Cache Automaton

Arun Subramaniyan Jingcheng Wang Ezhil R. M. Balasubramanian
David Blaauw Dennis Sylvester Reetuparna Das
University of Michigan-Ann Arbor
{arunsub,jiwang,ezhilrmb,blaauw,dmcs,reetudas}@umich.edu

ABSTRACT

Finite State Automata are widely used to accelerate pattern matching in many emerging application domains like DNA sequencing and XML parsing. Conventional CPUs and compute-centric accelerators are bottlenecked by memory bandwidth and irregular memory access patterns in automata processing.

We present *Cache Automaton*, which repurposes last-level cache for automata processing, and a compiler that automates the process of mapping large real world Non-Deterministic Finite Automata (NFAs) to the proposed architecture. Cache Automaton extends a conventional last-level cache architecture with components to accelerate two phases in NFA processing: state-match and state-transition. State-matching is made efficient using a sense-amplifier cycling technique that exploits spatial locality in symbol matches. State-transition is made efficient using a new compact switch architecture. By overlapping these two phases for adjacent symbols we realize an efficient pipelined design.

We evaluate two designs, one optimized for performance and the other optimized for space, across a set of 20 diverse benchmarks. The performance optimized design provides a speedup of 15× over DRAM-based Micron's Automata Processor and 3840× speedup over processing in a conventional x86 CPU. The proposed design utilizes on an average $1.2MB$ of cache space across benchmarks, while consuming $2.3nJ$ of energy per input symbol. Our space optimized design can reduce the cache utilization to $0.72MB$, while still providing a speedup of 9× over AP.

CCS CONCEPTS

• **Hardware** → **Emerging architectures**; • **Theory of computation** → *Formal languages and automata theory*;

KEYWORDS

Emerging technologies (memory and computing), Accelerators

ACM Reference format:
Arun Subramaniyan Jingcheng Wang Ezhil R. M. Balasubramanian David Blaauw Dennis Sylvester Reetuparna Das. 2017. Cache Automaton. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3123986

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10...$15.00
https://doi.org/10.1145/3123939.3123986

1 INTRODUCTION

Non-deterministic Finite Automata (NFA) is a powerful computational model which is widely used in a number of application domains such as data analytics and data mining [7, 41], network security [16, 25, 29, 44], bioinformatics [13, 34, 42], tokenization of web pages [30], computational finance [1, 28] and software engineering [2, 11, 32]. These applications require processing tens to thousands of patterns for a stream of input data. NFAs are especially useful for efficient regular expression matching, as they can provide high-speed analysis of unstructured textual data, which is being generated in large volumes in forms such as system logs, social media posts, emails, and news articles [17].

NFA computation is inherently hard to speedup using compute centric processing. Modern multi-core processors and accelerators [37, 38] are limited by the number of transitions they can do in a given cycle. Both CPUs and GPGPUs perform poorly as automata processing is dominated by irregular memory access patterns and memory bandwidth limitations.

In comparison, a memory centric processing model can facilitate highly parallel and energy efficient processing of finite state automata in hardware. For instance, Micron's DRAM-based Automata Processor (AP) [14] has been shown to accelerate several applications like entity resolution in databases [7](by 434×) and motif search in biological sequences [34] (by 201×). Recent efforts at Virginia's Center for Automata Processing have demonstrated that AP can outperform x86 CPUs by 256×, GPGPUs by 32×, and accelerators such as XeonPhi by 62×, across a wide variety of automata benchmarks [39].

The success of memory centric models such as AP relies on two factors: massive parallelism and eliminating overheads in moving data between memory and compute units. Massive parallelism is due to the fact that all states (mapped to columns in DRAM arrays) can be independently activated in a given cycle. An AP chip can support up to 48K transitions in each cycle. Thus it can efficiently execute massive Non-deterministic Finite Automata (NFA) that encapsulate hundreds to thousands of patterns.

Given the large benefits of memory centric models, we explore cache as a substrate for automata processing. Caches have two advantages over DRAMs. *First*, SRAM-based caches are faster and more energy efficient compared to DRAM. *Second*, caches are integrated on processor dies which are manufactured in cutting edge technology nodes and performance optimized logic. Thus cache based automata processing can utilize significantly faster interconnects and logic.

On the flip side, caches typically have lower capacity compared to DRAM. Interestingly, we observe that DRAM-based AP sacrifices a huge fraction of die area to accommodate the routing matrix and other non-memory components required for automaton processing. An AP die can store 12 Mbits of data, while a conventional DRAM die of equivalent area can store 200 Mbits of data (16.6× area overhead) [18]. Thus, while DRAM memory's packing density is high, DRAM automata processor's packing density is comparable to caches. A rank of AP (8 dies) can accommodate 384K states. Typical high-performance processors can have 20-40MB of last level cache [8] and can accommodate 640K-1280K states, if the entire cache is utilized to save NFAs.

In this paper, we propose the *Cache Automaton* architecture which repurposes existing Last Level Cache (LLC) slices to enable efficient automata processing. While the memory technology benefits of moving to SRAM are apparent, repurposing the 40-60% passive LLC die area for massively parallel automata computation comes with several challenges. A naive approach that processes an input symbol every LLC access (∼20-30 cycles @ 4GHz), would lead to an operating frequency comparable to DRAM-based AP (∼200 MHz), negating the memory technology benefits. Increasing operating frequency further can be made possible only by two insights.

First, architecting an in-situ computation model that is cognizant of the internal geometry of LLC slices. We observe that LLC access latency is dominated by wire-delays inside a cache slice, accessing upper-level cache control structures, and network-on-chip. Thus, while a typical LLC access can take ∼30 cycles, an SRAM array access is only 1 cycle. Fortunately, in-situ architectures such as cache automaton require only SRAM array accesses and do not incur the overheads of a traditional cache access. Furthermore, we leverage the internal geometry of LLC slices to build a hierarchical state-transition interconnect.

Second, accelerating the two phases of input symbol processing common in memory-centric automata processing models. These are *state-match*, where the set of active states whose label matches the input symbol are determined through an array read operation and *state-transition*, where the matching states activate their corresponding next states by propagating signals through wires and switches. Accelerating state-match is challenging because industrial LLC subarrays are optimized for packing density and designed in a manner that many bit-lines share I/O (or sense amplifiers) via a column multiplexer. In our LLC design modeled exactly after the Xeon E5 processor [10, 19], 8 states share a sense amplifier. This implies that reading out all states will require 8 cycles, resulting in a low throughput system. We observe that unlike conventional cache accesses, automata processing requires reading out all the bits which are sharing a sense amplifier and propose a sense amplifier cycling technique to address this bottleneck. In the optimized read sequence, all the bit-lines are pre-charged in parallel, which is then followed by sequential sensing of each bit-line. This optimization can improve our system throughput by 2×-3× depending on state packing density.

Accelerating state-transition at low-area cost requires the design of a scalable interconnect that efficiently encodes and supports multiple state-transitions on the same cycle, often to the same destination state. Automata switches also need to store a large array of connectivity bits that are representative of transition edges between states. We observe that an 8T SRAM memory array can be repurposed to become a compact state-transition crossbar for automata. Alternatively, traditional crossbar designs that require arbitration every cycle for determining input-output connections have prohibitive area costs. Furthermore they cannot support large fan-in of states, i.e. multiple inputs connecting to one output. Supporting this feature would need multiple arbitrations/traversals through the crossbars or multiple crossbars, making the state-transition either slow or have higher area overheads.

Even with such compact switches, supporting all-to-all connectivity between states requires prohibitively large and slow switches. We observe that large real-world NFA are typically composed of several connected components, each of which can be grouped into densely connected partitions with only few (8-16) interconnections between partitions. This motivated us to explore a hierarchical switch topology with local switches providing rich intra-partition connectivity and global switches providing sparse inter-partition connectivity. To this end, we develop a *Cache Automaton compiler* which scales to real world NFAs with several thousand states and maps them efficiently to hundreds of SRAM arrays. Our compiler uses graph partitioning techniques [23] to satisfy the connectivity constraints while maximizing space utilization.

To further improve throughput and parallelism, we develop a fully *pipelined* design. Pipelining is possible because Cache Automaton processes a stream of input symbols (MBs to GBs) sequentially, and SRAM access for the current input symbol can be overlapped with the switch propagation delay for processing the previous input symbol.

In summary this paper offers the following contributions:

- This is the first work to explore automata processing in cache. The proposed *Cache Automaton* architecture maps NFA states to SRAM arrays of last level cache slices. In particular we observe that memory centric automata processing models can leverage not only the faster SRAM memory technology of caches but also faster interconnect/logic in a processor die. Further, the capacity of caches are quite comparable to DRAM-based automata processors, which sacrifice the density to accommodate custom interconnect-based transition matrix.
- A critical component of the proposed architecture is a *programmable interconnect* which enables transitions between NFA states. By repurposing 8T SRAM arrays, we develop a *new compact switch* architecture customized for automata processing.
- To improve performance, we propose a fully *pipelined* architecture which is based on our observation that SRAM access for the current input symbol can be overlapped with interconnect delay for processing the previous input symbol.
- The bottleneck of the Cache Automaton pipeline is SRAM array access delay which is slow due to sharing of I/O (sense-amplifiers) between bit-lines. We observe that unlike conventional cache accesses, Cache Automaton needs to read all column multiplexed bits and propose *sense-amplifier cycling* techniques to speedup SRAM array access.
- We develop a *Cache Automaton compiler* which leverages many algorithmic insights and fully automate the process of mapping real world NFAs with tens of thousands of states to Cache Automaton which consists of hundreds of SRAM arrays.
- We evaluate two designs, one optimized for performance and the other optimized for space, across a set of 20 diverse benchmarks. The performance optimized design provides a speedup of 15× over Micron's AP and 3840× speedup over

processing in a conventional x86 CPU. The proposed design utilizes on an average 1.2MB of cache space across benchmarks, while consuming 2.3nJ of energy per input symbol. Our space optimized design can reduce the cache utilization to 0.72MB on an average across benchmarks, while still providing a speedup of 9× over AP.

2 CACHE AUTOMATON ARCHITECTURE

In this section we provide an overview of NFAs, explain the concept of *Cache Automaton* and provide a simple working example.

2.1 NFA Primer

A Non-deterministic Finite Automaton (NFA) is formally described by a quintuple $\langle Q,\Sigma,\delta,q_0,F\rangle$, where Q is a set of states, Σ is the input symbol alphabet, q_0 is the set of start states and F is the set of *reporting* or *accepting* states. The transition function $\delta(Q,\alpha)$ defines the set of states reached by Q on input symbol α. The non-determinism is due to the fact that an NFA can have multiple states active at the same time and have multiple transitions on the same input symbol.

The classic representation is transformed to ANML NFA representation [14] where *each state has valid incoming transitions for only one input symbol*. Thus each state in an ANML NFA can be labeled by one unique input symbol. There exists various algorithms (e.g. [35]) to transform a classical NFA to an optimized ANML NFA format. Figure 1 (a) shows the classical representation of a sample NFA which accepts patterns {`bat, bar, bart, ar, at, art, car, cat, cart`}. Figure 1 (b) shows the ANML NFA representation for the same automata. State `S1` in classical representation is now represented by three states `S1_a` (with label `a`), `S1_b` (with label `b`), and `S1_c` (with label `c`).

2.2 Cache Automaton Concept

ANML NFA computation entails processing a stream of input symbols one at a time. Initially, all the start states are active states. Each step has two phases. In *state match* phase, we identify which of the active states have the same label as the current input symbol. In the next *state transition* phase, we look up the transition table to determine the destination states for these matched states. These destination states would become the active states for the next step. Now, we discuss how Cache Automaton implements these two phases efficiently.

State Match: We adapt Micron's AP processor [14] design for implementing the state match phase. Each NFA state is mapped as a State Transition Element (STE) to a *column* of SRAM arrays in the last-level cache. The value of an STE column is set to the one-hot encoding of the 8-bit input symbol it is mapped to. This means that each STE (or column) is 256 bits and each bit position signifies an input symbol in the ASCII alphabet. Figure 2 (a) shows an SRAM array in Cache Automaton which holds 256 STEs. Every cycle, the current input symbol is broadcasted to all SRAM arrays as a *row address* and the corresponding row is read out. If an STE has a '1' bit set in the row, it means that the label of the state it has stored matches the input symbol. Thus, by broadcasting the input symbol to all SRAM arrays, it is possible to determine in parallel all the states which *match* the current input symbol.

The row corresponding to the input symbol is read out and stored in a match vector. An active state vector (one bit per STE; 256-bit vector in our example) stores which STEs are active in a given cycle. A logical AND of match and active state vectors determines the subset of active states which match the current input symbol. The destination states of these matched states would become the next set of active states. Section 2.6 discusses solutions to implement this state match phase efficiently in cache sub-arrays.

State Transition: This phase determines the destination states of the matched states found in the previous phase. These states would then become the next set of active states. We observe that a matrix-based crossbar switch (essentially a $N\times N$ matrix of input and output ports) is suitable to encode a transition function. In a crossbar, an input port is connected to an output port via a cross-point. Each STE connects to one input port of the switch. A cross-point is enabled if the input STE connects to a specific output STE. The result of state-matches serve as inputs to the switch, and the output of the switch is the next set of active states.

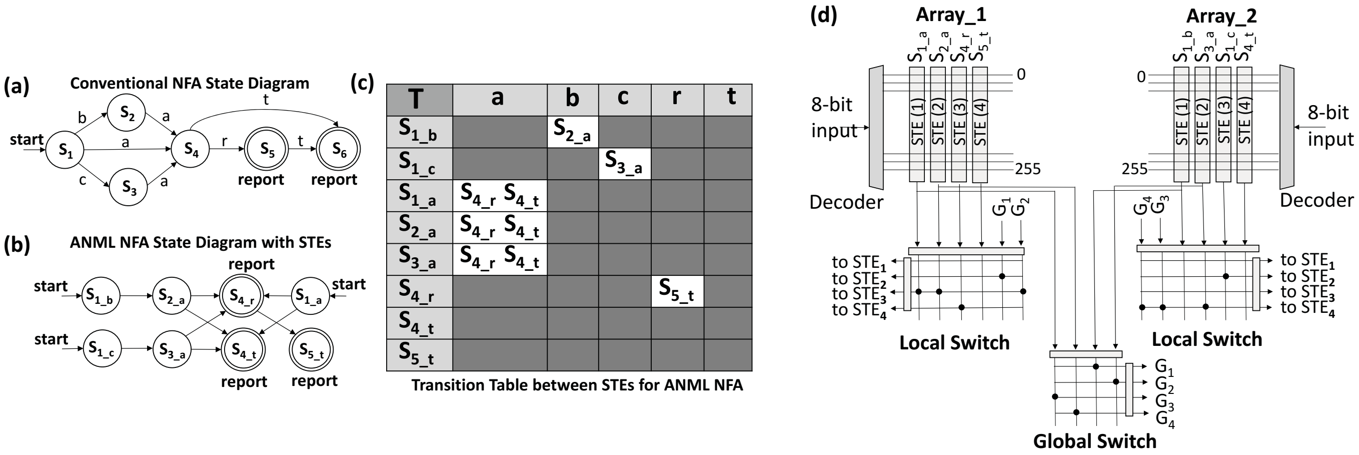
The switches in the cache automaton architecture have modest wiring requirements (256-512 input and output wires; see Table 2), as data-bus width is only 1-bit. However, cache automaton switches have two major differences from conventional switches. *First*, there is no need for arbitration. The connections between the input and output ports can be configured once during initialization for an NFA and then used for processing all the input symbols. Since there is no arbitration, the enable bits must be stored in the cross-points. Automaton switches have a large number of cross-points, and therefore we need a compact design to store the enable bit at each cross-point. *Second*, unlike a conventional crossbar, an output can be connected to multiple inputs at the same time. The output is a logical OR of all active inputs. Section 2.7 discusses our proposed switch architecture for Cache Automaton which supports the above features.

Ideally, the entire transition function could be encoded in one switch to provide maximum connectivity. However, such a design will be incredibly slow. To scale to thousands of states and many SRAM arrays, we adopt a hierarchical switch architecture as discussed in Section 2.4.

2.3 Working Example

We describe a simplified example which brings together all the above concepts. Figure 1 shows an example NFA which accepts patterns {`bat, bar, bart, ar, at, art, car, cat, cart`} and how it is mapped to SRAM arrays and switches. The figure starts with a classical representation of an NFA in terms of states and transitions (Figure 1 (a)). Figure 1 (b) shows ANML NFA representation for the same automata. Figure 1 (c) shows the transition table for ANML NFA with STEs. This example NFA requires only 8 STEs. Real world NFAs have tens of thousands of states which need to be mapped into hundreds of SRAM arrays.

For this example, let us assume we have two small SRAM arrays which can each accommodate 4 STEs as shown in Figure 1 (d). The NFA requires 8 STEs, so we split the states into 4 STEs in Array_1 and Array_2. Each array has a 6×4 local switch, and together they share a 2×2 global switch. Each STE can connect to all STEs in its array. In this example, only two STEs (STE_1 and STE_2) in an array



T	a	b	c	r	t
S_{1_b}		S_{2_a}			
S_{1_c}			S_{3_a}		
S_{1_a}	S_{4_r} S_{4_t}				
S_{2_a}	S_{4_r} S_{4_t}				
S_{3_a}	S_{4_r} S_{4_t}				
S_{4_r}				S_{5_t}	
S_{4_t}					
S_{5_t}					

Transition Table between STEs for ANML NFA

Figure 1: An example NFA and its mapping to two small SRAM arrays and switches. The NFA accepts patterns `{bat, bar, bart, ar, at, art, car, cat, cart}`.

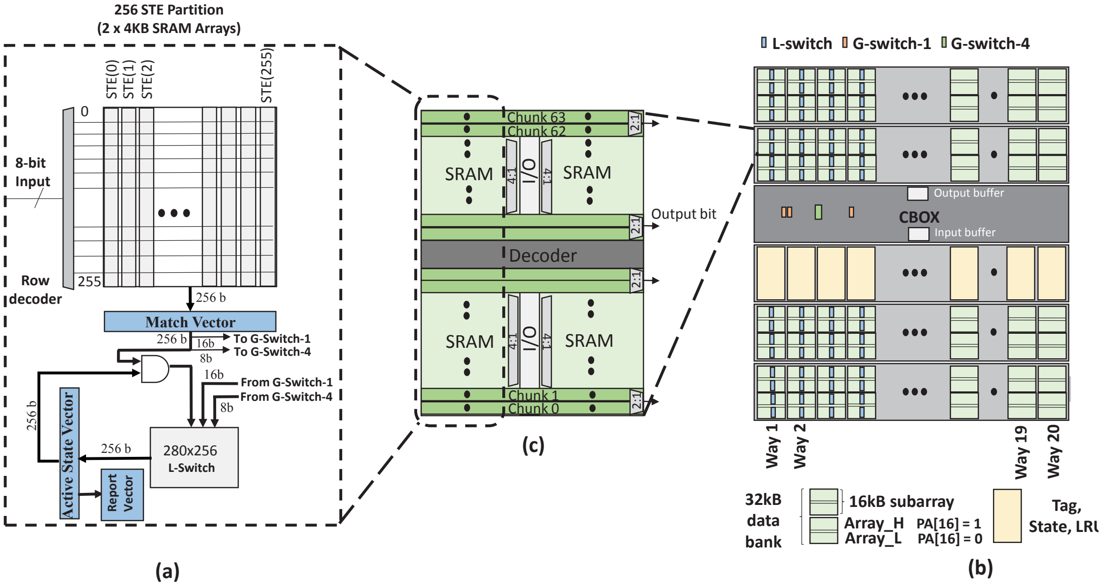


Figure 2: The figure shows (a) SRAM arrays re-purposed to store 256 STEs, (b) one 2.5MB Last-Level Cache (LLC) slice architecture., and (c) Internal organization of one 8KB sub-array.

are allowed to connect to all STEs in the other array via the global switch.

The transition table in Figure 1 (c) is mapped to local and global switches. For instance, S_{1_a} and S_{4_r}, are mapped to STE_1 and STE_3, of Array_1. Since S_{1_a} can transition to S_{4_r}, the local switch cross-point between STE_1 and STE_3 is set to connected (represented by black dot). The figure also shows how a connection via global switch is established for states S_{2_a} mapped to STE_2 of Array_1, and S_{4_t} mapped to STE_4 of Array_2. This is accomplished by (1) feeding STE_2 as an input to global switch, (2) connecting second input of global switch to G_4 output which feeds as an input to Array_2's local switch, (3) G_4 input is connected to STE_4 output (or S_{4_t}) of Array_2's local switch.

2.4 Cache Slice Design

The proposed cache automaton is implemented in the Last-Level Cache (LLC) in order to accommodate large NFA with thousands of states. Figure 2 (b) shows the overall organization of a slice of LLC with the Cache Automaton architecture. The depicted LLC slice is modelled *exactly* after Xeon E5 processors [10, 19]. Each LLC slice is 2.5MB. Intel processors support 8-16 such slices [8]. Each slice has a central cache control box (CBOX). Remainder of the slice is organized into 20 columns. A column consists of eight 16 KB data sub-arrays, and a tag array. Each column represents a way of set-associative cache. Internally a 16 KB data sub-array consists of *four* SRAM arrays with 256×128 6T bit-cells as shown in Figure 2 (c). Each array has 2 redundant columns and 4 redundant rows to map out dead lines. STEs are stored in columns of the 256×128 SRAM array. A 16 KB data sub-array can store up to 512 STEs. We define a *partition as group of 256 STEs* mapped to two SRAM arrays each of size 4KB.

Our interconnect design is based on the observation that real-world NFA states can typically be grouped into partitions, with states within a partition requiring rich connectivity and states in different partitions needing only sparse connectivity (details in Sections 3.1 and 3.2). To support this partitioning, we add one local switch (*L-Switch*) per partition providing rich intra-partition connectivity and global switches (*G-Switch*) for sparse connections between partitions across a way or multiple ways. Each L-Switch is of size 280 × 256, i.e., 280 input wires and 256 output wires. The input wires correspond to 256 STEs in the sub-array and 16 input wires from the G-Switch in the same way (*G-switch-1*) and 8 input wires from the G-Switch connecting the 4 ways (*G-switch-4*). A STE in a partition can connect to any other STE in its partition via the L-Switch. Also, 16 STEs from a partition can connect to other partitions in the same way via *G-Switch-1* and 8 STEs from a partition can connect to other partitions via *G-switch-4*. We also observe that even without connections between global switches, the proposed interconnect topology provides sufficient headroom to our compiler to map all evaluated NFA.

Figure 2 (a) shows a single partition. The input symbol match result is read out and stored in the *match vector*. The logical AND of the match vector and the active state vector is fed as input to the local-switch (256 STEs) and the global-switches (16 STEs and 8 STEs respectively). After the signals return from the global-switches to the local-switch, the next set of active states is available as the output of the local-switch. This is written back to the active-state vector. If any of the final or reporting states are active then an entry is made into the output buffer in the CBOX recording the match (Section 2.8).

2.5 Automaton Pipeline

The automaton processes a stream of input symbols *sequentially* and hence the time to process each input symbol determines the clock period. The clock period determines the rate of processing input symbols and hence the overall system performance. We observe that each input symbol is processed in two independent phases, SRAM access for state match and propagation through the interconnect (switches and wires) for state transition. Furthermore, SRAM access for the current input symbol can be overlapped with interconnect delay for processing the previous input symbol. Based on this observation we design a three-stage pipeline for *Cache Automaton*. Typical application scenarios process large amounts of input data (MBs to GBs), thus the pipeline fill-up and drain time are inconsequential.

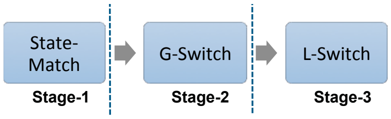


Figure 3: Three-stage pipeline design for Cache Automaton.

The pipeline stages are shown in Figure 3. The first stage of the pipeline is state-match or SRAM array read access. The output of this stage is stored in the match vector (Figure 2 (a)). It is moved to another buffer at the end of the stage to make room for the next state match. The second stage of the pipeline is propagation through the global switch (G-Switch). This includes the wire delay from SRAM array to global-switch. The output of this stage is stored in the output latches of G-Switch. The third and last stage of pipeline is propagation through local switch (L-Switch). This includes the wire delay from G-Switch to L-Switch. This stage writes the next state to active state vector and completes processing of the current input symbol. Note, the output of the L-switch updates the active state vector, and the active state cannot be updated until transition signals from other partitions have reached their destination L-switches via the G-switch. Thus G-Switch forms the second stage of pipeline followed by L-switch.

As can be inferred from the above discussion, the symbol processing time for *Cache Automaton* is determined by symbol-match delay and switch delay. Hence for high-performance it is critical to speedup both the symbol match delay and interconnect delay. The next two sections discuss techniques towards this end. Section 5.1 quantifies the delay of each pipeline stage and overall operating frequency.

2.6 Enabling Low-Latency Matches

The input symbol match for cache automaton is simply an SRAM array read operation. Conventional SRAM arrays share I/O or sense-amps for increased packing density. This results in column-multiplexing which increases the input symbol match time significantly. For example, a 4-way column multiplexed SRAM array shares one sense-amp across four bit-lines and hence can read out only 1 bit out of 4 bits in a cycle as shown in the left side of Figure 4. A read (or match operation) consists of decoding, bit-line pre-charging, and I/O or sensing as shown in the baseline timing on the right side of Figure 4. A 4-way column multiplexed array requires 4 cycles to read out all the bits in a row of the array.

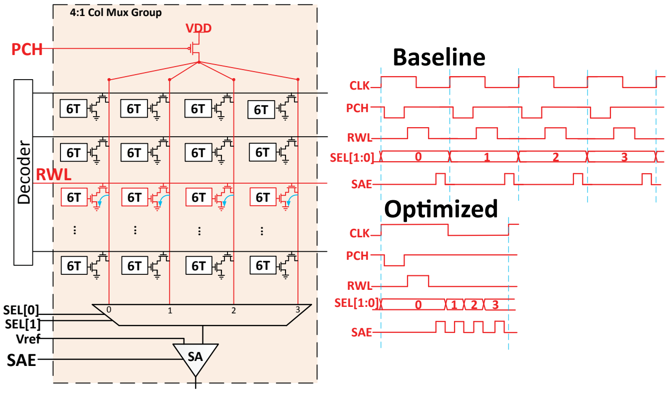


Figure 4: Design and timing diagram for a 4-way multiplexed SRAM column.

A dynamic scheme which checks active state vector, and matches fewer STEs, can save energy. Unfortunately this cannot improve performance because the *clock period is determined by worst case state-match time*. However, we observe that unlike conventional cache read accesses, automata state transitions need to read all bits which are column multiplexed. This can be leveraged to improve match latency, by cycling the sensing phase. All the bit-lines of an

SRAM array can be pre-charged in parallel, followed by sequentially sensing column multiplexed bits.

Figure 4 shows the timeline of an optimized read sequence for 4-way column multiplexing. First pre-charge (PCH) is asserted, followed by a read word-line (RWL) assertion. By the end of RWL assertion all bit-lines are ready for sensing. The sense-amp enable (SAE) is asserted in 4 steps. The column-multiplexer select signals (SEL) is set to 0 to read the first column, and changes to 1,2, and 3 with each SAE assertion. Typically SRAM arrays use pulse generators for control signals like SAE, PCH and RWL and are not generated using a separate higher frequency (i.e., 8GHz) clock. These consist of a chain of high-Vt, long channel, current-starved inverters and NAND gate that can be configured to generate pulses of widths 1/2–1/4 of clock period. In our case, a 125 ps (8 GHz) pulse can be generated for SAE and SEL. Power and area overhead for the pulse generator is minimal–8-10 inverters switching once every clock cycle (8 μW). Since sensing takes about 25% of the cycle time, this optimized read sequence can read all bits for a 4-way column multiplexed array 2× faster than the baseline. For 8-way column multiplexing the benefits of the optimization are higher.

2.7 Switch Design

This section discusses the proposed switch design. As explained in Section 2.2, an automaton switch needs to support two new features. First, since it has a large number of 1-bit ports, it needs to store a large number of cross-point enable bits. There is no need for dynamic arbitration. But the switch needs to provide a configuration mode which allows write to enable bits. Second, to allow efficient many-to-many state transitions, an output needs to be connected to multiple inputs and be the equivalent of logical OR of active inputs.

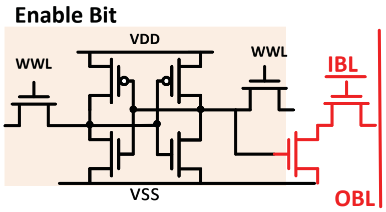


Figure 5: The 8T cross-point design for switches.

To support the above two features, we developed an 8 transistor (8T) cross-point design as shown in Figure 5. The enable bit which controls the connection of input bit-lines (*IBL*) to output bit-lines (*OBL*) is stored in a 6T bit-cell. The connection between *IBL* and *OBL* is via a 2T block. The switch supports two modes: crossbar mode and write mode. During the *crossbar mode*, the OBL's are pre-charged. If any of the IBL's carry a '0', the OBL is discharged. Thus outputs carry a wired AND of inputs. Note the inputs and outputs are active low. Thus the final result on an output wire is logical OR of all inputs. Each *OBL* has a dedicated sense-amp. During the *write mode*, the 6T enable bits can be programmed by writing to all bit-cells sharing one write word-line (WWL) in a cycle. The switch is provisioned with a decoder and wordline drivers for write mode. The proposed switch can take advantage of standard 8T push rule bit cells to achieve a compact layout.

2.8 Input Streaming and Output Reporting

Cache Automaton takes a steady stream of input symbols and produces intermittent output matches. Input symbols (1 byte each) are stored in a small 128 entry FIFO in the C-BOX as shown in Figure 2. The FIFO is associated with an input symbol counter which indicates the symbol cycles elapsed. One input symbol is read from the FIFO every cycle and broadcasted to all SRAM banks by using the existing address bus within a cache slice. Our model assumes that applications copy input data to a cache location. In the *Cache Automaton* mode, as input symbols get processed and deleted from the FIFO, the cache controller reads a cache block worth of input data via regular cache access and fills up the FIFO.

We follow a model similar to Micron's AP for output reporting. An output buffer has 64 entries, an entry for each match with a reporting state. An interrupt is sent to CPU if all output buffer entries are full. The report states in the NFA can be mapped to designated STEs in a partition. A 256-bit mask indicates if the reporting states are mapped to these STEs. A wired OR of the result of logical AND of active state mask and output reporting mask (report vector) triggers an output reporting event. An output reporting event creates a new entry in the output buffer consisting of active state mask, partition ID, input symbol, and input symbol counter.

2.9 System Integration

While repurposing the last-level cache for automata processing, the following system-level issues need to be kept in mind:

Sharing Model with CPU: Our architecture is aware of a *way* in LLC, but there is no mode for directly addressing a cache way in x86 instructions. A load address can be mapped to any way in the LLC. To overcome this limitation, Cache Automaton leverages Intel's Cache Allocation Technology (CAT) [20] to dynamically restrict the ways accessed by a program and thus exactly control which cache way the data gets written to. NFA computation is carried out only in 4-8 ways of each slice. The remaining 12-16 ways can be used by other processes/VMs executing on the same/different cores without leading to starvation in inclusive caches. By associating the NFA process to one of the highest `cgroups` (class-of-service), CAT can ensure that incoming data from processes in low-priority `cgroups` does not evict data in active NFA ways and guarantees QoS during steady-state. With regard to addressing, LLC hashing must be disabled during configuration time to place STE data into specific slices. This can be done by writing to special Model-Specific-Registers (MSRs) like those used to associate L3 cache ways with `cgroups` in CAT. Note that LLC hashing need not be disabled during normal execution.

Power Management: Since NFA computation has high peak power requirements for some benchmarks, the OS scheduler together with the power governor must ensure that the system TDP is not exceeded while scheduling other processes simultaneously on CPU cores. Based on the number of cache arrays, ways, slices allocated for NFA computation and average active states for representative inputs, the compiler can provide coarse-grained peak-power estimates (hints) to guide OS scheduling. In case the OS wishes to schedule a higher-priority process, the NFA process may also be suspended and later resumed by recording the number of input symbols processed and the active state vector to memory. It must be noted that while peak power is high, the energy consumed is orders of magnitude lower than that expended by conventional CPUs due to savings in data movement and instruction processing overheads.

2.10 Configuration and ISA Interface

We adopt a configuration model similar to Micron's Automata Processor (AP). Before processing the NFA, the switches have to be configured and the cache arrays have to be initialized with STEs. The switches can be configured by utilizing their write-mode, where they simply function as SRAM arrays. We assume switch locations are I/O mapped and addressable by CPU load instructions.

The initialization of cache arrays can be done by CPU load instructions which fetch data from memory to caches. Our compiler creates binary pages which consists of STEs stored in the order in which they need to be mapped to cache arrays. The compiler carefully orders the STEs based on physical address decoding logic of underlying cache architecture. These binary pages with STEs are loaded in memory, just like code pages. Most LLC cache sets are addressable by last 16 bits of memory address. These bits can be kept same for both virtual pages and physical pages by mapping the binary data (containing STEs) to huge pages [24]. LLC hashing is disabled during configuration as discussed in Section 2.9.

The average initialization time is dictated by the number of SRAM arrays occupied by an NFA. For our largest benchmark, we found this to be about $0.2ms$ on a Xeon server workstation. In contrast AP's configuration time can be up to tens of milliseconds [36]. Once configured, in typical use cases such as log processing, network traffic monitoring and DNA alignment, NFAs typically process GBs/TBs of data, thus processing time easily offsets configuration time. But configuration may be costly when frequently switching between structurally different automata. For these, optimizations like overlapping the configuration of one LLC slice with processing in others and prefetching may be explored. We leave this exploration to future work.

An ISA interface is required to specify (1) when to start processing in Cache Automaton mode and (2) start address from which input data needs to fetched to fill up input FIFO buffer. (3) the number of input symbols to be processed. One new instruction can encapsulate all the above information. Compiler can insert this special instruction in code whenever it needs to process NFA data. An interrupt service routine handles output buffer full reporting events.

3 CACHE AUTOMATON COMPILER

In this section we explain our compiler that takes as input an NFA description consisting of several thousands of states and efficiently maps them onto cache banks and sub-arrays. Care is taken to ensure maximum cache utilization while respecting the connectivity constraints of the underlying interconnect architecture. The algorithms proposed are general and the insights provided are also applicable for mapping NFAs to any spatial reconfigurable substrate with memory like FPGAs or memristor crossbar arrays.

The compiler takes as input an NFA described in a compact XML-like format (ANML) and generates a bit-stream containing information about the NFA state to cache array mapping and the configuration enable bits to be stored in the various crossbar switches of the automaton interconnect. We propose two mapping policies leading to two *Cache Automaton* designs, one optimized for performance *CA_P* and the other optimized for space utilization *CA_S*. Before proceeding to explain the mapping algorithm used by the compiler (Section 3.2), we motivate the insights behind each of these designs in Section 3.1.

3.1 Connectivity Constraints

Real-world NFAs are composed of several *connected components (CCs)* with only a few hundred states each. Each *connected component* describes a pattern or group of common patterns to be matched. Since these *connected components* have no state transitions between them, they can be treated as atomic units by the mapping algorithm. Each connected component can be viewed as operating independently matching against the input symbol in parallel.

Performance Optimized Mapping: In their baseline NFAs, most benchmarks have connected components with less than 256 states (refer Table 1), making it possible to fit at least one *connected component* in each partition (256 STEs stored in two 4KB SRAM arrays, Figure 2 (a)). This motivates our *performance-optimized* mapping scheme that greedily packs connected components onto cache arrays. We were able to operate all of the baseline NFA benchmarks while limiting the connectivity across cache arrays to within a way. Only cache arrays which are mapped to physical address with A[16] = 0 (Array_L in Figure 2 (c)) are used for mapping NFA and cache arrays with A[16] = 1 (Array_L in Figure 2 (c)) can be used for storing regular data provided that compiler can ensure that regular data are placed in 64KB segments in the virtual address space or OS does not use physical pages with A[16] = 1. As we discuss in Section 5.1, the performance-optimized design can operate at a frequency of 2 GHz.

Space Optimized Mapping: However, just using the baseline NFAs forgoes algorithmic optimizations on NFAs that seek to remove redundant automata states and state traversals. These redundancies are common in practice, since many patterns share *common prefixes* (for example, patterns like art and artifact) and these common prefixes can be matched once for all connected components together. Eliminating redundancies helps reduce the space footprint of the NFA. It also reduces the average number of active states, leading to reduction in dynamic energy consumption. This has been the motivation for several *state-merging algorithms* in literature that merge common prefixes across pattens [4]. However, it must be kept in mind that since these optimizations merge states across many connected components they tend to reduce the number of connected components and *increase the average connected component size*. Larger connected components require richer connectivity and demand more interconnect resources (crossbar switches and wires). To support such state-merged NFA, we propose a *space-optimized* mapping policy that leverages graph partitioning techniques (Section 3.2) to minimize outgoing edges between partitions. We also provision additional global crossbar switches to ensure connectivity across 4-8 ways of a cache slice. Our hierarchical switch design provides a richer average fan-out transitions and fan-in transitions per state compared to the AP (Section 5.4). However, richer connectivity comes at the cost of higher latency due to increased wire delay, therefore the *space-optimized* design operates at a lower frequency (1.2 GHz) compared to the *performance-optimized* design (2 GHz).

3.2 Mapping Algorithm

The algorithm takes as input the ANML NFA description of the benchmark, the number of cache arrays available and the size of each cache array. The output is a mapping of NFA states to cache arrays. It operates in three steps. In the first step, all connected components which have size less than *partition_size* (i.e., 256 states) are identified. As discussed earlier, a connected component forms the smallest mapping unit. Next, these connected components are mapped greedily onto the cache arrays to pack multiple connected components onto the same cache array when possible. We do not partition the connected component in the first stage, since the connected component inherently groups together states that have plenty of state-transitions between them and mapping these states to the same array leads to a more space-efficient packing. Connected components larger than *partition_size*, need to be partitioned across k-different partitions (in the same way or across multiple ways of the cache slice). We utilize the open-source graph partitioning framework *METIS* [23] to solve this *k-way* partitioning problem. *METIS* partitions the connected component into different partitions such that the number of outgoing state transitions between any two partitions is minimized. It works by first coarsening the input connected component, performing bisections on the coarsened connected component and later refining the partitions produced to minimize the edge cuts. We ensure that *METIS* produces load-balanced partitions with nearly equal number of states per partition. For all of our benchmarks (in Table 1) *METIS* consistently produces connected component partitions that have less than 16 state-transitions between them. The maximum number of outgoing state-transitions from an array determines the radix of the global-switch to be supported.

3.3 Case Study: Entity Resolution

Figure 6 presents the application of our mapping algorithm to the *space-optimized version* of the *Entity Resolution* benchmark. Entity Resolution is widely used for approximate string matching in databases. The benchmark has 5672 states with 5 connected components (CCs). The largest connected component CC_4 has size 4568 and the smallest one CC_0 has 75 states. Each of the arrays (Array_H and Array_L) in a 16kB subarray of the LLC slice shown supports 256 states. Our mapping algorithm proceeds as follows. For each unallocated array, starting from the smallest connected component, greedily pack as many connected components in the array as the array can accommodate. If the connected component size exceeds the size of an array, then we invoke the *k-way partitioning* algorithm in METIS for different values of k based on the connected component size.

From the final mapping obtained, it can be seen that a fairly dense packing of CCs is achieved. It can be seen that both CC_0 and CC_1 are allocated the same array. CC_2 takes up a separate array while CC_4 spans across 3 ways. Local switches at each array and global switches for both 1 and 4 ways support intra-array and inter-array state transitions respectively. Furthermore, the densely connected arrays for CC_4 (having many outgoing transitions) are also allocated to arrays in the same way.

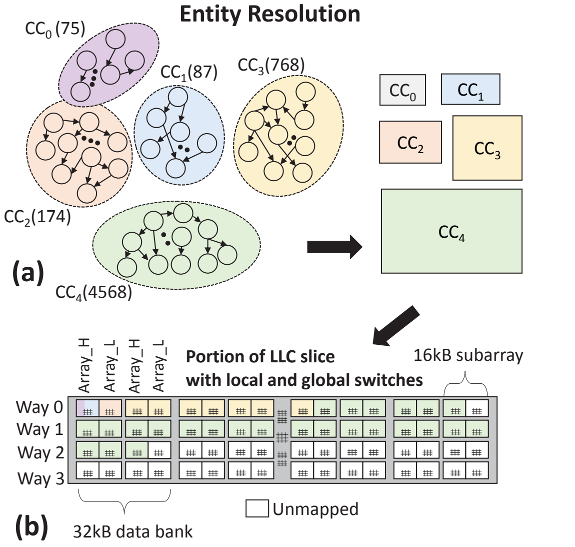


Figure 6: Figure showing mapping of connected components in *EntityResolution* to cache arrays. The connected components are labeled along their size in brackets

4 EVALUATION METHODOLOGY

In this section we first describe our workloads, followed by a discussion on the experimental setup and finally report the different system parameters.

NFA workloads: We evaluated the proposed approach and mapping schemes on a wide range of benchmark FSMs from the *ANMLZoo* [39] and the *Regex* [5] benchmark suites. These real world benchmarks span multiple domains including network packet monitoring [5], gene sequence matching [34] and natural language processing [49]. The NFAs used in this work form the core of many end-to-end applications. For example, the oligo_scan routine used for pattern matching in Weeder 2.0, an open-source tool for motif discovery in DNA sequences, uses an automaton similar to Protomata that contributes 30-62% of the total runtime. In the Apriori algorithm for frequent itemset mining, NFA processing accounts for 33-95% of the execution time, based on the frequency threshold. FSM-like computations form the core of many activities inside a web browser, taking about 40% of the loading time for many web pages [22, 48]. Table 1 summarizes some of the important characteristics of these FSMs and the parameters used in our simulations. We used the 10MB input traces for our evaluation. Similar trends in results are observed for larger inputs.

Experimental Setup: We utilize the open-source virtual automata simulator *VASim* [39] to simulate the proposed architecture. *VASim* allows for fast NFA emulation by traversing paths only for active states. The simulator takes as input the NFA partitions produced by METIS and simulates each input cycle by cycle. After processing the input stream, we use the per-cycle statistics on number of active states in each array to derive energy statistics.

Table 2 provides the various delay and energy parameters assumed in our design. To estimate the area, power and delay of the memory array we use a standard foundry memory compiler for the 28nm technology node. The nominal voltage for this technology is 0.9

		Performance optimized				Space optimized			
#	Benchmark	States	Connected Components	Largest CC Size	Avg.Active States	States	Connected Components	Largest CC Size	Avg.Active States
1	Dotstar03	12144	299	92	3.78	11124	56	1639	0.84
2	Dotstar06	12640	298	104	37.55	11598	54	1595	3.40
3	Dotstar09	12431	297	104	38.07	11229	59	1509	4.39
4	Ranges05	12439	299	94	6.00	11596	63	1197	1.53
5	Ranges1	12464	297	96	6,43	11418	57	1820	1.46
6	ExactMath	12439	297	87	5.99	11270	53	998	1.42
7	Bro217	2312	187	84	3.40	1893	59	245	1.89
8	TCP	19704	715	391	12.94	13819	47	3898	2.21
9	Snort	69029	2585	222	431.43	34480	73	10513	29.59
10	Brill	42568	1962	67	1662.76	26364	1	26364	14.29
11	ClamAV	49538	515	542	82.84	42543	41	11965	4.30
12	Dotstar	96438	2837	95	45.05	38951	90	2977	3.25
13	EntityResolution	95136	1000	96	1192.84	5672	5	4568	7.88
14	Levenshtein	2784	24	116	114.21	2784	1	2605	114.21
15	Hamming	11346	93	122	285.1	11254	69	11254	240.09
16	Fermi	40783	2399	17	4715.96	39032	648	39038	4715.96
17	SPM	100500	5025	20	6964.47	18126	1	18126	1432.55
18	RandomForest	33220	1661	20	398.24	33220	1	33220	398.24
19	PowerEN	14109	1000	48	61.02	12194	62	357	30.02
20	Protomata	42011	2340	123	1578.51	38243	513	3745	594.68

Table 1: Benchmark Characteristics

	L_switch [L]				G_switch(1 way) [G1]				G_switch(4 ways) [G2]			
Design	Size	Delay	Energy	Area	Size	Delay	Energy	Area	Size	Delay	Energy	Area
CA_P	280x256	163.5ps	0.191pJ/bit	0.033mm^2	128x128	128ps	0.16pJ/bit	0.011mm^2	-	-	-	-
	Number of switches				Number of switches				Number of switches			
	64				8				-			
CA_S	280x256	163.5ps	0.191pJ/bit	0.033mm^2	256x256	163ps	0.19pJ/bit	0.032mm^2	512x512	327ps	0.381pJ/bit	0.1293mm^2
	Number of switches				Number of switches				Number of switches			
	128				8				1			

Table 2: Switch Parameters

V. Our 8T crossbar switches are similar to an 8T SRAM array, except without the associated decoding and control logic overheads present in a regular 8T SRAM array. The energy for access to 6T 256 × 256 cache sub-arrays was estimated to be $22pJ$. The global wire delays were determined using wire models from the design kit using SPICE modeling. Our analysis takes into account cross-coupling capacitance of neighboring wires and metal layers. The global wires have pitch 1μm and are routed on $4X$ metal layers with double track assignment and repeaters spaced $1mm$ apart. The wire delay was found to be $66ps/mm$ and wire energy was found to be $0.07pJ/mm/bit$.

5 RESULTS

In this section we first present the speedups obtained by the proposed Cache Automaton (CA) architecture, followed by analysis of cache space utilization, energy consumption and reachability of the proposed automaton architecture. As discussed in Section 3.1, we evaluate two designs for Cache Automaton. The first design is optimized for performance, and provides lower connectivity. We refer to the performance optimized design as *CA_P* and space optimized design as *CA_S* throughout the results section.

5.1 Overall Performance

Overall performance of the *Cache Automaton* is dictated by the clock-period of pipeline. Table 3 shows the delay of various pipeline stages across both performance optimized and space optimized designs.

For the performance optimized design (*CA_P*), the state-match stage accesses 256 STEs. In our proposed architecture modelled after Xeon's LLC slice, each 16KB data sub-array is 8-way multiplexed. Internally the sub-array is organized into two 8KB chunks which can operate independently. Each chunk has two halves: Array_H and Array_L which share I/O and 32 sense-amps. Each half consists of 256 × 128 6T SRAM arrays. A column multiplexer in each half feeds 32 sense-amps, allowing only 32 bits to be read in a cycle per chunk. Thus, together across the two chunks, it is possible to match 64 STEs in a cycle. The SRAM arrays can operate from 1.2 GHz to 4.6 GHz frequency range [10, 19]. We limit the highest possible operating frequency for each SRAM array to 4 GHz or 256 ps cycle time. Thus four cycles or 1024ps is necessary to match 256 STEs without sense-amplifier cycling. With our proposed sense-amplifier cycling optimization, the state match time for 256 STEs is 438ps as shown in Table 3.

Design	State-Match	G-Switch	L-Switch	Freq. Max	Freq. Operated
CA_P	438 ps	227 ps	263 ps	2.3 GHz	**2 GHz**
CA_S	687 ps	468 ps	304 ps	1.4 GHz	**1.2 GHz**

Table 3: Pipeline stage delays and operating frequency.

The G-Switch stage requires 227ps composed of 99ps due to wire-delay and 128ps due to global switch. The distance between SRAM array and global switch is estimated to be 1.5mm assuming a slice dimension of $3.19mm \times 3mm$. The L-Switch stage requires 263ps. The pipeline clock period or frequency is determined by the slowest stage. Thus the maximum possible frequency for (*CA_P*) is 2.2 GHz. We choose to operate at 2 GHz. For the space optimized design (*CA_S*), an operating frequency of 1.4 GHz can be achieved and we choose to operate at 1.2 GHz. The space optimized design is slower due to longer wire delays between the arrays and global switch, and larger global switches.

In the Cache Automaton, since all state-matches and state-transitions can happen in parallel, the system has a *deterministic throughput of one input symbol per cycle and is independent of input benchmarks*. This is true for Micron's **Automata Processor (AP)** as well which

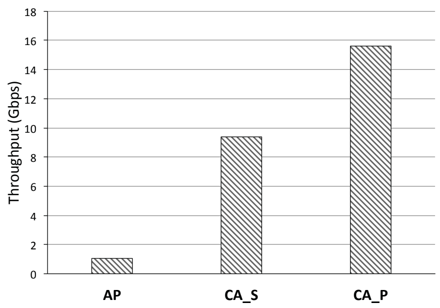


Figure 7: Overall performance of Cache Automaton compared to Micron's Automata Processor in Gb/s.

operates at **133 MHz** frequency. Figure 7 shows the overall achieved throughput of Cache Automaton in Gb/s across all benchmarks. Overall, the performance optimized design provides a speedup of 15× over Micron's AP. Prior studies for same set of benchmarks have shown 256× speedup over conventional x86 CPU [39], thus the Cache Automaton provides a 3840× speedup over processing in CPU. Our space optimized design provides a speedup of 9× over AP.

5.2 Cache Utilization

Figure 8 shows the cache utilization in MB for different applications considering both the *CA_P* and *CA_S* designs.

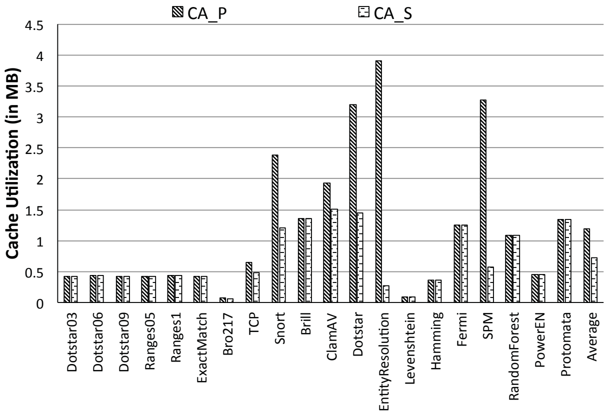


Figure 8: Cache utilization of benchmarks for the two evaluated designs of Cache Automaton.

The *CA_S* design shows large space savings for Entity Resolution (3.64 MB), SPM (2.7 MB), Dotstar (1.76 MB) and Snort (1.19 MB). The savings achieved compared to *CA_P* are proportional to the redundant state activity in each of the benchmarks. SPM in particular benefits from merging several start states. Although Entity Resolution in the *CA_S* design has only 5672 states, it has high routing complexity with a high average out-degree (> 6 per FSM state). Benchmarks from *Regex* have small connected components and do not show much benefit from *prefix merging*. RandomForest and Fermi perform a large number of distinct pattern matches and subsequently show lesser state-redundancy with a high number of active states per cycle. These benchmarks show little to no benefit when compared to the *CA_P* design. It must be kept in mind that these space savings can be directly translated to speedup by matching against multiple NFA instances. Also, the *space-optimized* automata tend to have a lower average active set, on account of lesser redundant state activity, leading to dynamic energy savings. Averaged across all benchmarks, we see that the *CA_P* and *CA_S* designs utilize 1.2 MB and 0.725 MB cache space respectively.

5.3 Energy Consumption

This section discusses the energy consumption and power consumption of Cache Automaton. The energy consumption of Cache Automaton depends on two factors. *First*, number of active partitions. Note that even if one STE is active in a partition, it results in an array access and local switch access. *Second*, number of dynamic transitions between partitions, because these result in global switch accesses and expend wire energy. Both these factors are controlled by the mapping algorithms used by the compiler. By grouping STEs based on connected components the number of active partitions are drastically reduced. By adopting graph partitioning techniques, our compiler successfully reduces number of transitions between partitions.

Since there is no publicly available data regarding AP's energy consumption, we use an *Ideal AP* energy model which assumes *zero energy for interconnects, routing matrix, and an optimistic 1 pJ/bit for DRAM array access energy*. Conventional DRAMs have been reported to consume anywhere between 2.5pJ/bit to 10 pJ/bit for array access energy (activation energy) [9, 31].

Figure 9 shows the energy expended per input symbol for performance optimized (*CA_P*) and space optimized (*CA_S*) Cache Automaton designs, as well as for *Ideal AP* with same mappings as used by Cache Automaton. Several observations can be made. Benchmarks with higher active state set (see Table 1) such as Entity Resolution, SPM, Fermi consume higher energy. These benchmarks also utilize global switches more frequently than other benchmarks. The *CA_S* mapping has consistently lower energy consumption than *CA_P* both for Cache Automaton and AP. This is because *CA_S* mapping merges many redundant states and thus wastes lesser energy per input symbol on redundant transitions. On an average Cache Automaton (*CA_P*, *CA_S*) consumes 3× lesser energy than *Ideal* AP with same mapping (Ideal AP w/*CA_S*) . The lowest energy consumption of 2.3nJ/symbol is obtained by *CA_S* design, which is 3.1× better than best configuration for AP assuming an ideal energy model. Thus for systems which are energy constrained, we recommend a space optimized mapping. Similar to the Micron AP [27], we also employ partition disabling circuits triggered when there is no active state within a partition, detected using a simple wired OR of all the bits in the active state vector.

Figure 9 shows the average power consumption across benchmarks. The power consumption follows the general trends of energy consumption. As expected the power consumption of Cache Automaton is higher, but much lower than TDP of the processor at 160W (Xeon E5-2600 v3). Thus we do not expect Cache Automaton to create any power overdrive or thermal problems. Our prototype Cache Automaton which supports NFA processing only in 8 ways of a cache slice can consume a maximum power of 75W and has capacity to store 128K STEs. Note, *CA_S* and *CA_P* have a maximum

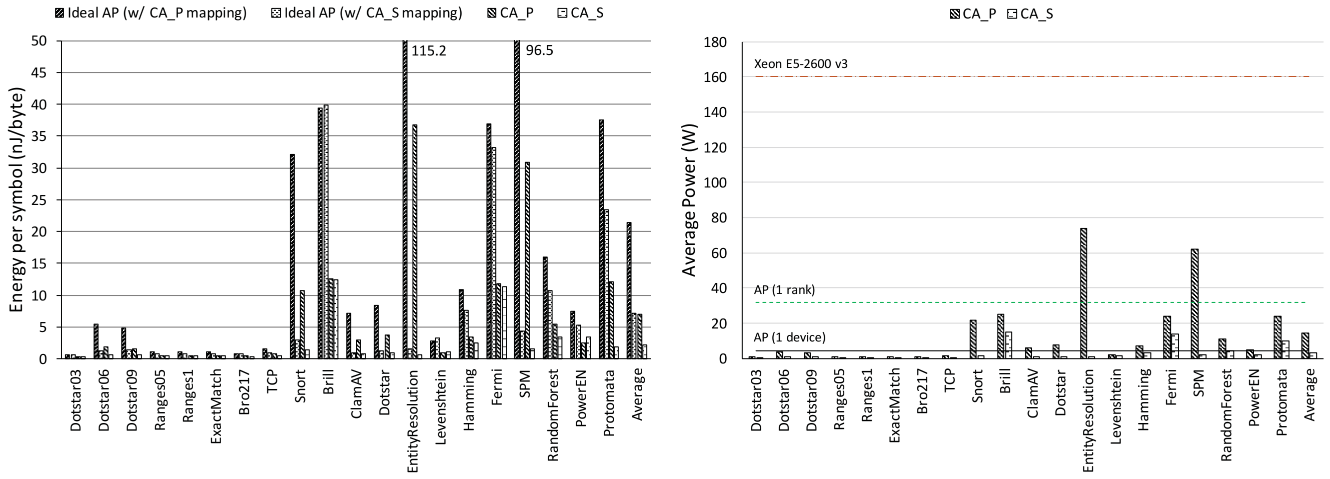


Figure 9: (a) Overall energy consumption of Cache Automaton compared to *Ideal* Automata Processor. (b) Overall power consumption of the two evaluated designs of Cache Automaton.

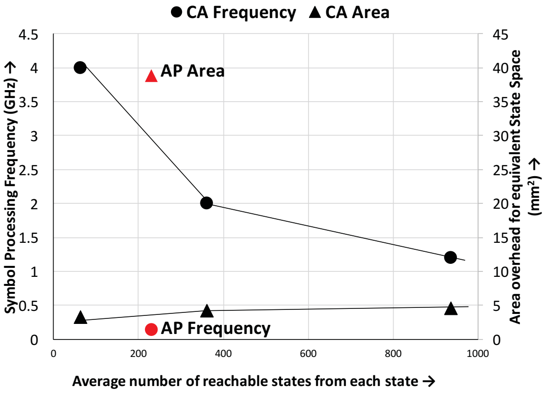


Figure 10: Figure shows the performance (frequency for symbol processing), reachability, and area overheads of various Cache Automaton (CA) designs and DRAM-based Automaton Processor (AP).

power consumption of 14.9W and 71.3W. Thus, a system designer can trade-off between performance and power.

5.4 Reachability and Area Overheads

Memory centric models such as Micron's AP and proposed Cache Automaton architecture do all state transitions for an input symbol in parallel. Thus an important parameter which determines the performance of these architectures and is representative of their scalability to accommodate complex NFAs is the average reachability of a state. Micron's AP provides an average reachability of 230.5 states from any state (Fan-out), while operating at 133 MHz. Figure 10 plots the frequency of Cache Automata (left Y-axis) with respect to reachability. A highly performance optimized design can operate at 4 GHz, but provide a small reachability of 64 states. Larger degree of reachability requires more and bigger global switches, hence has a performance penalty. Proposed *CA_P* can operate at 2 GHz while still providing a reachability of 361 states, which is 1.5× better than AP. Proposed *CA_S* can operate at 1.2 GHz while providing reachability of 936 states. Note that Cache Automaton supports a maximum of 256 incoming transitions per state (Fan-in), in contrast to only 16 supported by AP.

Figure 10 also plots the area overhead of Cache Automaton (right Y-axis) with respect to reachability. Area overhead is reported for supporting state space equivalent to 32K STEs. Proposed *CA_P* and *CA_S* designs incur a modest area overhead of 4.3mm^2 and 4.6mm^2 (less than 2% of die area for Xeon E5 server processor which has area of 354mm^2), but offer high reachability and high performance. In comparison, AP incurs a high area overhead of 38mm^2 for supporting transition matrix in DRAM dies.

5.5 Discussion

This section discusses the impact of various optimizations such as sense-amplifier cycling and parameters such as wire delays. Table 4 column w/o SA cycling shows the frequency of Cache Automaton without sense-amplifier cycling. It can be noted that the pipeline can still operate up to 1 GHz frequency without this optimization. Another alternative to boost frequency without sense-amplifier cycling is to under utilize the cache space and read fewer column multiplexed bits in each cycle.

The proposed designs use global metal layers for connecting local and global switches. This is motivated by two factors. First, global metal layers are much faster. Second, global metal layers are used only for on-chip networks and are usually underutilized. It is also possible to reuse the wires of hierarchical bus (H-Bus) or H-Tree interconnects used inside a LLC slice. However, these interconnects are much slower (300 ps/mm [12]). Table 4 column with H-Bus shows the frequency of Cache Automaton when reusing wires of H-Bus interconnect within a LLC slice. The operational frequency is still 7.5×-11× better than AP.

Design	Achieved	w/o SA cycling	with H-Bus
CA_P	2 GHz	1 GHz	1.5 GHz
CA_S	1.2 GHz	500 MHz	1 GHz

Table 4: Impact of optimizations and parameters.

5.6 Comparison with ASIC implementations

The Unified Automata Processor (UAP) [15] and HARE [17] are two recently proposed accelerators that have demonstrated impressive line rates for automata processing and regular expression matching on a number of network intrusion detection and log processing benchmarks. UAP is noteworthy because of its generality and ability to efficiently support many finite automata models using state transition packing and multi-stream processing at low area and power costs. Similarly, HARE has been able to saturate DRAM bandwidth (256 Gbps) while scanning up to 16 regular expressions.

The major advantage of Cache Automaton is its ability to execute several thousand state transitions in parallel (e.g., 128K state transitions in a single cycle using 8 ways of LLC slice). This massive parallelism enables matching against several thousand patterns (e.g., 5700 in Snort ruleset), while achieving ideal line rate, i.e., 1 symbol/cycle. In contrast, HARE incurs high area and power costs ($80mm^2$,125W) when scanning for more than 16 patterns and UAP's line rate drops for large NFA patterns with many concurrent activations, 0.27–0.75 symbols/cycle [15].

Metric	HARE (W=32)	UAP	CA_P	CA_S
Throughput (Gbps)	3.9	5.3	15.6	9.4
Runtime (ms)	20.48	15.83	5.24	8.74
Power (W)	125	0.507	7.72	1.08
Energy (nJ/byte)	256	0.802	4.04	0.94
Area (mm^2)	80	5.67	4.3	4.6

Table 5: Comparison with related ASIC designs.

For fair quantitative comparison, we use Dotstar0.9, containing 1000 regular expressions and ~38K states as used in UAP/HARE for a 10MB input stream. It must be noted that CA can support >3000 such regular expressions using less than 8 ways of LLC slice and shows greater benefits for larger number of patterns and if more ways are used to store NFA. From Table 5, it can be seen that *CA_P* and *CA_S* provide 3.9× and 2.34× speedup over HARE and 3× and 1.8× speedup over UAP respectively. While UAP is more energy efficient than *CA_P* due to efficient compression of state-transitions, *CA_S* can provide comparable energy efficiency while repurposing the LLC. Note that the 16 kB local memory in UAP can accommodate only few Dotstar0.9 patterns without memory sharing across lanes and we expect several additional DRAM accesses for reading new patterns. This energy is not accounted in Table 5. UAP incurs lesser area overhead than *CA_P* and *CA_S*, but its 8-entry combining queue may be insufficient to support benchmarks with several thousand active states (e.g., Fermi–4715).

6 RELATED WORK

To the best of our knowledge, this is the first work that demonstrates the feasibility of in-situ FSM processing in the last-level cache. Below we discuss some of the closely related works.

Compute-Centric Architectures: Compute-centric architectures typically store the complete state-transition matrix as a lookup table in cache/memory. These architectures have two main limitations: (1) need for high memory bandwidth or memory capacity especially for large NFA with many active states and (2) high instruction processing overheads per state transition (as many as 24 *x86* instructions for a single DFA state transition [6]). As a result, several CPU/GPU-based automata processing engines have either limited themselves to DFAs [3, 6, 45, 46] or have explored SIMD operations to deal with the high cache miss rates and branch misprediction rates [26, 30]. Several speculative and enumerative parallelization approaches have also been proposed to speedup FSM processing [21, 30, 33, 47, 48]. However, all of the above approaches have been evaluated only for small DFA. Scaling these approaches to NFAs is non-trivial because of the huge computational complexity involved [43].

ASIC implementations: While several regular expression matching and NFA processing ASIC designs have been proposed in literature [37, 38], we extensively discuss and evaluate two most relevant and recent designs, HARE [17] and the Unified Automata Processor (UAP) [15] in Section 5.6. In general, while ASIC implementations offer high line rates for small DFA/NFA, they are limited by the number of parallel matches and state transitions.

Memory-Centric Architectures: Memory-centric architectures like the Micron Automata Processor (AP) [14, 40] leverage the inherent bit-level parallelism of DRAM to support multiple parallel state matches at bandwidths that far exceed available pin bandwidth with reduced instruction processing overheads. A programmable routing matrix enables efficient support for state-transitions without the associated data movement costs as in a compute-centric architecture. Since the DRAM AP is not optimized for logic, it is slower and runs at a low frequency (133 MHz), with the routing matrix accounting for nearly 30% of the die area. In this work we demonstrate that several real world NFA can be efficiently mapped onto LLC arrays and propose two fully-pipelined designs that provide nearly an order of magnitude throughput improvement compared to AP at low area overheads. Similar to the Micron AP whose throughput scales with the number of ranks and devices, the *Cache Automaton* architecture can also exploit increased cache capacity in multi-chip/multi-socket servers for throughput scaling based on the peak power requirements of different NFAs.

7 CONCLUSION

This paper proposes the *Cache Automaton* architecture to accelerate NFA processing in the last-level cache. Efficient NFA processing requires both highly parallel state-matches as well as an interconnect architecture that supports low-latency transitions and rich connectivity between states. To optimize for state-matches we propose a sense-amplifier cycling scheme that exploits spatial locality in state-matches. To enable efficient state transitions, we adopt a hierarchical topology of highly compact 8T-based local and global switches. We also develop a *Cache Automaton* compiler that fully automates the process of mapping NFA states to SRAM arrays. The two proposed designs are fully pipelined, utilize on an average 1*MB* of cache space across benchmarks and can provide a speedup of 12× over AP .

ACKNOWLEDGMENTS

We thank the anonymous reviewers and fellow members of the M-Bits research group for their feedback which greatly helped improve this work. This work was supported by the NSF under the CAREER-1652294 award.

REFERENCES

[1] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (June 1975), 333–340.

[2] Rajeev Alur and Mihalis Yannakakis. 1998. Model checking of hierarchical state machines. In *ACM SIGSOFT Software Engineering Notes*, Vol. 23. ACM, 175–188.

[3] Michela Becchi and Patrick Crowley. 2007. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM, 145–154.

[4] Michela Becchi and Patrick Crowley. 2008. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 2008 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2008, San Jose, California, USA, November 6-7, 2008*. 50–59.

[5] Michela Becchi, Mark A. Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*. 79–89.

[6] Michela Becchi, Charlie Wiseman, and Patrick Crowley. 2009. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 30–39.

[7] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. 2015. Entity Resolution Acceleration using MicronâĂŹs Automata Processor. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[8] William J. Bowhill, Blaine A. Stackhouse, Nevine Nassif, Zibing Yang, Arvind Raghavan, Oscar Mendoza, Charles Morganti, Chris Houghton, Dan Krueger, Olivier Franza, Jayen Desai, Jason Crop, Brian Brock, Dave Bradley, Chris Bostak, Sal Bhimji, and Matt Becker. 2016. The Xeon® Processor E5-2600 v3: a 22 nm 18-Core Product Family. *J. Solid-State Circuits* 51, 1 (2016), 92–104. https://doi.org/10.1109/JSSC.2015.2472598

[9] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R. Johnson, Stephen W. Keckler, Minsoo Rhu, and William J. Dally. 2017. Architecting an Energy-Efficient DRAM System for GPUs. In *High Performance Computer Architecture (HPCA)*.

[10] Wei Chen, Szu-Liang Chen, Siufu Chiu, Raghuraman Ganesan, Venkata Lukka, Wei Wing Mar, and Stefan Rusu. 2013. A 22nm 2.5 MB slice on-die L3 cache for the next generation Xeon® processor. In *VLSI Technology (VLSIT), 2013 Symposium on*. IEEE, C132–C133.

[11] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*. Springer, 359–364.

[12] Subhasis Das, Tor M. Aamodt, and William J. Dally. 2015. SLIP: reducing wire energy in the memory hierarchy. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. 349–361. https://doi.org/10.1145/2749469.2750398

[13] Sutapa Datta and Subhasis Mukhopadhyay. 2015. A grammar inference approach for predicting kinase specific phosphorylation sites. *PloS one* 10, 4 (2015), e0122294.

[14] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098.

[15] Yuanwei Fang, Tung Thanh Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. 533–545. https://doi.org/10.1145/2830772.2830809

[16] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. 2008. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review* 38, 5 (2008), 29–40.

[17] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. 1–12. https://doi.org/10.1109/MICRO.2016.7783747

[18] Linley Gwennap. 2014. Micron Accelerates Automata:New Chip Speeds NFA Processing Using DRAM Architectures. In *Microprocessor Report*.

[19] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. 2013. An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel® Xeon® Processor E5 Family. *J. Solid-State Circuits* 48, 8 (2013), 1954–1962. https://doi.org/10.1109/JSSC.2013.2258815

[20] Intel. 2017. Cache Allocation Technology. (2017). https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology

[21] Peng Jiang and Gagan Agrawal. 2017. Combining SIMD and Many/Multi-core Parallelism for Finite State Machines with Enumerative Speculation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*. 179–191. http://dl.acm.org/citation.cfm?id=3018760

[22] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. 2009. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*.

[23] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[24] Linux kernel. 2017. Huge Pages. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[25] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, Vol. 36. ACM, 339–350.

[26] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. 2012. Parabix: Boosting the efficiency of text processing on commodity processors. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*. 373–384. https://doi.org/10.1109/HPCA.2012.6169041

[27] Micron. 2016. Method and system to dynamically power-down a block of a pattern-recognition processor. (2016). Patent US 9389833 B2.

[28] Micron. 2017. Micron Automata Processing. http://www.micronautomata.com/

[29] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. 2014. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 309–328.

[30] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 529–542.

[31] Omar Naji, Christian Weis, Matthias Jung, Norbert Wehn, and Andreas Hansson. 2015. A high-level DRAM timing, power and area exploration tool. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE, 149–156.

[32] Alexandre Petrenko. 2001. Fault model-driven test derivation from finite state models: Annotated bibliography. In *Modeling and verification of parallel processes*. Springer, 196–205.

[33] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. MicroSpec: Speculation-Centric Fine-Grained Parallelization for FSM Computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*. 221–233.

[34] Indranil Roy and Srinivas Aluru. 2016. Discovering motifs in biological sequences using the micron automata processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 13, 1 (2016), 99–111.

[35] Indranil Roy, Ankit Srivastava, and Srinivas Aluru. 2016. Programming Techniques for the Automata Processor. In *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*. 205–210. https://doi.org/10.1109/ICPP.2016.30

[36] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. 2016. High Performance Pattern Matching Using the Automata Processor. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. 1123–1132. https://doi.org/10.1109/IPDPS.2016.94

[37] Prateek Tandon, Faissal M. Sleiman, Michael J. Cafarella, and Thomas F. Wenisch. 2016. HAWK: Hardware support for unstructured log processing. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 469–480. https://doi.org/10.1109/ICDE.2016.7498263

[38] 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 *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*. 461–472. https://doi.org/10.1109/MICRO.2012.49

[39] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*. 105–166.

[40] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy II, Jack Wadden, Mircea R. Stan, and Kevin Skadron. 2016. An overview of micron's automata processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. 14:1–14:3. https://doi.org/10.1145/2968456.2976763

[41] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential pattern mining with the Micron automata processor. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 135–144.

[42] Zhen-Gang Wang, Johann Elbaz, Françoise Remacle, RD Levine, and Itamar Willner. 2010. All-DNA finite-state automata with finite memory. *Proceedings of the National Academy of Sciences* 107, 51 (2010), 21996–22001.

[43] Yi-Hua E. Yang and Viktor K. Prasanna. 2011. Optimizing Regular Expression Matching with SR-NFA on Multi-Core Systems. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*. 424–433. https://doi.org/10.1109/PACT.2011.73

[44] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. ACM, 93–102.

[45] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*. IEEE, 93–102.

[46] Xiaodong Yu, Bill Lin, and Michela Becchi. 2014. Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence. *IEEE Journal on Selected Areas in Communications* 32, 10 (2014), 1822–1833.

[47] Zhijia Zhao and Xipeng Shen. 2015. On-the-Fly Principled Speculation for FSM Parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 619–630.

[48] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "embarrassingly sequential": parallelizing finite state machine-based computations through principled speculation. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 543–558.

[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.