T1D cell uses substantially fewer transistors and has lower leakage current [10, 12, 23]. Both cell types provide the wired-OR functionality. The scalability of gain-cells has been extensively studied in FinFET technology [4, 6, 22], suggesting that gain cells are promising to scale to smaller technology nodes in FinFETs, and to maintain an area advantage over 8T.

2T1D Switch Cell: The 2T1D DRAM cell holds the connectivity value in the switch, which is '1' if the switch is connected and '0' if it is disconnected. A connected switch implements an existing transition between two STEs in a state machine. Figure 5 shows the details of the 2T1D cell. The cell itself consists of a PMOS transistor for the write operation, an NMOS for the read operation, and an N-Type Gated-Diode for reducing coupling effect.

The cell has two modes: write mode and route mode. As shown in Figure 5, during the write mode, *Write-Worldline* is '1' and the value on the *Write-Bitline* is stored in the node "X". The *Write-Bitline* value controls a switch between STEs to be connected or disconnected. *Write-Bitline* is VDD for the connected switch and GND otherwise. During the route mode, the values that are stored determine whether there is a connection between a source STE (active state) and destination STEs (potential next states).

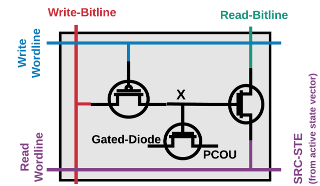

Figure 5: 2T1D switch cell

In the state transition part of Figure 1, vertical wires are *Read-Wordlines* and horizontal wires are *Read-Bitlines*. There is one switch in each cross point and the ones with the black dots show that the switch is connected. If the switch is connected and the source STE is in an active state, then corresponding *Read-Bitlines* activate the potential next states. In more detail, the *Read-Bitlines* are discharged. Therefore, the

sense amplifier connected to the *Read-Bitline* will sense '0' and then is converted to '1' after a NOT gate.

8T Switch Cell: We adopt the switch cell design from CA [37]. The cell consists of a 6T SRAM cell and two additional transistors, which connect the cell to a bitline. This allows a 6T cell to drive the bitlines only when the cell holds '1' and the input signal is '1'. This implies that 8T cells can support OR functionality.

4 EMBEDDED AUTOMATA PROCESSOR

In this section, we explain the design of eAP for one bank. The bank design of eAP_2T1D and eAP_8T is very similar. The banks are replicated to in order to accommodate a large number of automata. The overall capacity of eAP_2T1D and eAP_8T is different and is discussed in Section 7.6.

4.1 eAP Bank Design

Figure 6 shows the general overview of a bank in eAP. Each bank consists of multiple *subarrays* (Figure 6 (a,b)), which share a global decoder, global sense amplifiers, and a set of global bitlines. Each subarray has its own local sense amplifiers and local decoder. Based on subarray-level parallelism (SALP) idea [25], with small changes in the global decoder, we can access to more than one row by reducing the shared resources and enable activation to different subarrays to be done in parallel. Therefore, activation and precharging can be done locally within each subarray. In this paper, we utilize SALP for the state-matching phase in order to match an input symbol with multiple automata in parallel.

In our design, a memory bank supports two modes; *normal mode (NM)*, ie. for data storage as last-level cache, and *automata mode (AM)* (Figure 6 (b)). During the NM, the global decoder only selects one of the connected subarrays based on the input address, and then selects one row within the subarray. During the AM, all the local decoders get the same address (input symbol) from the global decoder and activate the same row in each subarray, in parallel, based on the input symbol. The entire row corresponding to that symbol is read to the sense amplifiers, yielding a vector of all the states accepting the input symbol, i.e. the *Match Vector* in Figure 1 (b). This arrangement is shown in Figure 6 (c) maps to the blue square-blocks. There is no need for column addressing because all the local sense amplifiers (*match vectors*) should be read and propagated to the *state transition* stage. *AM* only requires read operations. The configuration of STEs (memory columns) is done at the context-switch time in normal mode using write operations.

In Figure 6 (c), each bank has eight columns of automata processing arrays (APA) with a maximum capacity of 4096 STEs each. Each APA consists of eight *tile*s and each tile contains two automata processing units (APUs). Each APU hosts a memory subarray of 256×256 for state-matching (blue squares) and a RCB subarray (smaller gray square) with an aggregate size of 256 nodes as local interconnect. Inside each APA, tiles are connected to work collaboratively through a global switch to process larger connected components that do not fit in a single APU. These choice of parameters are based on some prior organizations [14, 37].

The global FCB switch allows 16 states in each APU, called *port nodes* (PNs), to communicate with all PNs of different APUs in the same APA. The global FCB is positioned in the middle of the APA to minimize the longest required wire to/from bottommost and topmost APUs.

For uncommon cases in which a CC does not fit into an RCB interconnect (such as EntityResoloution, see Fig. 3), eAP repurposes state-matching subarrays as FCB interconnects. Specifically, it combines the state-matching subarray of one of the APUs in the tile (as a full crossbar interconnect) and the state-matching of the other APU in the same tile. When a subarray needs to be configured as a FCB instead of regular state-match operation, the FCB/SM signal (Fig. 6.d right blue square) of that tile is set to one. This signal selects the wordlines of the target subarray to be driven by the match vector register bits instead of the decoder output (See Fig. 6.d). This mode halves state capacity of the contributed tile but provides the ability to accept CCs without any limitation on interconnect shape.

To support this functionality, an array of 2:1 multiplexer needs to be added for one of the subarrays in each tile (FCB/SM multiplexers in the right blue square of Fig. 6.d). This has less than 2.5% area overhead based on industry 28 nm 2:1 mux area numbers[2]. This reconfiguration promotes a tile to embed any connected component (with size less than 256) plus having 16 PNs to communicate with other APUs in the same APA to provide more flexible interconnect topology in a column.

4.2 Pipeline Design

To process a single input symbol, two memory accesses are required; one for finding the *match vector* in the state-matching phase, and one for finding the *potential next state vector* in the state-transition phase (see Figure 1 (b)). The result of state-matching of the current symbol is stored in the *Match Vector* registers, which acts as pipeline registers, and can be overlapped with the state transition routing from the previous input symbol matches.

Cache Automaton [37] proposes a three-stage pipeline for automata processing, shown in Figure 7 (a) (SM: State-Match, GS: Global-Switch, LS: Local-Switch). However, we have found that this pipeline has a data-hazard issue. To process input symbol i+1, the result of state-match of the current

[2]This is obtained using a standard cell library provided under NDA, so while we can describe the result, we cannot identify the vendor.

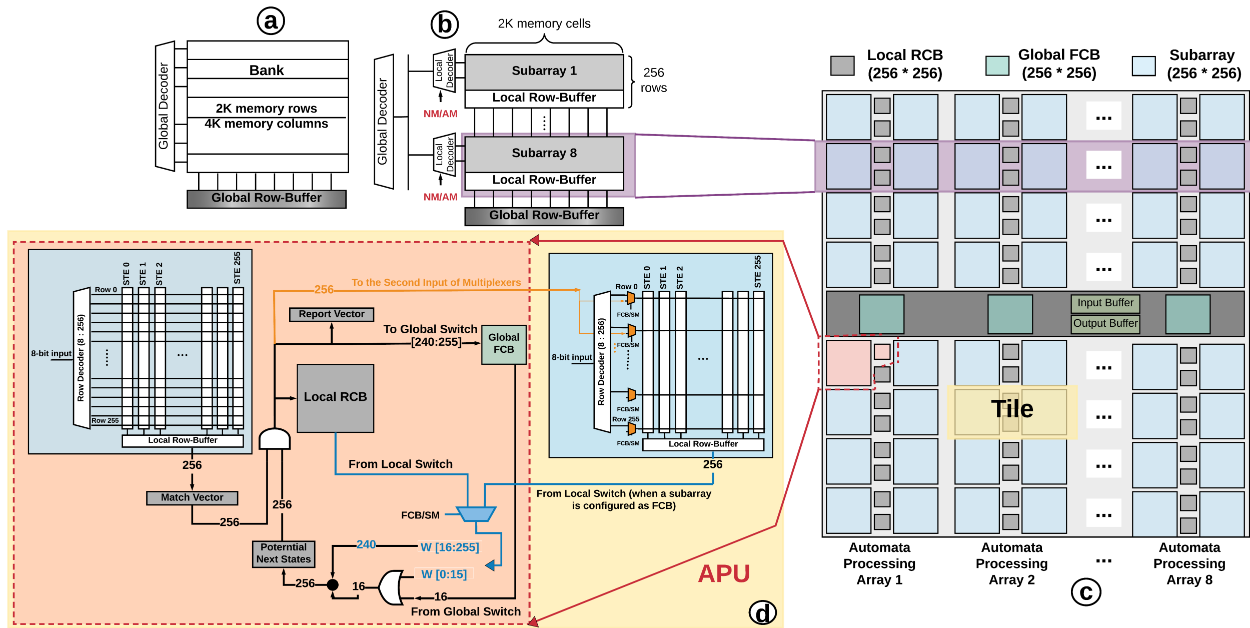

Figure 6: (a) Bank abstraction. (b) Physical implementation of a bank. (c) The general overview of eAP architecture in one bank. (d) Inside one tile with datapath and communication to local (RCB) and global switches (FCB)

cycle (i+1) and state-transition (including LS and GS) of the previous cycle (i) should be ready at the end of stage-2. However, the LS output is only ready at the end of the third stage. To solve this, one pipeline stall is necessary for each input to resolve the hazard, which decreases the throughput by a factor of 2. Another solution (to avoid data hazard) is merging GS and LS in one stage, but they need to operate sequentially (see Figure 1(d) in [37]). This means that stage 1 has one memory access whereas stage 2 has two consecutive memory accesses. Figure 7 (b) shows our refined version of CA pipeline design. This has been verified with the authors.

Unlike CA, our proposed pipeline tries to balance the amount of work between the two stages of the pipeline, since the final frequency is determined based on the slowest stage. Figure 6 (d) represents the interconnect organization. Both global and local switches can operate in parallel in one stage and the result from the global switch is ORed with the corresponding wires from the local switch (Figure 7 (c)). Performing an additional 16-bit OR operation costs much less than one memory access. Similar pipeline optimization (parallel GS and LS) can be applied to CA. Performance results for both designs are shown in Section 7.3.

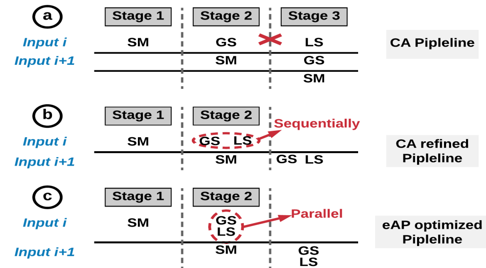

Figure 7: CA (a) vs eAP (b) pipeline

4.3 Input and Output

eAP has two asynchronous FIFOs to hold the input symbols in the input buffer (IB) and reports in the output buffer (OB). The host CPU communicates with the IB and OB using interrupt triggered memory-mapped IO or DMA while the interrupt service routine (ISR) is responsible to fill in the IB and evict the OB. Assuming 1.5 GHz and 2.5 GHz working frequency for eAP_2T1D and eAP_8T, respectively and 1 MHz frequency for interrupt, an IB of size 2.5KB can store enough data to feed the eAP until the next IB interrupt. Recently, Wadden et al. [41] have characterized the reporting statistics of ANMLZoo's benchmark. The results show that 10 out of 12 benchmarks produce less than 0.5 reports per cycle (on average). This investigation motivates us to use 512 entries for the OB (4 bytes each for report meta-data) to keep a similar interrupt rate as the IB.

After writing the automata configuration bits in the normal operation mode (NM), eAP switches to automata mode (AM) and starts consuming inputs from the IB. Buffers have two output signals (E and F) to show if they are full or empty. E-signal of the IB and F-signals can raise the interrupt signal of the CPU to service the device as needed. In Automata mode, in each cycle, the symbol at the front of the IB is popped

and drives the shared address bus of all banks contributing to eAP symbol matching. Each APU is equipped with a *report vector mask* to identify report states in each cycle by simply performing a bitwise AND operation with the *active vector*. We use the Report Aggregator Division (RAD) mechanism proposed in [41] (which is an improvement over Micron's AP reporting procedure) to fill up the OB with report state IDs and cycle information. RAD adaptively shrinks the report message based on the current number of active states to use the OB space efficiently. When the OB is filled up, an interrupt signal is raised to ask for service from the host CPU and free space for future report events.

4.4 System Integration

This section discuss the possible integrations for eAP with 2T1D GC-eDRAM cells (eAP_2T1D) and 8T SRAM cells (eAP_8T). High-bandwidth On-Package Memory (OPM) introduces a new on-package memory layer between off-chip DRAM and on-chip cache in the conventional memory hierarchy. Intel has included eDRAM as an OPM in its Haswell, Broadwell, and Skylake architectures to fill the gap between on-chip and off-chip memory bandwidth. For Haswell and Broadwell processors, eDRAM with 1T1C cells was used as L4 cache [2, 20]. For eAP_2T1D, we replace the 1T1C eDRAM cells with 2T1D and then, repurpose a portion of banks in L4 cache for automata processing.

For eAP_8T, we assume the same integration as Cache Automaton [37]. Cache Automaton repurposes last-level cache (L3) slices for automata processing and access the cache *ways* by leveraging Cache Allocation Technology (CAT) [1]. For both eAP designs, in automata mode, the compiler generates a configuration array (the state-match and interconnect configuration bits) and writes it in the eAP memory address space to start offloading the input task.

5 COMPILER

Our compiler has two main tasks. First, it should check if a connected component can fit into a RCB switch template or needs to be mapped to an FCB. Second, it should provide a mapping from each state of the automaton to its hardware representation (STE). To accomplish the decision problem (RCB or FCB), a fixed matrix representation of the RCB interconnects is initially generated (See Figure 4), called a diagonal matrix (DM). We assign a '1' in row i and column j, if there is a switch at location i and j in RCB interconnect. For any given automaton, we first number nodes using BFS traversal, starting from a fake root connected to all nodes that are start-nodes in the automata. Then, we calculate the connectivity matrix of a given automaton using BFS assigned numbers. If the calculated matrix is a subset of the DM, then it can be fit into a diagonal switch box (RCB). Otherwise, the given automaton should fit into a FCB.

For diagonal automata, we search through all the previously-assigned RCB interconnect blocks and try to find the one with the least free capacity that can still fit the current automaton being placed. We keep the same BFS order of labels to assign inputs of the assigned interconnect block, but with an offset equal to the last-used input of that interconnect block, instead of 1 for the first automaton (connected component) that was assigned to this interconnect block. If there is no such partially used interconnect with enough spare capacity, we initialize a new RCB interconnect block from the pool of available interconnect blocks.

6 EVALUATION METHODOLOGY

Applications: We evaluate the eAP architecture using ANMLZoo [42] and Regex [5] benchmark suites. ANMLZoo represents a set of applications including machine learning, data mining, and security. We use the standard 10MB inputs stream included in ANMLZoo.

Experimental Setup: We evaluate eAP on memory arrays with 2T1D cells (we call it eAP_2T1D) and 8T cells (we call it eAP_8T). In eAP_8T, both state-matching and interconnect memory arrays are based on 8T cells. This is because we sometimes repurpose state-matching arrays for interconnect, and they should be able to provide the required logical OR functionality (6T SRAM cells are unable to provide OR functionality because multiple cells cannot drive one bitline). We compare eAP_2T1D and eAP_8T with CA, and the AP, all using (or scaled to) 28nm technology. In CA, state-matching is based on 6T and interconnect is based on 8T SRAM arrays. To calculate area, power, and row-cycle time of memory arrays, we use a standard memory compiler. For 2T1D analysis, we rely on the results from the fabricated chip in [48].

We develop an in-house cycle-accurate automata simulator[3] to perform software optimization on the automata, map them to eAP architecture, and extract per-cycle statistics for the energy estimation.

7 RESULTS

This section first presents the architectural contributions of our interconnect compared to FCB. Then, area, performance, and power evaluations are presented.

7.1 Interconnect Efficiency

In this section, we first compare the overall architectural benefits of our proposed interconnect design, RCB, over the CA interconnect architecture, FCB. As we presented earlier in Section 3, in CA, the FCB is a memory block of 256×256, while RCB interconnect is a memory block of 96×96, meaning that the RCB design consumes 7.1× fewer switches (or memory

[3] Contact the authors for the source code.

Table 1: Comparison of our interconnect approach (hybrid RCB and FCB) with CA interconnect (FCB only) with both 256×256 and 128×128 subarrays. Our idea requires up to 7.1X fewer switches (memory cells) than CA in 256×256 design and up to 5.6X fewer switches than CA in 128×128 design.

Benchmark	#States	#Transitions	#Connected Components	Largest Connect Component Size	Baseline #FCB (256×256)	Our Idea #FCB (256×256)	Our Idea #RCB (96×96)	Switch Reduction	Baseline #FCB (128×128)	Our Idea #FCB (128×128)	Our Idea #RCB (54×54)	Switch Reduction
Brill	42658	62054	1962	67	168	0	168	7.1X	336	0	336	5.6X
Dotstar	96438	94254	2837	95	378	0	378	7.1X	768	0	768	5.6X
EntityResolution*	95136	219264	1000	96	500	500	0	None	1000	1000	0	None
Fermi	40783	57576	2399	17	160	0	160	7.1X	320	0	320	5.6X
Hamming	11346	19251	93	122	47	0	47	7.1X	96	0	96	5.6X
Levenshtein	2784	9096	24	116	12	0	12	7.1X	24	0	24	5.6X
PowerEN	40513	40271	2857	52	160	0	160	7.1X	320	0	320	5.6X
Protomata	42009	41635	2340	123	165	0	165	7.1X	336	0	336	5.6X
RandomForest	33220	33220	1661	20	139	0	139	7.1X	264	0	264	5.6X
Snort*	100500	81380	5025	222	270	19	252	4.9X	546	21	525	4.7X
SPM	69029	211050	2687	20	419	0	419	7.1X	792	0	792	5.6X
BlockRings**	44352	44352	192	231	192	0	192	7.1X	432	48	384	3.7X
Dotstar03	12144	12264	299	92	49	0	49	7.1X	104	0	104	5.6X
Dotstar06	12640	12939	298	104	50	0	50	7.1X	104	0	104	5.6X
Dotstar09	12431	12907	297	104	50	0	50	7.1X	104	0	104	5.6X
Ranges05	12621	12472	299	94	50	0	50	7.1X	104	0	104	5.6X
Ranges1	12464	12406	297	96	50	0	50	7.1X	104	0	104	5.6X
ExactMath	12439	12144	297	87	50	0	50	7.1X	104	0	104	5.6X
Bro217	2312	2130	187	84	10	0	10	7.1X	24	0	24	5.6X
TCP**	19704	21164	738	391	81	3	80	5.6X	161	15	148	2.8X
ClamAV**	49538	49736	515	542	210	2	208	6.7X	413	13	400	4.9X

* Not all the connected components in EntityResolution and Snort fit in RCB blocks, because their connectivity pattern have long-distance loops. They need to re-purpose state-matching as FCB.
** In TCP, ClamAV, and BlockRing (for 128×128 design), Some connected components are large and do not fit in one state-matching subarray. Therefore, global switches (FCBs) are required to connect transitions between two (or more) local switches.

cells) than FCB. To evaluate whether an FCB of 256×256 is larger than necessary, we also apply our interconnect reduction technique to FCB subarrays of size 128×128, and conclude that the RCB subarray can be reduced to 54×54. This means that when the FCB baseline subarrays are 128×128, RCB requires 5.6× fewer switches than FCB. Compared to the baseline FCB design, RCB has a faster row cycle time because of shorter wires and consumes less power.

To study the applicability of RCB design in real-world and synthetic automata applications, we calculate the number of required RCB and FCB blocks for each application. The compiler iterates over the connected components (CCs) and checks if they can fit in a RCB switch block. If not, a FCB switch is needed to accommodate connectivity. In Table 1, we compare the number of required routing blocks of our ultimate interconnect approach, which is a hybrid of RCB and FCB, versus the baseline FCB that is proposed in CA and assumes full connectivity for all the connected components (but we evaluate FCBs of both 256×256 and 128×128).

As shown in Table 1, most of the connected components of the applications can entirely map to RCB blocks and no FCB block is needed. This means that when using RCB blocks, the total number of switches (memory cells) required for these applications is 7.1× and 5.6× less than when using FCB blocks in 256×256 and 128×128 design, respectively. This again confirms that the FCB is over-provisioned for automata applications, even at 128×128. The largest CC size in most of the applications is less than 128 states and thus, they fit in the 128×128 design. However, all CCs in BlockRings have 231 states, and requires global switches (which are also FCBs) for connecting local switches to provide larger connectivity.

In EntityResolution, there are many long-distance loops, and none of the CCs can fit in the RCB switch block (Figure 3). In Snort, our interconnect accommodates most of the CCs in RCB blocks (only 19 FCB and 252 RCB in 256×256 design), whereas the baseline uses 270 FCBs. Levenshtein is a difficult-to-route automaton. The AP compiler can fit this benchmark in an AP chip with 48K states. However, the total number of states in Levenshtein is 2784. This implies that many of the STEs and interconnect resources of an AP chip are wasted in order to deal with the routing congestion. However, in our interconnect model, we just need 12 RCB switches in 256×256 design (9% routing resources of an eAP bank and 0.07% of routing resources on eAP 128 banks) to accommodate all the automata in Levenshtein.

Our compiler provides optimizations such as forcing constraints on the number of fan-in and fan-out of each node. Based on our sensitivity analysis, forcing each automaton to have maximum fan-in and fan-out of 5 results in the minimum number of switches. Our interconnect optimization is general and can be applied to any memory-based interconnect, such as variations of gain-cells or non-volatile memory, where memory cells can implement OR-functionality for routing.

7.2 Overall Area Overhead

In this section, we discuss the area overhead of state-matching arrays, interconnect arrays, and total overhead for supporting state capacity equivalent to 32K STEs (one eAP bank). Furthermore, we separate architectural contributions from technology contributions in our analysis.

A subarray size of 512 by 128 with 2T1D cell is fabricated in 65nm with area $0.085mm^2$ [48]. From the die image, we estimate the area for a block of 256 by 256 to be $0.084mm^2$ (60% of which is spent for memory cells and 40% is spent for decoder and sense amplifiers), which is $11mm^2$ to support 32K states. The projected area to support 32K states in 28nm is $2mm^2$. Thus, the area of a 2T1D cell in 28nm is estimated $0.143\mu m^2$ and calculated as:

$$\frac{0.6\times 2\times 10^6}{32\times 1024\times 256} = 0.143\mu m^2 \quad (1)$$

Bhoj et al. [6] presented two architectures for 2T1D cells in 30nm FinFET technology. According to their work, the area of a 2T1D memory cell is between $0.137 - 0.163\mu m^2$, which is consistent with our scaling assumptions.

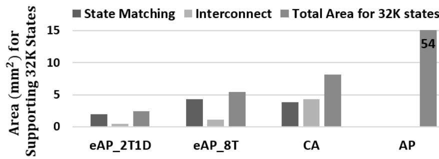

Figure 8: Comparing area overhead of eAP, CA, and AP, normalized for 32K states, all in 28nm. CA interconnect is ~4× larger than eAP_8T (architectural contribution) and ~8× higher than eAP_2T1D.

Figure 8 shows the area overhead for state-matching, interconnect, and total overhead of different architectures, assuming supporting 32K states. Compared to CA, eAP_8T reduces area overhead of interconnect ~4× (resulting from architectural contribution, i.e., RCB design) and eAP_2T1D reduces area overhead of interconnect ~8× (~4× resulting from architectural contribution, i.e., RCB design and ~2× resulting from technology choice).

Overall area overhead (both state-match and routing) of eAP_2T1D is 2.2×, 2.3×, 22× less compared to eAP_8T, CA, and the AP, respectively, all in 28nm technology.

7.3 Overall Performance

Zhang et al. [48] report that the read-cycle frequency of 6T SRAM array is twice that of a 2T1D gain-cell array in 65nm technology. We assume a similar ratio in order to estimate the read-access frequency of a 2T1D array of size 256×256 (for FCB) in 28nm, using the read-access frequency of 6T SRAM array of size 256×256 in 28nm (which is 229 ps and calculated using standard SRAM compiler in nominal voltage 0.8V). In other work on 2T1D, Bhoj et al. [6] presented two architectures for 2T1D cells in 30nm FinFET technology. According to their work, a 2T1D memory array can operate at 2GHz, which is consistent with our assumption. Despite the area reduction in RCB (96×96), we still assume the worst-case delay for RCB to be the same as FCB.

CA proposes a sense-amplifier cycling technique and assumes 4× reduction in the read-access delay. However, sensing is just 25% of the total row-access delay. We re-calculated the delay in local and global switches in CA with best-case assumptions using an SRAM memory compiler. Fixing (1) switch delay calculation and (2) pipeline data-hazard problem in CA reduces the frequency from 2.2GHz to 1.43GHz. This has been verified with the authors.

Based on the SPICE simulation in CA, the wire delay is calculated as $66ps/mm$. Considering a cache slice of $3.19mm\times 3mm$, the switch delay is estimated as 99ps, assuming 1.5mm wire length. We assume the same wire delay for FCB and RCB in eAP. This is the worst-case assumption for RCB as it requires shorter wires.

Table 2 shows the delay for pipeline stages in CA and eAP in 28nm. As discussed in Section 4.2, in the CA-refined pipeline, L-Switch and G-Switch should be done sequentially in one stage, which means the pipeline delay is 698ps (349ps+349ps). In eAP optimized pipeline, L-switch (RCB) and G-switch (FCB) can be done in parallel. Therefore, pipeline delays for eAP_2T1D and eAP_8T are 599ps and 349ps, respectively. Similar optimization proposed for eAP can be applied to CA (we call it CA_opt) which improves CA frequency from 1.43GHz to 2.2GHz. Therefore, the architectural contribution of our optimized pipeline improves the clock frequency of eAP (both 2T1D and 8T) ~2× and CA ~1.5×.

Table 2: Pipeline stages delay. All designs are in 28nm.

Design	State-Match	L-Switch	G-Switch	Freq. Max	Freq. Operated
eAP_2T1D	500 ps	599 ps	599 ps	1.66 GHz	1.5 GHz
eAP_8T	349 ps	349 ps	349 ps	2.8 GHz	2.5 GHz
CA	438 ps	349 ps	349 ps	1.43 GHz	1.3 GHz
CA_opt	438 ps	349 ps	349 ps	2.2 GHz	2 GHz

Like commodity DRAM, 2T1D cells also require periodic refreshes to retain stored bit values. The refresh operation is a sequence of dummy reads and write-backs to the memory rows. eAP_2T1D refresh time is 0.01%, which is calculated by dividing the time required for refreshing 256 rows (meaning 256 reads and 256 writes) by the retention time. Refresh is performed among all the subarrays in parallel and blocks the normal read/write operations.

7.4 Throughput per Unit Area

In the AP, CA, and eAP, each input symbol can be processed in one cycle. Therefore, they have a deterministic throughput of one input symbol per cycle, which is independent of

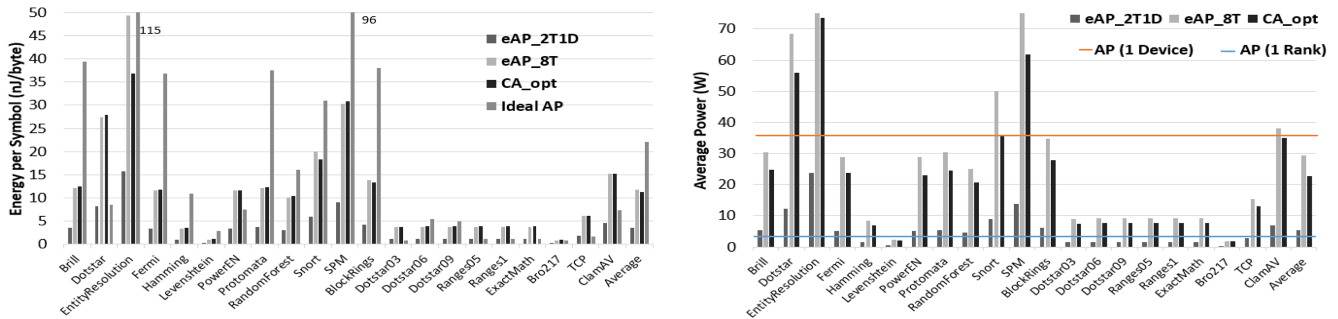

Figure 9: (left) Overall energy consumption of eAP_2T1D compared to eAP_8T, CA_opt, and ideal AP. (right) Overall power consumption of eAP_2T1D compared to eAP_8T, CA_opt, and the AP (reported by Micron).

input benchmarks. Another important metric in addition to frequency is state-matching capacity; if the capacity is not enough to accommodate all the automata in one iteration, several passes of the input stream, each with some reconfiguration overhead, are needed.

Table 3 represents the throughput of different architecture normalized to the area. The throughput here is defined as the number of states that can run in parallel multiplied by clock frequency (Tera-states per second). The AP is based on 45nm and operates at 133 MHz frequency, while CA and eAP are based on 28nm. To compare the different architectures in the same semiconductor technology node, we also show the technology projection of the AP in 28nm.

Table 3: Throughput normalized per area. eAP_2T1D performs best due to its interconnect/technology benefits.

eAP_2T1D	eAP_8T	CA	CA_opt	AP 45nm	AP 28nm
27.1	15.13	5.25	8.07	0.03	0.13

Overall, eAP_2T1D achieves 1.7×, 5.1×, 3.3×, and 210× better throughput-per-unit-area over eAP_8T, CA, CA_opt, and the AP, respectively, all in 28nm technology. As expected, eAP_8T has 1.8× better throughput-per-area over CA_opt. CA design uses 6T arrays of size 256×256 for state-matching and 8T arrays of 280×256 for interconnect, and thus, the interconnect overhead is more than 50%. eAP_8T adopts 8T arrays for both state-matching (of size 256×256) and interconnect of size (96×96), and thus, the interconnect overhead is ~4× less than state-matching resources.

7.5 Energy/Power Consumption

This section discusses the energy and power consumption of eAP_2T1D and eAP_8T, and compares them to prior works. To calculate energy consumption, we need to know the number of active partitions for state-matching and switch blocks, and the number of transitions between local switches to consider for the energy consumed driving wires.

Note that it is not possible to power-gate state-matching memory arrays on a cycle-by-cycle basis. In order to power-gate these subarrays, it is necessary to know the potential next states beforehand. However, in the pipeline, the state-matching results and the next potential state are calculated simultaneously, which prevents the power-gating (one can still power-gate an array that is unoccupied). This observation is not considered in CA. We update the energy/power results in CA paper [37] based on this observation. For the AP, we adopt the *ideal AP* model presented in CA. All the statistics per cycle are extracted from our compiler.

Static power consumption of eAP_2T1D system consists of two main components: (1) the leakage current of the cell itself and (2) the refresh power to keep the data alive. The refresh power of 2T1D-based gain-cells is the dominant portion of static power [48]. Moreover, the static power of 2T1D memory array is 20% of static power in 6T SRAM array. We use the same ratio to calculate static power for eAP_2T1D. We estimate the dynamic energy consumption for RCB and FCB 8T blocks using a standard memory compiler.

Figure 9 (left) shows the energy per input-symbol for eAP_2T1D, eAP_8T, and CA_opt on 28nm, and ideal AP model. We can observe that benchmarks with a larger number of states, such as EntityResolution, Dotstar, Snort, and SPM consume higher energy. This is because these benchmarks have utilized more state-matching and switch arrays to accommodate a larger number of states. Furthermore, EntityResolution cannot utilize lower-energy RCB resources for the local interconnect (as shown is Table 1) and needs to use FCB, which results in higher energy consumption. Overall, the energy consumption of eAP_2T1D is about 3× less than eAP_8T and CA_opt. Energy efficiency of eAP_2T1D comes from its density and a compact RCB design, which results in consuming lower dynamic energy due to shorter wires and a smaller number of switches.

Figure 9 (right) shows the average power consumption across benchmarks. The power consumption of eAP_2T1D

is 5.4× and 4.1× less compared to eAP_8T and CA_opt, respectively. As expected, the power of the eAP_2T1D is the highest, because it has the fastest clock speed.

7.6 Performance Scaling

In this section, we study the scalability of different designs. In order to show the effect of larger benchmarks on the performance, we increase the number of automata in the ANMLZoo up to 1024× and study two power-constrained and non-power-constrained scenarios. We assume CA, CA_opt, and eAP_8T can utilize the 40MB L3 cache [37], which is equal to accommodating 1280K STEs. The Intel 4th-generation Core processor (Haswell and Broadwell) has a 0.5Gb/1Gb embedded memory die connected to CPU as L4 cache [20, 21]. For eAP_2T1D, we assume 1Gb of eDRAM with 128 banks. Therefore, eAP_2T1D can support up to 4096K STEs.

Table 4 summarizes the key properties of eAP_2T1D relative to eAP_8T, CA, CA_opt, and AP. In short, eAP_2T1D has a density advantage compared to other designs. When the area allocation for automata processing is small enough that the total power is not a limiting factor, the density advantage will apply. At some point, enough area is allocated that power becomes a limiting factor. Then eAP only has a capacity advantage.

Table 4: Summary of different memory-based automata architectures (for 32K states, including interconnect)

	eAP_2T1D	eAP_8T	CA	CA_opt	AP
Freq. (GHz)	1.5	2.5	1.3	2	0.133
Power (W)	4.15	29.69	22.57	14.69	2.6
Area (mm^2)	2.47	5.41	8.12	8.12	140

However, some benchmarks in ANMLZoo represent just a portion of actual applications (normalized to fill one AP chip). While one bank is enough for regex-based applications such as Snort, Brill, and Dotstar, which will not require much power, the density advantage will pertain; but other applications require orders of magnitude more states. This will then require multiple passes over the input, with each pass implementing a portion of the overall automata set. In such cases, reconfiguration overheads will apply, and as mentioned, this is more costly for CA.

Figure 10 shows the performance of CA, CA_opt, eAP_2T1D, and eAP_8T averaged on ANMLZoo, normalized to the AP performance, with and without power constraints.

In the non-power-constrained scenario, we assume CA, CA_opt, and eAP can utilize their maximum capacity. In this scenario, the relationship among the designs follows Table 4, except for the additional factor of reconfiguration overhead, so the speedup of eAP_8T is 5×, 3.6×, and 1.4× over CA, CA_opt, eAP_2T1D, respectively. This is because eAP_8T has the highest clock speed.

In the power-constrained scenario, we assume the maximum power of 75W for all the designs. This, in turn, reduces the allowable number of active processing blocks. eAP_8T has 1.6×, 1.1×, and 1.4× better performance over CA, CA_opt, eAP_2T1D on the original-size benchmarks (1X), because eAP_8T has a higher frequency than others. However, when increasing the benchmark size, reconfiguration and multi-processing of the input become a limiting factor for CA and CA_opt (due to less capacity and lower frequency) and eAP_8T (due to high power consumption).

eAP_2T1D shows up to 2×, 2.1×, and 4.9× better performance over CA, CA_opt, eAP_8T when increasing the benchmark-size up to 1024×. This is because eAP_2T1D has higher density and lower power consumption. The performance benefits of eAP increase with larger automata. Furthermore, the advantages of eAP_2T1D over CA, CA_opt, and eAP_8T increase when increasing the input size.

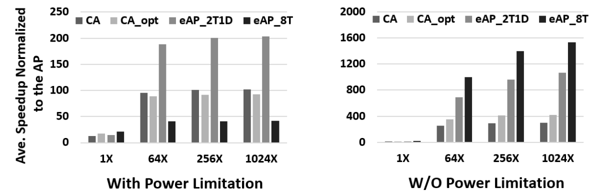

Figure 10: Performance scaling with benchmark size

8 CONCLUSIONS

In this paper, we propose eAP, a high-speed, dense, and low-power reconfigurable architecture for automata processing. We exploit inherent bit-level parallelism in memory to support multiple concurrent transitions in NFA and utilize subarray-level parallelism in memory to process thousands of automata in parallel. Motivated by connectivity patterns in the real-world automata benchmarks, we propose a *reduced* crossbar interconnect for state transitions, which compacts the switch patterns in a full-crossbar interconnect and provides a 7× reduction in the number of switches. This in turn reduces power consumption and delay due to shorter wires. Overall, eAP presents 5.1× and 207× better throughput normalized to area compared to the previously designed in-memory automata accelerators, Cache Automaton (CA) and the Automata Processor, respectively. Benefits of eAP are even higher for applications that require multiple passes.

9 ACKNOWLEDGMENTS

We thank the anonymous reviewers whose comments helped improve and clarify this manuscript. This work is funded, in part, by the NSF (CCF-1629450) and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA.

REFERENCES

[1] Intel. 2017. [n. d.]. Cache Allocation Technology. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[2] Aditya Agrawal. 2014. *REFRESH REDUCTION IN DYNAMIC MEMORIES*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[3] Amogh Agrawal, Akhilesh Jaiswal, Bing Han, Gopalakrishnan Srinivasan, and Kaushik Roy. 2018. Xcel-RAM: Accelerating Binary Neural Networks in High-Throughput SRAM Compute Arrays. *arXiv preprint arXiv:1807.00343* (2018).

[4] E Amat, A Calomarde, F Moll, R Canal, and A Rubio. 2016. Feasibility of Embedded DRAM Cells on FinFET Technology. *IEEE Trans. Comput.* 65, 4 (2016), 1068–1074.

[5] Michela Becchi, Mark Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 79–89.

[6] Ajay N Bhoj and Niraj K Jha. 2009. Pragmatic design of gated-diode FinFET DRAMs. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*. IEEE, 390–397.

[7] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 using Automata Processing across Different Platforms. In *24th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE.

[8] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. 2016. Entity resolution acceleration using the automata processor. In *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 311–318.

[9] Leland Chang, David M Fried, Jack Hergenrother, Jeffrey W Sleight, Robert H Dennard, Robert K Montoye, Lidija Sekaric, Sharee J McNab, Anna W Topol, Charlotte D Adams, et al. 2005. Stable SRAM cell design for the 32 nm node and beyond. In *VLSI Technology, 2005. Digest of Technical Papers. 2005 Symposium on*. IEEE, 128–129.

[10] Ki Chul Chun, Pulkit Jain, Tae-Ho Kim, and Chris H Kim. 2012. A 667 MHz logic-compatible embedded DRAM featuring an asymmetric 2T gain cell for high speed on-die caches. *IEEE Journal of Solid-State Circuits* 47, 2 (2012), 547–559.

[11] Ki Chul Chun, Pulkit Jain, Jung Hwa Lee, and Chris H Kim. 2011. A 3T gain cell embedded DRAM utilizing preferential boosting for high density and low power on-die caches. *IEEE Journal of Solid-State Circuits* 46, 6 (2011), 1495–1505.

[12] Ki Chul Chun, Wei Zhang, Pulkit Jain, and Chris H Kim. 2011. A 700MHz 2T1C embedded DRAM macro in a generic logic process with no boosted supplies. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*. IEEE, 506–507.

[13] Computer Sciences Corporation. 2012. Big Data Universe Beginning to Explode. http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode.

[14] Paul Dlugosch, Dean Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *Parallel and Distributed Systems, IEEE Transactions on* 25, 12 (2014).

[15] DNV GL. 2016. Are you able to leverage big data to boost your productivity and value creation? https://www.dnvgl.com/assurance/viewpoint/viewpoint-surveys/big-data.html.

[16] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*. IEEE, 533–545.

[17] Victor Mikhaylovich Glushkov. 1961. The abstract theory of automata. *Russian Mathematical Surveys* (1961).

[18] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[19] Linley Gwennap. 2014. New Chip Speeds NFA Processing Using DRAM Architectures. In *In Microprocessor Report*.

[20] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. 2014. Haswell: The fourth-generation intel core processor. *IEEE Micro* 34, 2 (2014), 6–20.

[21] Fatih Hamzaoglu, Umut Arslan, Nabhendra Bisnik, Swaroop Ghosh, Manoj B Lal, Nick Lindert, Mesut Meterelliyoz, Randy B Osborne, Joodong Park, Shigeki Tomishima, et al. 2014. A 1Gb 2GHz embedded DRAM in 22nm tri-gate CMOS technology. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, 230–231.

[22] Zoran Jaksic. 2015. *Cache memory design in the FinFET era*. Ph.D. Dissertation. Universitat Politècnica de Catalunya.

[23] Zoran Jaksic and Ramon Canal. 2012. Enhancing 3T DRAMs for SRAM replacement under 10nm tri-gate SOI FinFETs. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE, 309–314.

[24] Rasha Karakchi, Lothrop O Richards, and Jason D Bakos. 2017. A Dynamically Reconfigurable Automata Processor Overlay. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE, 1–8.

[25] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. 2012. A case for exploiting subarray-level parallelism (SALP) in DRAM. *ACM SIGARCH Computer Architecture News* 40, 3 (2012), 368–379.

[26] Marzieh Lenjani and Mahmoud Reza Hashemi. 2014. Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities. *IET Computers & Digital Techniques* 8, 1 (2014), 30–48.

[27] Cong Liu and Jie Wu. 2013. Fast deep packet inspection with a dual finite automata. *IEEE Trans. Comput.* 62, 2 (2013), 310–321.

[28] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. 2018. Architectural Support for Efficient Large-Scale Automata Processing. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[29] Rui Liu, Xiaochen Peng, Xiaoyu Sun, Win-San Khwa, Xin Si, Jia-Jing Chen, Jia-Fang Li, Meng-Fan Chang, and Shimeng Yu. 2018. Parallelizing SRAM arrays with customized bit-cell for binary neural networks. In *Proceedings of the 55th Annual Design Automation Conference*. ACM, 21.

[30] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. 2012. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 461–472.

[31] Pascal Meinerzhagen, Adam Teman, Robert Giterman, Andreas Burg, and Alexander Fish. 2013. Exploration of sub-VT and near-VT 2T gain-cell memories for ultra-low power applications under technology scaling. *Journal of Low Power Electronics and Applications* 3, 2 (2013), 54–72.

[32] Micron. [n. d.]. RLDRAM Memory. https://www.micron.com/products/dram/rldram-memory.

[33] Indranil Roy and Srinivas Aluru. 2014. Finding motifs in biological sequences using the micron automata processor. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 415–424.

[34] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, and Hongning Wang. 2018. A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 665–674.

[35] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. Scalable and Efficient in-Memory Interconnect Architecture for Automata Processing. *IEEE Computer Architecture Letters* (2019).

[36] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. 2017. Frequent subtree mining on the automata processor: challenges and opportunities. In *International Conference on Supercomputing (ICS)*. ACM.

[37] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache Automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*.

[38] Prateek Tandon, Faissal M Sleiman, Michael J Cafarella, and Thomas F Wenisch. 2016. Hawk: Hardware support for unstructured log processing. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 469–480.

[39] Adam Teman, Pascal Meinerzhagen, Andreas Burg, and Alexander Fish. 2012. Review and classification of gain cell eDRAM implementations. In *Electrical & Electronics Engineers in Israel (IEEEI), 2012 IEEE 27th Convention of*. IEEE, 1–5.

[40] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards machine learning on the Automata Processor. In *International Conference on High Performance Computing*. Springer, 200–218.

[41] Jack Wadden, Kevin Angstadt, and Kevin Skadron. 2018. Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 749–761.

[42] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, et al. 2016. ANMLZoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 1–12.

[43] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An overview of micron's automata processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2016 International Conference on*. IEEE, 1–3.

[44] Ke Wang, Elaheh Sadredini, and Kevin Skadron. [n. d.]. Hierarchical Pattern Mining with the Micron Automata Processor. *International Journal of Parallel Programming (IJPP)*.

[45] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential Pattern Mining with the Micron Automata Processor. In *ACM International Conference on Computing Frontiers*.

[46] Michael HLS Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. 2016. Using the automata processor for fast pattern recognition in high energy physics experimentsĂŤ A proof of concept. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 832 (2016), 219–230.

[47] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. 2017. REAPR: Reconfigurable engine for automata processing. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 1–8.

[48] Wei Zhang, Ki Chul Chun, and Chris H Kim. 2013. A write-back-free 2T1D embedded DRAM with local voltage sensing and a dual-row-access low power mode. *IEEE Transactions on Circuits and Systems I: Regular Papers* 60, 8 (2013), 2030–2038.

[49] Keira Zhou, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. 2015. Brill tagging on the micron automata processor. In *Semantic Computing (ICSC), 2015 IEEE International Conference on*. IEEE, 236–239.