Sunder: Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration

Elaheh Sadredini
University of California, Riverside
elaheh@cs.ucr.edu

Reza Rahimi
University of Virginia
rahimi@virginia.edu

Mohsen Imani
University of California, Irvine
m.imani@uci.edu

Kevin Skadron
University of Virginia
skadron@virginia.edu

ABSTRACT

Automata processing is an efficient computation model for regular expressions and other forms of sophisticated pattern matching. The demand for high-throughput and real-time pattern matching in many applications, including network intrusion detection and spam filters, has motivated several in-memory architectures for automata processing. Existing in-memory architectures focus on accelerating the pattern-matching kernel, but either fail to support a practical reporting solution or optimistically assume that the reporting stage is not the performance bottleneck. However, gathering and processing the reports can be the major bottleneck, especially when the reporting frequency is high. Moreover, all the existing in-memory architectures work with a fixed processing rate (mostly 8-bit/cycle), and they do not adjust the input consumption rate based on the properties of the applications, which can lead to throughput and capacity loss.

To address these issues, we present Sunder, an in-SRAM pattern matching architecture, to processes a reconfigurable number of nibbles (4-bit symbols) in parallel, instead of fixed-rate processing, by adopting an algorithm/architecture methodology to perform hardware-aware transformations. Inspired by prior work, we transform the commonly-used 8-bit processing to nibble-processing (4-bit processing) to reduce hardware requirements exponentially and achieve higher information density. This frees up space for storing reporting data *in place*, which significantly eliminates host communication and reporting overhead. Our proposed reporting architecture supports in-place report summarization and provides an easy access mechanism to read the reporting data. As a result, Sunder enables a low-overhead, high-performance, and flexible in-memory pattern-matching and reporting solution. Our results confirm that Sunder reporting architecture has *zero* performance overhead for 95% of the applications and incurs only 2% additional hardware overhead.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480934

CCS CONCEPTS

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

KEYWORDS

Automata processing, hardware accelerator, in-SRAM processing, near-data computing, pattern matching, reconfigurable computing

ACM Reference Format:
Elaheh Sadredini, Reza Rahimi, Mohsen Imani, and Kevin Skadron. 2021. Sunder: Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480934

1 INTRODUCTION

Pattern-based algorithms (pattern matching, pattern recognition, etc.) are exceedingly common in network security [3, 7, 15, 23, 46, 64, 65], bioinformatics [8, 16, 31, 38, 50], data mining [10, 45, 51, 58, 59], machine learning [50], natural language processing [39, 66], verification [52], and many other application domains [57]. Patterns from these applications are often massive in number, complex in structure, dynamic in behavior, and need to support a variety of inexact matches. Besides, such applications are getting pushed further into real-time scenarios (e.g., network processing), and in many cases, sophisticated processing must be done in edge devices. However, pattern matching is a memory-bound task, and off-the-shelf von Neumann architectures struggle to meet today's big-data and streaming line-rate processing requirements.

One leading methodology for inexact pattern matching is to use regular expressions to identify these complex patterns. Regular expressions are a widely used subset of pattern specification language, and they are efficiently implemented via Finite Automata (FA) [22]. To address the memory-wall challenges [61], in-memory architectures for automata processing have been introduced to benefit from the massive internal memory bandwidth by performing massively-parallel symbol matching using memory arrays [17, 41, 44, 48]. They all support the execution of Non-deterministic Finite Automata (NFA) by providing a reconfigurable infrastructure to implement finite automata in memory arrays. The massive bit-level parallelism of memory arrays allows a large number of state machines to be executed in parallel, leveraging the high density of memory arrays. If device capacity is not enough for an application, either more hardware units or multiple rounds of reconfigurations are required.

Many studies have shown in-memory automata processing provides remarkable speedup over the existing software approaches, FPGA solutions, and regex accelerators on a wide range of applications [8, 10, 37–39, 45, 48, 50, 58, 59, 66].

In-memory automata processing has three processing stages; state matching, state transition, and report gathering, and these can be combined in a pipelined fashion. In the state-matching stage, the current input symbol is decoded and all the states whose symbols match against it are detected by reading the fetched memory row. In the state-transition stage, active states' successors are determined by propagating activation signals via a programmable interconnect. In the report-gathering phase, the report data are accumulated and eventually analyzed for the final action or decision.

Prior in-memory automata accelerators have mostly overlooked the real cost of reporting stage and optimistically assumed that reporting is not a bottleneck [2, 32, 41, 48], thus, only evaluated the first two stages (i.e., the kernel). However, reporting incurs a significant cost when it is considered precisely [42]. For example, the reporting architecture in the Micron's Automata Processor (AP) [17] has 40% area overhead [21] and up to 46× performance overhead due to stalls and host communications [55]. To improve the AP reporting architecture, Wadden et al. [55] propose finer-grain reporting buffers to reduce the report vector sparsity. However, their approach (1) needs to store relatively large metadata, (2) requires more complex peripherals to connect the smaller report buffers, (3) causes frequent stalls in the execution because the reporting queue gets filled quickly when an application reports frequently, and (4) does not have any control to partially select or summarize the report data when needed; all of which make their solution inefficient and hard to scale.

To address these issues, we propose Sunder, a highly reconfigurable in-SRAM automata processing design with a flexible, compact, simple, and low-overhead memory-mapped reporting architecture. Inspired by Impala [41], we first transform the common, fixed 8-bit automata processing rate (which requires 2^8 memory rows) to a multiple of 4-bit automata or *nibble processing* (which requires groups of 2^4 memory rows). This can greatly reduce the required number of rows in each subarray in memory-based automata processing solutions. We opportunistically utilize the saved memory rows in the state matching subarrays to store the reporting data locally and densely in the same subarray as state-matching symbols. The low-overhead benefits are achieved by re-using existing memory rows and peripherals in state-matching subarrays, which translates to area saving. This, in fact, provides exclusive in-place reporting buffers for each automaton and avoids shared bus conflicts (from state-matching array to report buffers); thus, reducing the number of stalls and achieving performance benefits.

Overall, our reporting architecture (1) greatly eliminates data movement and stalls due to reporting (zero stalls for 95% of the applications), (2) reuses existing peripheral and circuitry in state matching subarrays, thus, has minimal hardware overhead (less than 2%), (3) provides an easy and flexible mechanism for the host to analyze or summarize any portion of reporting data at any time, and (4) efficiently supports both sparse and dense reporting behavior.

In addition, in contrast to all prior work, Sunder presents a reconfigurable symbol processing rate for automata processing (i.e., 4-bit, 8-bit, and 16-bit symbols per cycle), which enables throughput and density benefits for a diverse set of applications. This is facilitated by packing multiple nibble processing in one memory subarray, which allows for easy and low-overhead reconfiguration. This is unlike Impala [41], where the processing rate is fixed in hardware and multiple subarrays are used in parallel, which requires additional hardware for aggregating the final results.

Liu et al. [32] demonstrated that not all the states in an NFA are enabled during execution; thus do not need to be configured on the hardware. This reduces the hardware resources for an automaton on the in-memory automata accelerators (by splitting an automaton between the CPU and the AP), which improves the performance when the application is very large and would require several rounds of reconfigurations. However, this approach generates more intermediate results (or reports), which need to be transferred to the CPU. Our proposed reporting architecture is complementary to their technique and can significantly improve reporting efficiency when larger intermediate reports are generated.

In summary, the paper makes the following **contributions**:

- We present a compact, simple, low overhead, and localized memory-mapped reporting architecture for in-memory automata processing to significantly reduce data movement and host communication. Our key insight is that nibble processing, enabled by prior work [41], also enables reports to be buffered directly in the same arrays that perform matching, thus significantly minimizing stalls due to full report buffers.
- We present a low-overhead reconfigurable processing rate in hardware for in-memory automata processing, by processing multiple nibbles in one memory array and accumulating the partial results with multi-row activation in memory. This enables throughput vs. density trade-off across a set of diverse applications.
- For the same state density, Sunder provides 9× larger reporting buffer than the Micron's Automata Processor and at the same time, has 2.2× lower overall area overhead in the same technology size. Moreover, on average, Sunder provides 280× higher throughput compared to the Automate Processor and 4-8× higher throughput compared to the state-of-the-art SRAM-based solutions (Impala [41] and Cache Automaton [48]), assuming an AP-style reporting architecture.

2 BACKGROUND AND MOTIVATION

2.1 In-memory Automata Processing

This section presents an abstract architecture for processing one symbol per cycle in common in-memory automata architectures. Generally, automata processing has three main stages for each input symbol, *state match*, *state transition*, and *reporting*.

In Figure 1, an example of classic and homogeneous NFA is presented. In a homogeneous NFA (STE: State Transition Element), all transitions entering a state must happen on the same input symbol [19]. For example, edges entering STE_3 happen if the input symbol is G. This property maps well with the in-memory implementation that finds matching states in one clock cycle and allows a label-independent interconnect.

STE_0-STE_3 on the homogeneous example are one-hot encoded in four memory columns. In the *state matching* phase, the current

input symbol is decoded, and the set of states whose rule (label) matches that input symbol is recognized through reading a row of memory (*match vector*). Then, the set of *potential next states* is combined with the *match vector*, which represents the set of currently *active states* that can initiate state transitions (i.e., these two vectors are ANDed). During the *state-transition* phase, the *potential next states* (to be activated for the next cycle) are detected by propagating signals from the *active state vector* through the interconnect module to update the active state vector.

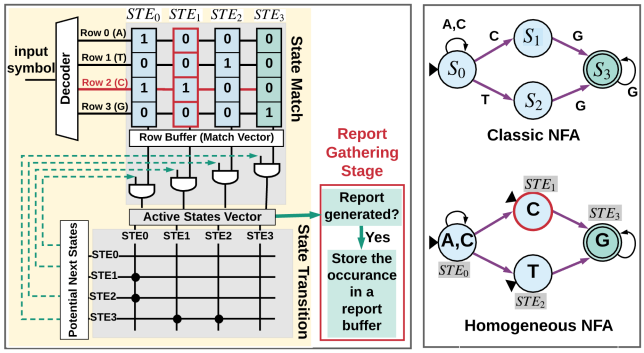

Figure 1: (left) In-memory automata processing model, (right) Classic NFA and it equivalent homogeneous NFA.

In the example, the STE_1 matching symbol is C, and the corresponding position in the memory column encoding STE_1 is set to '1' (i.e., in the third row). Because our example has four symbols (i.e., A, T, C, and G), only four memory rows are enough for one-hot encoding of symbols. Assume STE_0 is a current active state. The potential next cycle active states are the states connected to STE_0, which are STE_0, STE_1, and STE_2. Assume the input symbol is 'C', then, the third row (Row2) is read into the row buffer (*match vector*). Bitwise AND on the *match vector* (i.e., 1100) and *potential next states* (1110) determines STE_0 and STE_1 as the next active states.

Automata Reporting Stage: STE_3 is the reporting state. In Figure 1, every time STE_3 is set to '1', this indicates a matching has occurred, and a report is generated, thus, the occurrence of the report along with the report cycle information (i.e., metadata) should be captured in a report buffer. For example, in real applications, when a malicious network packet is matched with the intrusion detection rules (and rules are represented by NFAs), a report will be generated in the system. In in-memory automata processing architectures, many memory subarrays are working in parallel, each processing one or a few NFAs, and they all can potentially generate reporting data every cycle. For example, SPM [58] can generate up to 1394 reports per report-cycle - see Table 1. This causes many stalls in the processing and negatively affect the performance. Theoretically, each *STE* can be a reporting state; thus, a sparse or dense report vector from each subarray is generated (each time there is a least one report) and will be sent to the host for further processing.

2.2 Existing Reporting Architectures

Existing in-memory automata processing architectures either (I) optimistically assume that reporting is not a performance bottleneck [41, 43, 48], as they are focusing on improving the state matching and interconnect stages (i.e., overall throughput is determined by $\frac{frequency \times (\#bits/cycle)}{reporting-overhead}$ and prior work calculates $frequency \times (\#bits/cycle)$ as the overall performance), or (II) they fail to support a low-overhead and scalable reporting solution [17, 55].

Micron's Automata Processor (AP) [17] uses a hierarchical reporting architecture with two levels of buffers for offloading reporting state bits. At the system level, the AP contains 32 D480 chips. Each D480 chip (Fig. 2) contains two independent half-cores that have independent automata states and edges. Each half-core has three separate reporting regions, where each reporting region is responsible for a maximum of 1024 reporting *STEs*. Each reporting STE is routed to one if these three reporting regions.

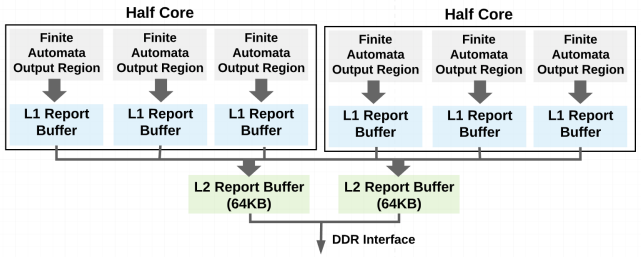

Figure 2: The Automata Processor reporting architecture.

At runtime, if *any* of the 1024 reporting bits are activated, a full 1024-bit vector and 64-bit metadata, are offloaded to the L1 storage buffer assigned to the triggered report state as shown in Figure 2. When an L1 storage buffer is full, its content is offloaded to one of two L2 buffers shared with the other half-core for eventual export off-chip. This architecture must stall during this offloading process because it does not support simultaneous push and pop operations. These L2 buffers are then transferred to the host. The AP reporting architecture can efficiently handle dense reporting, but it is very ineffective when reporting is sparse and incurs a significant performance overhead (up to 46× [55]).

To address the high cost of sparse reporting workloads on the AP, Wadden et al. [55] introduce a reporting compression scheme, called Report Aggregator Division (RAD), to break up a large output report vector into smaller chunks in spatial automata architectures. This approach reduces the amount of data sent from the spatial accelerator to the host by offloading subsections of the report vector that contain report bits, thus, improves the stall rate by using fine-grained report vectors. However, their solution has several drawbacks: (1) the cost of metadata is increased, as each smaller packet needs its own metadata; (2) similar to the AP, it still uses the centralized and shared reporting buffers with its negative impact on routing complexity and high propagation delay of reporting signals from the report states to the buffers; (3) it does not provide any summarization functionality in hardware to reduce the off-chip communication for applications that do not need cycle-accurate report data; and (4) it does not improve the reporting overhead for dense reporting behavior (e.g., SPM).

The main contribution of this paper is designing an efficient and flexible reporting architecture to gather, transfer, and summarize reporting information at a very low cost for both dense and sparse reporting behaviour.

Table 1: Reporting behavior summary

Benchmark	Static Analysis				Dynamic Behaviour (Input Dependent)				
	#Family	#States	#Report States	#Report States/ States (%)	#Reports	#Report Cycles	#Reports/Cycles	#Reports/ Report Cycles	#Report Cycles/ #Cycles (%)
Brill [56]	Regex	42658	1962	4.6	1092388	118814	1.067	9.19	11.33%
Bro217 [6]	Regex	2312	187	8.1	17219	17210	0.017	1.00	1.64%
Dotstar03 [6]	Regex	12144	300	2.5	1	1	0.000	1.00	0%
Dotstar06 [6]	Regex	12640	300	2.4	2	2	0.000	1.00	0%
Dotstar09 [6]	Regex	12431	300	2.4	2	2	0.000	1.00	0%
ExactMatch [6]	Regex	12439	297	2.4	35	35	0.000	1.00	0%
PowerEN [56]	Regex	40513	3456	8.5	4304	4303	0.004	1.00	0.41 %
Protomata [56]	Regex	42009	2365	5.6	127413	105722	0.124	1.21	10.08%
Ranges05 [6]	Regex	12621	299	2.4	39	38	0.000	1.03	0%
Ranges1 [6]	Regex	12464	297	2.4	26	26	0.000	1.00	0%
Snort [56]	Regex	66466	4166	6.3	1710495	995011	1.670	1.72	94.89%
TCP [6]	Regex	19704	767	3.9	103415	103198	0.101	1.00	9.84%
ClamAV [56]	Regex	49538	515	1.0	0	0	0.000	0.00	0%
Hamming [56]	Mesh	11346	186	1.6	2	2	0.000	1.00	0%
Levenshtein [56]	Mesh	2784	96	3.4	4	4	0.000	1.00	0%
Fermi [56]	Widget	40783	2399	5.9	96127	13444	0.094	7.15	1.28%
RandomForest [56]	Widget	33220	1661	5.0	21310	3322	0.021	6.41	0.32%
SPM [56]	Widget	100500	5025	5.0	47304453	33933	46.19	1394	3.24%
EntityResolution [56]	Widget	95136	1000	1.1	37628	28612	0.037	1.32	2.73%

#Report States: the number of states that are designed to be the report states in the application.
#Reports: the total number of reports when streaming 1MB of input data.
#Report Cycles: the number of cycles that at least one report is generated.

2.3 Motivation

All previous memory-centric implementations for automata processing [2, 9, 17, 18, 32, 41, 43, 48, 62] suffer from three problems. First, they all work with a fixed (mostly 8-bit) symbol processing rate decided at design time. Second, they all have either failed in realizing an efficient report architecture design or overlook the real cost of reporting stage for dense reporting behaviour. Third, their rudimentary reporting architecture does not provide any support to summarize reporting data in hardware.

Processing rate: existing in-memory automata accelerators have a fixed processing rate set at design time—typically 8 bits [17, 43, 48]. Impala [41] investigates different processing rates (i.e., 4-bit, 8-bit, 16-bit); however, the processing rate is fixed at design time and is not reconfigurable. This limits maximum capacity and throughput utilization in a wide range of automata applications with different symbol-set size. For example, genomics applications usually have four symbols (i.e., A, T, C, and G), whereas data mining applications (such as SPM [58]) can have millions of unique symbols [40].

Reporting architecture issues: the reporting architecture is responsible for collecting per-cycle report information and storing them in a buffer temporarily to be transferred to the host. Designing such a hardware module is not straightforward because there are a few concerns that need to be considered. (I) Report states are generated in different memory arrays and need to be routed toward the global reporting buffers, potentially with high latency. (II) Choosing the right buffer bit-width is challenging due to its effect on area cost. A wide buffer solution (e.g., [17]) is attractive for an area-efficient design, as many report states are combined to create a single row of the report buffer, which results in smaller buffer control logic. However, a wide buffer can be more troublesome for applications with sparse and persistent reporting behavior, as the buffer gets filled up frequently, mostly with 0s. On the other hand, a narrow buffer solution (e.g., [55]) works effectively for applications with sparse reporting behavior and can physically be placed near where the report states are generated. Because buffers are narrow and their capacity is limited, we need many of them to cover all the report states. The cost of the control and access logic of the reporting buffers from the host becomes an issue as each needs to be controlled separately. *We believe the lack of a feasible and efficient reporting architecture in prior work is one of the main concerns of integrating an efficient automata processing accelerator in a system.*

Reporting strategy: None of the prior work on reporting architecture provides report summarization support, which can help to reduce the reporting I/O cost. Instead, they move the entire reporting data from the reporting buffers to the host and have the software to extract the information. For example, if an application only wants to know if a specific state has been triggered since the last time the report buffer was flushed, the host processor must first read all the reporting data of the buffer associated with that state and calculate the row-wise logical OR of the reporting cycles.

3 ANALYZING REPORTING BEHAVIOR

To motivate our efficient memory-mapped reporting architecture design, we first analyze the reporting behavior of a wide range of real-world automata benchmarks from ANMLZoo [56] and Regex [6] benchmark suites using their associated input streams. We use Virtual Automata Simulator (VASim) [54] to simulate the applications on the 1MB input stream provided with the benchmark suites and track all the reports throughout automata execution.

Table 1 shows a summary of the automata report statistics and behavior. *#States* represents the number of states in each application. *#Report States* shows the number of states labeled as reporting states in the application. As the fifth column represents, minimum 1% and

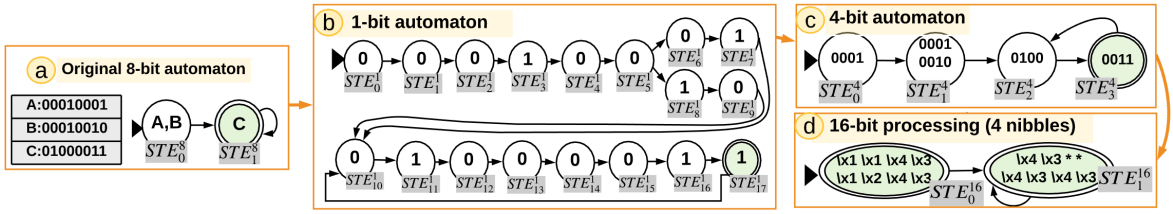

Figure 3: An 8-bit automaton (a) is converted to the minimized *1-bit* automaton (b). The *4-bit* automaton (c) is generated from the *1-bit* automaton. Finally, the 4-bit automaton (C) is strided to a 16-bit processing (d) using nibble units.

maximum 8.5% of the states in the applications are reporting state. We use this observation to optimize the resources for our reporting architecture. *#Reports* shows the total number of generated reports across the entire execution of the application. *#Report Cycle* shows the number of cycles in which at least one report is generated. For example, ExactMatch generates exactly one report in 35 cycles. One input symbol is processed per clock cycle; thus, the total number of cycles for the entire application to run is 1,000,000 for 1MB input stream. The last column shows the percentage of the cycles where at least one report is generated.

Reporting behavior: as Table 1 suggests, the reporting behavior varies significantly from application to application. Some applications report very infrequently (i.e., Dotstar03-09, ClamAV, Ranges05, Ranges 1). This is mainly because the automata in these applications are either a set of virus scanning signatures or detecting a bad behavior in a network, and this reporting behavior is expected. Hamming and Levenstein applications are designed for approximate string matching. Their input is generated randomly, and only a few strings within the scoring metrics were identified.

SPM reports nearly every 30 cycles (1,000,000/33,933), and in each reporting cycle, 1394 reports out of 5025 report states are generated on average (i.e., 20% of the reporting states generate a report every 30 cycles). This implies that the reporting architecture should handle the bursty and dense reporting behavior of such applications to avoid significant performance loss.

Snort reports nearly every cycle, and 1.72 reports are generated on average in each reporting cycle. This implies that the reporting architecture needs to handle frequent but sparse reporting behavior efficiently. Other applications, such as Fermi, and RandomForest, report less frequently (e.g., once every 3000 cycles), and generate roughly 7 reports in each reporting cycle. They exhibit infrequent and relatively less sparse reporting behavior. *These all imply that when designing a reporting architecture, hardware and application considerations must have a stall-free, efficient, and general-purpose solution for a variety of behaviors.*

Application-specific report analysis: In addition to understanding the reporting behavior, it is crucial to realize how and when the generated reports for an application will be transferred to the host. For example, in network security applications, the generated reports (which demonstrate a malicious behavior in the network) should be immediately sent back to the host to make a quick decision. Moreover, some applications, such as SPM [58], may need to check if a range of input stream has generated a report or not, and they do not need to know all the reports during the entire execution of an application. Likewise, some applications only need to know if at least one report has happened during a portion or entire execution of the input stream; thus, a summarized reporting would be enough.

A well-designed reporting architecture should transfer the minimal required reporting data (i.e., summarized, a portion, or the entire reports) when the application needs it. To the best of our knowledge, all the existing in-memory automata processing solutions send the entire reporting data when the allocated buffers are filled as buffer flush instruction, and there is no control from the host/application to transfer data selectively. *In Sunder, for the first time, our proposed reporting architecture provides access from the host to request the entire, a portion, or summarized reporting data at any point of time efficiently and cost-effectively* (Section 5.1.2).

Dependency to input stream: the reporting behavior in each application changes with changing the size and characteristics of the input stream (i.e., dynamic behavior), and therefore, the underlying architecture should be robust and still efficient in these cases. The sensitivity analysis of reporting is discussed in Section 7.5.

4 ALGORITHMIC TRANSFORMATION

Sunder leverages the fact that 4-bit automata consume exponentially fewer memory rows for state encoding than 8-bit automata (2^4 vs. 2^8). The unused memory rows in a standard memory or cache subarray can be used to store the reporting data at a minimal cost locally. This section explains the algorithmic aspects of transforming an NFA with m-bit symbols (m is usually 8 and 2^8 memory rows are required for one-hot encoding of states) to 4-bit symbol automata (i.e., *nibble processing*). 4-bit symbols only require 2^4 memory rows for one-hot symbol encoding. We then stride the nibbles to achieve our desired processing rate, and configure Sunder's processing rate accordingly.

Transforming to nibble processing: we use FlexAmata [40], which is an automata transformation tool, and transforms an *m-bit* automaton to an equivalent *n-bit* automaton. In our architecture, *m* is an application-dependent (i.e., depends on the number of unique symbols in the application, and this is usually 8 because of the byte-oriented processing nature of the problems), and *n* is 4. The main reason to transform to 4-bit automata (instead of 2-bit, 5-bit, etc.) is that 4-bit processing has the lowest transformation overhead.

Figure 3 explains how an 8-bit NFA is transformed into a 4-bit NFA. In the notation STE_x^y, x is state index and y is the symbol size. The original homogeneous NFA (a) has two states and accepts language $(A|B)C^+$. FlexAmata generates a binary NFA (b) and minimizes the states when possible. For example, the first 6 bits of

symbols A and B can be merged. Then, the 4-bit NFA (c) is generated from the bit-automaton. In the 4-bit NFA, STE_0^4 is a start state and STE_3^4 is the final state. Each state processes one or more 4-bit symbols. STE_{13}^1 in 1-bit-automata is equivalent to reaching the state STE_2^4 in the 4-bit automaton. Although this transformation seems intuitive in this simple example, the real-word automata are very complex with loops and different rule properties, making the conversion non-intuitive.

Temporal striding: as expected, the nibble processing scheme halves the processing rate compared to the 8-bit automata. To increase the throughput (equals or more than 8-bit processing), we utilize the *Vectorized Temporal Striding* technique introduced in Impala [41] to reshape the 4-bit automaton and find its equivalent automaton that processes multiple nibbles per cycle. Temporal Striding [4, 12] and its vectorized version are transformations that repeatedly square the input symbol of an automaton and adapt its matching symbols and transition graph accordingly. Figure 3 (d) shows how a 4-bit automaton is temporally strided to a 16-bit automaton with nibble units (i.e., a vector of four 4-bit symbols) - see Impala [41] for details. The State and transition overhead of nibble processing transformation is discussed in Section 7.2.

5 SUNDER ARCHITECTURE

In this section, we explain how Sunder implements *reconfigurable nibble processing* and reporting architecture together in the state matching subarrays. Moreover, we explain how state transition (interconnect) is implemented in Sunder. Figure 4 shows the Sunder architecture for one processing unit (PU), which can accommodate an NFA with up to 256 states. Each PU includes the state matching and reporting array, state transition (interconnect) unit, and the global memory-mapped switches to provide inter PU connecting when processing larger automaton (with up to 1024 states).

5.1 State Matching & Reporting Subarray

The green region in Figure 4 depicts one memory subarray of size 256×256, where matching symbols are encoded in the yellow region (upper rows of the subarray), and the reporting data is stored in the gray region (lower rows of the subarray) or partially in the yellow region (depends on the configured processing rate). In prior solutions [17, 43, 48], one memory subarray of size 256×256 is used to only encode 8-bit symbols for state matching stage. However, Sunder leverages nibble processing from prior work [41], and proposes to utilize one memory subarray of size 256×256 to encode up to 16-bit symbols (four 4-bit symbols in the first 64 rows of the subarray in Figure 4) and to store up to 60Kb reporting data! *This is achieved only at the expense of 2% hardware overhead (the blue regions in Figure 4) compared to prior solutions.*

To perform state matching and to store reporting data in the same subarrays in one cycle, we utilize the dual-port functionality of 8T SRAM memeory cells [25], with two sets of sense amplifiers (SA) and two sets of decoders. In the *state matching/reporting subarray* of Figure 4 (green region), up to four 4-bit symbols are decoded using the four 4:16 decoders on the right side of the subarray. The bitwise NOR of multiple activated rows (up to four rows) are sensed by the row buffer on the bottom of the subarray (i.e., *Row-buffer B*) - see Section 5.1.3.

The reporting data is stored in and read from the same subarray (i.e., the lower part of the *state matching/reporting subarrays* in Figure 4) using the left-side 8:256 decoder and the row buffer on the top of the subarrays (i.e., *Row-buffer A*). Moreover, the left-side 8:256 decoder is also used to write the state-matching data in the *Automata Mode (AM)*, and also read/write normal cache data in the *Normal Mode (NM)*.

5.1.1 Reconfigurable Nibble Processing. Different from all prior work, Sunder supports a reconfigurable symbol processing rate (i.e., 4-bit, 8-bit, and 16-bit symbols per cycle). This is unlike Impala [41], where the processing rate is fixed in hardware (i.e., if the hardware is designed for 16-bit processing, the 8-bit processing is not able to utilize half of the subarrays). Each state is encoded in one memory column by embedding multiple 4-bit symbols. The processing rate can be determined by the user based on the application size and requested throughput. If the application is small, automata can be transformed to process more nibbles in one cycle, which results in higher throughput at the expense of utilizing unused hardware resources. On the other hand, if the application is large and several rounds of reconfiguration are needed to process the entire set of rules/automata, a smaller processing rate that avoids overhead from extra states can be selected to optimize for space.

In the state matching/reporting subarray, Row[0:15] encodes the first nibble, Row[16:31] encodes the second nibble, Row[32:47] encodes the third nibble, and Row[48:63] encodes the fourth nibble of the symbol. When the processing rate is 4 bits per cycle, only the first 16 rows are used to encode the 4-bit symbols using a one-hot encoding scheme; thus, only the associated decoder to the first 16 rows will be enabled. This means the remaining rows (i.e., Row[16:255]) are used for storing the reporting data. Likewise, in 8-bit processing, the first 32 rows are used to encode two nibbles; thus, the first two decoders are enabled, and the remaining rows can be used for reporting data. Finally, when the processing rate is 16 bits per cycle, Row[0:63] will be used for state encoding (for four nibbles), and their associated decoders will be enabled accordingly. In this scenario, Row[64:255] can be used to store the reporting data.

The partial state matching results from nibbles are combined using bitwise operations with multi-row activation of SRAM arrays. For example, for the 16-bit processing, four memory rows are activated (with the four 4:16 decoders), then their matching results are bitwise ANDed to generate the final matching results. Jeloka et al. [26] have shown the stability of simultaneously activating 64 wordlines on SRAM subarrays by lowering the wordline voltage and verified this across 20 fabricated chips.

5.1.2 Reporting Architecture. Sunder proposes to localize the reporting data within the same memory subarrays performing the state matching, with minimal hardware overhead. This helps to avoid long wires from report states to buffers and their likely latency and routing congestion. It also helps to share many of the report buffer peripherals with the existing state-matching logic. Thanks to the nibble processing technique, which exponentially saves the memory footprint in the state matching subarrays, and the choice of dual-port 8T cells to isolate read port from write port (Section 5.1.3), Sunder can store the reporting data in each cycle at the bottom rows of the state-matching subarrays. Sunder introduces

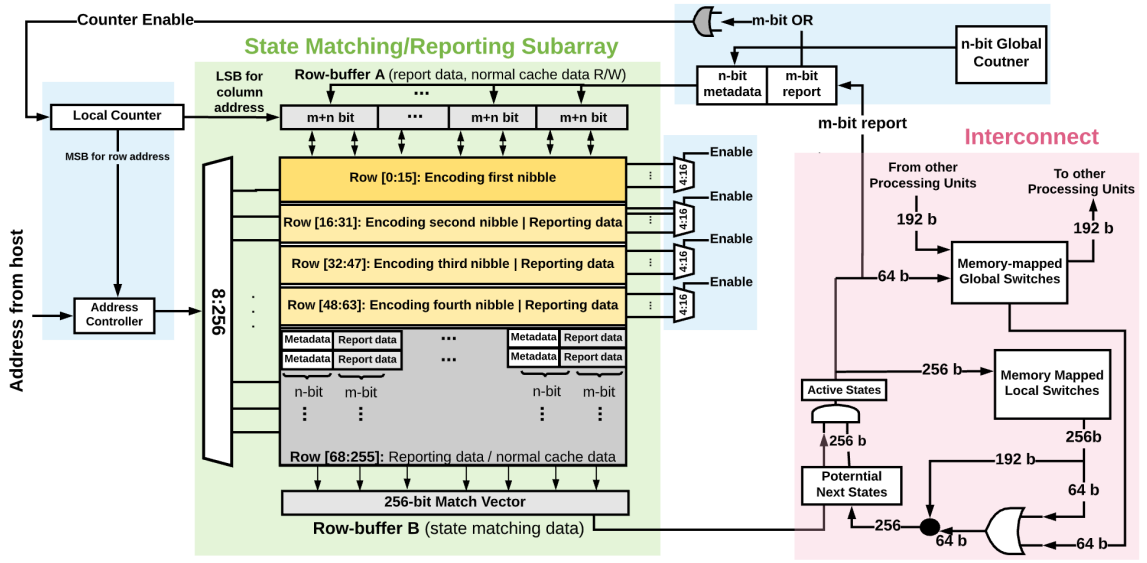

Figure 4: Sunder architecture for state matching/reporting (green), state transition or interconnect (pink), and additional modules to enable reconfigurable nibble processing and reporting architecture (blue).

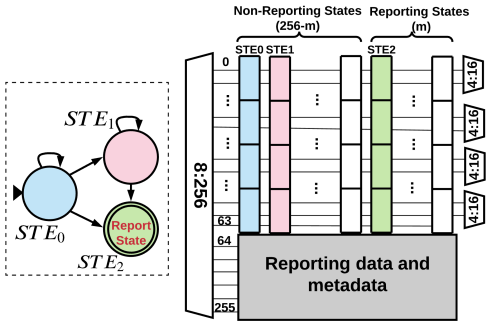

Figure 5: Mapping STEs to reporting/non-reporting regions

several unique features, which can greatly reduce the overhead of reporting.

Storing reporting data in to the *state matching/reporting subarray*: assume the processing rate is 16-bit (i.e., four 4-bit nibbles); therefore, the first 64 rows in the *state matching/reporting subarray* in Figure 4 are used for encoding the states. We assume m reporting states in each memory subarrays, and we map the reporting states in an automaton to the m-reporting-enabled states in the memory subarrays, which are the last m memory columns as shown in Figure 5 (STE_2 is a reporting state and is encoded in the reporting columns). Having a pre-defined reporting region is necessary to efficiently detect if there is at least one generated report.

At run-time, in the automata-mode, after the current active states have been calculated, we check if there is at least reporting data is generated. This is done by ORing the m-bit reporting states (Figure 4 - blue region - upper right side) driven from the *active state* vector (pink region). If at least one report is generated, we need to write the m-bit report data (out of 256 bit) and the cycle in which the report has occurred (n-bit metadata) into the reporting rows. The cycle count (i.e., n-bit metadata) is generated from a global counter in the hardware.

Reporting data and metadata are written into the reporting region row-wise, starting from row 64 in 16-bit processing (or starting from row 32 in 8-bit processing). To track the currently available location in the reporting region (i.e., the currently generated m-bit report is written into which row and which columns), a local counter is used. The counter size is calculated as:

$$LocalCountersize = \lceil \log(\#ReportRows) \rceil + \left\lceil \log(\frac{256}{m+n}) \right\rceil \quad (1)$$

#ReportRows is the number of rows configured for storing reporting data (i.e., in the 16-bit processing, 192 rows can be preserved for reporting data). m is the number of states in a subarray that can be a reporting state (Figure 5), and n is the global counter size. For example, in 16-bit processing, *#ReportRows* is 192, therefore, $\lceil \log(\#ReportRows) \rceil$ is 8. Assuming 8 states out 256 states in a subarray can be reporting states (i.e., m=8) and 10MB input size (i.e., n=24), the local counter is 16-bit. The 8-bit MSB (i.e., $\lceil \log(\#ReportRows) \rceil$) is used for the address decoder to activate a row, and the 8-bit LSB (i.e., $\lceil \log(\frac{256}{m+n}) \rceil$) selects the bitlines for the next available location in a row. Therefore, the corresponding bitlines are pre-charged to write $m+n$-bit reporting data and metadata into the selected columns. The address controller simply selects the address from the host when writing the state matching data at the configuration time, or from the local counter in the Automata Mode. It also masks the row address depends on the number of reporting rows at the configuration time.

As 8T cells have different ports for read and write, the state matching phase and reporting phase (from the previous cycle) can be pipelined. This approach does not need any additional hardware resources such as an arbiter or global buffer, as report information is locally stored in the same memory array as matching data has

been stored. This way, accessing the report information is much easier, as it translates to simply reading data from memory.

Reporting architecture highlights: Sunder introduces unique features that have never been explored before, and can greatly reduce the overhead of reporting in automata processing applications.

Report summarization: an important concern in the reporting architecture is the I/O cost. We observed that not all the applications required cycle-accurate report information (such as SPM [58]). All the previous accelerators are designed to read bulky cycle-accurate report information and post-process them on the host. In Sunder, report summarization is achieved by performing the column-wise NOR operation among report rows using *Port 2* in Figure 6, thanks to wired-NOR functionality of the 8T SRAM subarrays (please note that when report summarization is requested by the host, the state matching is stalled for 1-2 cycles as *Port 2* is used for the multi-row activation required by summarization). This feature is beneficial for applications that have a very frequent reporting behavior, where the existence of a report in a specific duration matters. In other words, the user does not care about the specific cycle that the report has happened (Evaluation in Section 7.5).

Selective reporting: Sunder provides great freedom to the host to read the report status of every state at any cycle with a constant time while the conventional approaches fill the report buffers with report data that might not be interesting at that particular time, and this introduces more stalls to transfer reporting data.

Optimized for different reporting behaviors: when the application has a dense but infrequent reporting behavior, the reporting region has minimal usage. On the other hand, when the application has a sparse but frequent reporting behavior, the reports are compacted in the report-storing subarrays; thus reducing the number of stalls.

FIFO strategy for the reporting buffers: our study on real-world applications reveals that they only generate at least one report in less than 12% of total execution cycles (see Table 1). This implies that in more than 88% of the cycles, no report is generated, and nothing will be written in the subarray. We take advantage of this observation and start reading the reporting data from the beginning of the reporting region. This is enabled by using *Port 1* (in Figure 6) for reading reporting data and using *Port 2* for performing the state matching, both at the same time. When the report buffer is full, the reports will be written to the buffer starting from the head. If the report generation rate is higher than consumption and the report buffer is full, the execution is stalled.

5.1.3 Enabling In-Situ Reporting using Dual-Port Memory Cell Structure. To enable state matching and reading or writing reporting data at the same time in one memory subarray, we utilize dual-port functionality of 8T SRAM cells, presented in Figure 6. The cell supports read/write operation through *Port 1* and read-only operation through *Port 2*. *Port 1* is used for (1) writing initial automata configuration (i.e., encoded symbol-set) to the state-matching subarray (i.e., the upper yellow rows in Figure 4), and (2) reading/writing report data from/to the report region (i.e., the lower gray rows in Figure 4). *Port 2* is used for performing state matching operation by reading the matching data from the memory rows. To perform state matching on multiple nibbles (up to four), the four 4:16 decoder in Figure 4 are used, each activate one row in the subarray, and

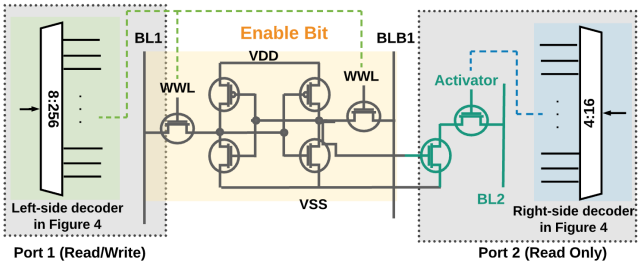

Figure 6: Dual port 8T SRAM cell

BL2 calculates the wired-NOR functionality of the activated rows (explained bellow).

An 8T SRAM cell consists of a classical 6T SRAM cell and two additional transistors, which connect the memory cell to *BL2*. An 8T SRAM cell read operation from *Port 2* starts by precharging the bitline (BL2); then evaluation is done by two serial transistors, one derived by *enable bit* value and the other (i.e., activator) by the wordlines of the 4:16 decoders (Figure 4, right-side decoders). This means the 6T cell drive the *Port 2* bitline (BL2) only when the cell holds '1' and the right row decoder has activated the target cell row. Otherwise, it does affect the bitline (BL2) value. This implies the BL2 implements the wired-NOR functionality of the activated rows. The WWL wordlines are derived by the left-side decoder (8:256) in Figure 4 for read/write operations (same functionality as classical 6T SRAM cell). Sunder also benefits from the wired-NOR functionality enabled by 8T cells for the multi-row activation in the full-crossbar interconnect subarrays.

5.2 Interconnect

The interconnect allows active states to move forward in time to the next states. If (I) the current input symbol matches the state *S* and (II) any of parents of state *S* were activated in the previous cycle, then the state *S* will be activated. The second condition implies that the interconnect should provide the OR-functionality, which is feasible with 8T SRAM switch cells. Similar to prior work [43, 48], we use a memory-mapped full-crossbar interconnect based on 8T SRAM memory cells, to provide wired-NOR functionally on bitlines. The left-side blue wordlines are driven by the left-side decoder (and are connected to WWL in 8T cells) for writing the connectivity data into the enable bits at the configuration time (Figure 6). The right-side purple wordlines are driven *active state vector* (see Figure 4), which determines the currently active states, and are connected to the *activators* in 8T cells. The bitlines (or columns) drive the same set of states (i.e., one column per state). Since every column intersects with every row, the interconnect provides connections between every pair of 256 states, thus, avoiding interconnect congestion even for highly connected NFA.

6 SYSTEM INTEGRATION

Sunder can be realized by repurposing the last level cache (LLC) of recent processors, such as Intel Xeon with large L3 cache capacity (as Sunder's subarrays have the same size as a conventional L3 cache [14]). In the Sandy Bridge microarchitecture, the LLC is split into independent slices (usually equal to the number of cores), connected via a ring topology. To access the target slice, the physical address should go through a hashing block. This hash block distributes addresses uniformly across the slices with the granularity of cache

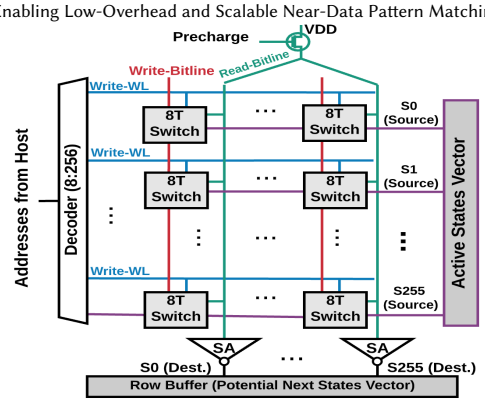

Figure 7: Memory-mapped local & global interconnect

lines, so it is possible to have two consecutive memory addresses mapped to two different memory arrays in different slices. However, to configure Sunder, we need flat accesses to certain arrays. Intel has not published the details of the hashing function. To find physical addresses for the slices repurposed for Sunder, we can use the existing efforts to reverse engineer the hash function [35]. Inside a slice, to access the correct memory array in a certain cache way, we use Cache Allocation Technology (CAT) [1] to restrict the ways accessed by the program.

To cover all the addresses in a certain slice, we can set the page size to 1GB using *mmap* at configuration time. To translate the virtual addresses of the arrays to the physical addresses, page table information in */proc/self/pagemap* can be used. Automata are configured in the cache by writing configuration values at these addresses. At runtime, to collect the report information, the host application issues load instruction at the regions assigned to the report arrays for immediate processing or issues *clflush* to store the report data into the DRAM for post-processing.

7 PERFORMANCE EVALUATION

7.1 Evaluation Methodology

NFA workloads: we use ANMLZoo [56] and Regex [6] benchmark suites to evaluate Sunder. They present a range of applications, such as network intrusion detection, natural language processing, and data mining. A summary of the applications is represented in Table 1.

Experimental setup: we use our open-source in-house Automata compiler and simulator[1] to perform the preprocessing steps, and simulate Sunder, Cache Automaton (CA) [48], Impala [41], and the Automata Processor (AP) [17], and also to perform the automata transformation for nibble processing. The simulator takes NFA in ANML format and processes the input cycle-by-cycle. Per-cycle statistics are used to calculate the number of active states, the number of reports, and communication overhead. To estimate area, delay, and power of the memory subarray in Sunder, Cache Automaton, and Impala model, we use a standard memory compiler (under NDA) for 14nm technology and nominal voltage 0.8V (details in Table 2). For example, Impala uses SRAM subarrays of size 16×16 with 6T cells for state matching, or Sunder uses SRAM Subarrays of size 256×256 with 8T cells for both state matching and the interconnect. The global wire-delays are calculated using SPICE modeling in CA. Because the 8T-cell design has wider transistors than the 6T-cell design in the memory compiler, 8T subarrays are faster and have a higher area overhead than 6T subarrays.

Reporting architecture: Impala and CA overlook the real cost of reporting, and they mainly evaluate the matching kernel. To provide a fair and thorough comparison across different solutions, we assume the AP-style reporting architecture for the CA and Impala.

Parameter selection: on average, 3.9% of the states are the reporting states (Table 1, fifth column). Therefore, on average, 10 out of 256 states (3.9%×256) are reporting states. Based on this observation, we allocate 12 bits for the reporting data and 20 bit for metadata (i.e., the global counter to count for 1Mb of input data), depicted in Figure 4. To allow capturing the reporting information for larger input, the stride value is concatenated with all zeros in the reporting data and is written in the *metadata + report data* region.

AP-style reporting parameters: following [55], each L1 report buffer size is 481Kb and each L2 report buffer size is 64KB (in total, 11.3MB L1 and 4MB L2).

Table 2: Subarray parameters for state-matching and interconnect (including peripheral overhead) in 14nm technology.

Usage	Cell Type	Size	Delay (ps)	Read Power (mW)	Area (μm^2)
State-matching (Impala)	6T	16×16	180	0.58	453
State-matching (CA)	6T	256×256	220	5.52	9394
Interconnect (CA, Impala, Sunder) State-matching (Sunder)	8T	256×256	150	6.07	20102

7.2 State and Transition Overhead

This section discusses the state and transition overhead for different processing rates (i.e., 1, 2, and 4-nibble processing). Table 3 shows the number of states and transitions in each bitwidth, normalized to the number of states and transition in the original 8-bit design. We observed that benchmarks with higher symbol density (i.e., states that accept larger alphabet), such as Brill, EntityResolution, Hamming, Protomata, and RandomForest, have higher state and transition overhead in different bitwidths.

On average, 1, 2, 4-nibble designs have 3.1×, 1.0×, and 1.2× more states and 4.5×, 1.0×, and 1.8× more transitions over the original 8-bit designs. The increase in the number of states translates to utilizing more memory-column resources in in-memory designs. The increase in the number of transitions translates to utilizing more switches in our memory-mapped full crossbar interconnect (Figure 7) and does not incur extra resource overhead.

This means that compared to the original 8-bit processing, 4-nibble processing (or 16-bit processing) provides 2× throughput benefit only at the expense of 1.2× more memory columns. Moreover, 4-nibble processing requires 64 (4×2^4) memory rows to encode four 4-bit symbols, whereas the original 8-bit processing requires 256 (2^8) memory rows to encode one 8-bit symbol. Sunder opportunistically utilized the 192 (256-64) unused memory rows to store the reporting data with a simple and compact solution. This confirms that our algorithm/architecture methodology provides throughput and area benefits compared to prior works.

[1] https://github.com/gr-rahimi/APSim

Table 3: Number of state and transitions in Sunder normalized to the original 8-bit automata.

Benchmark	Sunder State			Sunder Transition		
	1-nibble (4-bit)	2-nibble (8-bit)	4-nibble (16-bit)	1-nibble (4-bit)	2-nibble (8-bit)	4-nibble (16-bit)
Brill	5.3×	1.0×	1.9×	11.9 ×	1.0×	1.8×
Bro217	2.0×	1.0×	1.0×	2.1 ×	1.0×	7.4×
Dotstar03	2.2×	1.0×	1.0×	2.6 ×	1.0×	1.1×
Dotstar06	2.3×	1.0×	1.0×	3.0 ×	1.0×	1.1×
Dotstar09	2.4×	1.0×	1.0×	3.5 ×	1.0×	1.2×
ExactMatch	2.0×	1.0×	1.0×	2.0 ×	1.0×	1.0×
PowerEN	2.3×	1.0×	1.1×	3.1 ×	1.0×	1.0×
Protomata	6.0×	1.0×	1.2×	12.5 ×	1.0×	1.1×
Ranges05	2.0×	1.0×	1.0×	2.1 ×	1.0×	1.0×
Ranges1	2.1×	1.0×	1.0×	2.2 ×	1.0×	1.0×
Snort	2.5×	1.0×	1.1×	3.8 ×	1.0×	1.4×
TCP	2.5×	1.0×	1.1×	3.9 ×	1.0×	1.3×
Hamming	6.5×	1.1×	1.3×	9.7 ×	1.1×	1.4×
Levenshtein	2.8×	1.1×	2.2×	1.9 ×	1.1×	3.5×
Fermi	2.2×	1.0×	1.0×	2.1 ×	1.0×	1.3×
RandomForest	5.3×	1.0×	1.0×	9.4 ×	1.0×	1.0×
SPM	2.7×	1.1×	2.3×	2.7 ×	1.1×	4.6×
EntityResolution	3.2×	0.7×	0.9×	2.8 ×	0.7×	1.6×
Average	3.1×	1.0×	1.2×	4.5 ×	1.0×	1.8×

Table 4: Reporting overhead for four nibble processing.

Benchmark	Sunder w/o FIFO		Sunder w/ FIFO		AP (8-bit)	AP+RAD (8-bit)
	#Flushes	Reporting Overhead	#Flushes	Reporting Overhead	Reporting Overhead	Reporting Overhead
Brill	666	1.04×	0	1×	7.07×	2.95×
Bro217	0	1×	0	1×	1.6×	1.3×
Dotstar03	0	1×	0	1×	1×	1×
Dotstar06	0	1×	0	1×	1×	1×
Dotstar09	0	1×	0	1×	1×	1×
ExactMatch	0	1×	0	1×	1×	1×
PowerEN	0	1×	0	1×	1.1×	1.05×
Protomata	0	1×	0	1×	5.8×	2.32×
Ranges05	0	1×	0	1×	1×	1×
Ranges1	0	1×	0	1×	1×	1×
Snort	1	1.01×	0	1×	46×	9×
TCP	0	1×	0	1×	3.8×	2.5×
ClamAV	0	1×	0	1×	1×	1×
Hamming	0	1×	0	1×	1×	1×
Levenshtein	0	1×	0	1×	1×	1×
Fermi	0	1×	0	1×	2.3×	1.5×
RandomForest	0	1×	0	1×	1.6×	1.3×
SPM	9212	1.06×	3870	1.03×	9.7×	9.7×
EntityResolution	0	1×	0	1×	2.25×	1.8×
Avg. Overhead	NA	1×	NA	1×	4.69×	2.23×

7.3 Performance Overhead Analysis

Table 4 summarize the reporting overhead for Sunder (with and without FIFO strategy), the AP, and the AP augmented with the Report Aggregator Division (RAD) proposed by Wadden et al. [55] (AP+RAD). For all these architectures, automata matching computations and communication happens within a single cycle. Therefore, the "nominal" time it takes for the matching kernel (i.e., only state matching and transition) to run automata on the input symbol stream is equal to the symbol cycle time of the device multiplied by the number of symbols in the input symbol stream. To have apples-to-apples comparison across different solutions, we assume the AP-style reporting architecture (Figure 2) for both Impala and CA (as they overlook the real-cost of reporting overhead), and add the reporting overhead to the "nominal" kernel execution cycles in CA and Impala.

Performance overhead: the number of flushes is the total number of times an application needs to flush the whole reporting region due to overflow. While a subarrray is flushing out the reporting data, the symbol processing for the whole application is stalled. The reporting overhead (the stalls due to gathering and sending the reporting data to the host) represents the slowdown over the nominal execution cycles. Some benchmarks have little or no reporting overheads, even on the AP reporting architecture (e.g., Dotstar, Ranges, ClamAV). This is simply because these benchmarks report infrequently or not at all (Table 1). Some benchmarks incur extremely large reporting overheads on the AP-style reporting. For example, Snort incurs a 46× slowdown over ideal performance, and 7 out of 19 benchmarks spend more time processing reporting overheads than processing automata transitions. AP+RAD reduces the reporting overhead of the applications with sparse reporting behavior, such as Snort and Protomata. However, it does not improve the overhead of dense reporting, such as SPM, mainly because RAD technique has finer-grain reporting granularity that reduces the sparsity of sparse reporting behavior and has almost no impact on dense reporting.

As expected, Sunder reporting architecture incurs negligible overhead, which is less than 1.06× slowdown with no FIFO design, and this can even further decrease to less than 1.03× when applying the FIFO strategy (which reads from the beginning of the report array during the application execution). This simply means that Sunder's end-to-end performance is almost equal to the kernel performance, and minimizing the data movement overhead between CPU and memory is an ultimate mission in processing-in-memory architectures! Technically, Sunder does not incur stalls during the execution of an application due to reporting, can be used for real-time processing with a reliable and predictable performance.

SPM has extremely high-frequency reporting behavior and is the only application that has reporting overhead in Sunder architecture (3% reporting overhead - 6th column in Figure 4). Interestingly, the SPM application mostly requires to know if a single report has happened for specific input intervals with no interest in knowing the exact cycles that report events have occurred. This means that our report summarizing technique can further reduce the reporting overhead for the applications with extremely high reporting behavior.

7.4 Comparison with Prior Work

Overall performance: The delays and frequencies of different pipeline stages for Sunder, Impala, CA, and the AP are shown in Table 5 (derived from Table 2). Sunder uses memory subarrays with 8T cells for both state matching and interconnect. Sunder, CA, and Impala have similar hierarchical interconnect designs, and both local and global switches are evaluated in parallel (Figure 4). We assume an SRAM slice of $3.19mm \times 3mm$ based on CA. As a result, the distance between SRAM arrays and global switch is assumed to be $1.5mm$. The wire delay was found to be $66ps/mm$ from SPICE modeling; therefore, the wire delay for global switches is $99ps$. Global switch delay for CA and Sunder is 249 ps, composed of read-access latency (150ps) and wire delay latency (99ps). Impala state-matching subarray is ~5X smaller; therefore, we assume $20ps$ wire-delay for Impala. Therefore, the global switch delay for Impala is 170 ps.

The frequency is determined based on the slowest pipeline stage. To consider potential estimation errors, we assume the operating frequency to be 10% less than what we have calculated. The AP is

Table 5: Delays and operating frequency in pipleline stages. The AP's detail implementation is not publicly available.

Architecture	State Matching	Local Switch	Global Switch	Max Freq. (GHz)	Operating Freq. (GHz)
Sunder (14nm)	150 ps	150 ps	249 ps	4.01	3.6
Impala (14nm)	180 ps	150 ps	170 ps	5.55	5
CA (14nm)	220 ps	150 ps	249 ps	4.01	3.6
AP (50nm)	-	-	-	0.133	0.133
AP (14nm)*	-	-	-	1.69	1.69

* Projected to 14nm

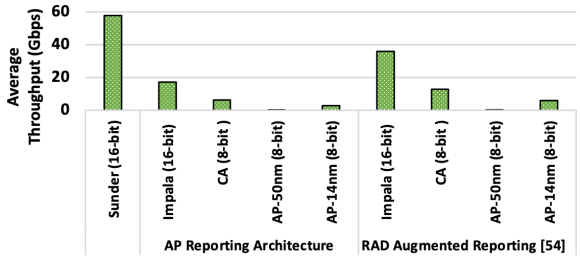

Figure 8: Throughput of different automata accelerators.

designed in 50nm DRAM technology. To have a fair comparison, we project the frequency to 14nm, which is an ideal assumption.

The overall throughput of in-memory automata processing architectures is determined by $\frac{frequency \times (\#bits/cycle)}{reporting-overhead}$. This is unlike prior work that calculates $frequency \times (\#bits/cycle)$ as the overall performance and overlooks the reporting overhead. The $reporting - overhead$ is the average reporting overhead in Table 4, and is equal for CA, Impala, and AP. Based on Table 4, Sunder's reporting architecture has almost no performance overhead across all the benchmarks, thus, throughput for all the applications is fixed and calculated by multiplying frequency (3.6 GHz) by the number bits processed per cycle (i.e., 16 bits/cycle). Impala has a fixed 16-bit per cycle processing rate, whereas Sunder has a reconfigurable 16-bit per cycle processing rate. CA and AP design only work with 8-bit per cycle rate. Figure 8 compares the average throughput across the 19 benchmarks for Sunder against Impala, CA, and the AP for both AP-style reporting architecture and RAD reporting proposed in [55]. Sunder achieves 280× (133×), 22× (10.4×), 10× (4.8×), and 4× (1.9×) higher throughput compared to the AP (50nm), AP (14nm), CA, and Impala, respectively, with considering AP reporting architecture (with considering AP+RAD reporting architecture). This benefit comes from the fact that Sunder has almost no reporting overhead (i.e., reporting does not cause a slowdown in performance), which can provide a deterministic throughput of one input symbol per cycle!

Area Overhead: Figure 9 compares the area overhead of state-matching, interconnect, and reporting of Sunder with Impala, CA and, and the AP (all in 14nm) for 32K STEs. Impala uses four 16×16 SRAM subarrays (6T cells) for the state matching, thus, has the minimum area overhead for this stage. Impala and CA reporting overhead are modeled after the AP reporting architecture. In Sunder, the reporting architecture is infused in the state matching subarray, and there is only an additional 2% overhead for the addition circuitry (i.e., blue area in Figure 4). Both state matching and interconnect switches in Sunder are designed with 8T SRAM subarrays, which

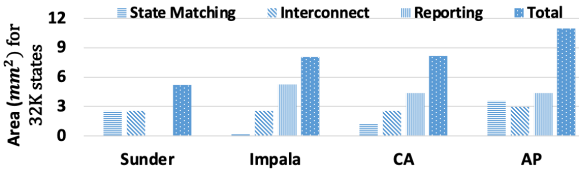

Figure 9: Comparing area overhead for 32K STEs.

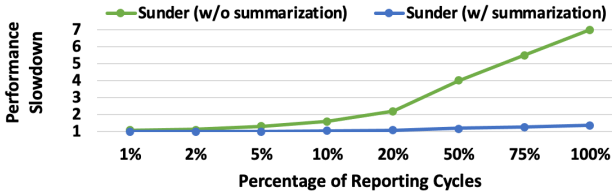

Figure 10: Performance slowdown for various reporting rates.

is 2.1× larger than the 6T subarrays (Table 2). Overall, Sunder has 2.1×, 1.6×, and 1.5× lower area overhead than AP, Impala, and CA, respectively. This benefit comes from the compact and in-place reporting enabled by algorithmic transformation. Moreover, Sunder incurs almost no performance penalty for the reporting, while the other solutions cause up to 46× slowdown due to stalls for reporting.

7.5 Input Stream Sensitivity Analysis

Variations on the input steam change the reporting behavior. To evaluate this, we perform a sensitivity analysis on the percentage of reporting cycles, sweeping from 1% to 100%. We assume 12 reporting states in each subarray (based on the analysis in Table 1). Figure 10 represents the performance slowdown with and without summarization technique (Section 5.1.2). As expected, Sunder reporting architecture incurs negligible performance overhead when the reporting cycles are less than 5%. In the absolute worst-case scenario, which is reporting 100% of the times, Sunder with no summarizing incurs only 7× performance overhead. However, if the application only needs to know the report occurrence, then Sunder can summarize the reporting region in 16-row batches, which improve the performance overhead to only 1.4×. However, the AP-style reporting incurs up to 46× slowdown with only 3.24% of report cycles (SPM in Table 1).

8 RELATED WORK

Data movement is highly expensive, much more expensive than the computation [11, 28–30]. Generally, automata processing on von Neumann architectures exhibits highly irregular memory access patterns with poor temporal and spatial locality, which often leads to poor cache and memory behavior [5, 13, 24, 53, 56, 60], and this increases the cost of data movement.

Existing regular expression accelerators [18, 20, 34, 49] and FPGA solutions [27, 53, 63, 64] are mainly targeting network security applications. However, recent effort on in-memory automata processing architectures are optimizing for general-purpose pattern matching, and they have shown their effectiveness on a wide-range of applications [17, 36, 41, 47, 48] They all optimize for NFA processing by

exploiting the inherent bit-level parallelism of memory, and these allow many patterns to be processed at the same time.

The Automata Processor (AP) [17] presented the first in-memory automata processing solution by repurposing DRAM arrays. Wadden et al. [55] introduce a reporting compression scheme (RAD) to reduce the reporting overhead in spatial automata accelerators, such as the AP. Impala [41] and Cache Automaton (CA) [48] repurpose a portion of the LLC (when in automata mode) to lay out the patterns in the cache subarrays, and thus, providing a solution that can be on the same chip as the CPU, as well as higher-frequency operation due to use of SRAM. Impala transforms automata to process four 4-bit symbols in parallel using smaller subarrays. However, Impala's processing rate is not reconfigurable. Moreover, both Impala and CA overlook the real cost of reporting.

Prior work has already shown that the AP performs at least an order of magnitude better than GPUs and multi-core processors [10, 39, 45, 50, 58, 59], and CA performs at least an order of magnitude better than the AP [48]. Liu et al. [33] proposed an optimized GPU solution for NFA processing by identifying the source of data movement and achieved significant speedup over existing GPU solutions, and even outperforming the AP for several applications. On average, Sunder outperforms the AP 280×, and therefore, we do not compare with the GPU solutions.

9 CONCLUSIONS

We introduce Sunder, a fully reconfigurable, efficient, and low overhead in-SRAM pattern processing accelerator. Sunder integrates our analysis of the prior architectures and sources of inefficiencies, and our study of the static structure and dynamic behavior of real-world applications, to implement the next-generation of in-memory automata processing. Transforming an automaton for better hardware utilization exponentially reduces memory usage and increases information density. This frees up space in the memory subarrays and creates an opportunity to store the reporting data locally in each subarray to significantly reduce the host communication and stabilize the processing throughput across the execution of an application. Sunder's reporting architecture incurs less than 2% hardware overhead (as the reporting data are co-located in the state matching subarrays and use shared resources). On average, Sunder has two orders of magnitude higher throughput than Micron's AP and one order to magnitude higher throughput than the state-of-the-art SRAM-based solutions.

Moreover, our software and hardware methodology enables three orders of magnitude higher throughput per unit area compared to the Micron's AP, and this low-cost, high-throughput solution hopefully shows a path toward commercial viability and unlocks the full potential of automata processing by making it accessible to an increasing set of pattern processing applications with real-time requirements.

ACKNOWLEDGMENTS

We thank the anonymous reviewers whose comments helped improve and clarify this manuscript. This work is funded, in part, by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA, the startup funding provided by the University of California, Riverside, SRC Task No. 2988.001, National Science Foundation (NSF) #2127780, and Department of the Navy, Office of Naval Research, grant #N00014-21-1-2225.

REFERENCES

[1] [n. d.]. Intel Cache Allocation Technology. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html. [online; accessed November 23, 2020].
[2] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. 2018. ASPEN: A scalable In-SRAM architecture for pushdown automata. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[3] Zachary K Baker and Viktor K Prasanna. 2004. Time and area efficient pattern matching on FPGAs. In *12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*.
[4] Michela Becchi and Patrick Crowley. 2008. Efficient regular expression evaluation: theory to practice. In *4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM.
[5] Michela Becchi and Patrick Crowley. 2013. A-dfa: A time-and space-efficient dfa compression algorithm for fast regular expression evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)* (2013).
[6] Michela Becchi, Mark Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*.
[7] Michela Becchi, Charlie Wiseman, and Patrick Crowley. 2009. Evaluating regular expression matching engines on network and general purpose processors. In *5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*.
[8] 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 International Symposium on High-Performance Computer Architecture*. IEEE.
[9] Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. 2019. Automata processing in reconfigurable architectures: In-the-cloud deployment, cross-platform evaluation, and fast symbol-only reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2019).
[10] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. 2016. Entity resolution acceleration using the Automata Processor. In *IEEE International Conference on Big Data*.
[11] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. 2016. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters* 16, 1 (2016).
[12] Benjamin C Brodie, David E Taylor, and Ron K Cytron. 2006. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH Computer Architecture News* 34, 2 (2006).
[13] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010).
[14] 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 *2013 Symposium on VLSI Circuits*. IEEE.
[15] Young H Cho and William H Mangione-Smith. 2005. A pattern matching coprocessor for network security. In *42nd Design Automation Conference*. IEEE.
[16] Yoginder S Dandass, Shane C Burgess, Mark Lawrence, and Susan M Bridges. 2008. Accelerating string set matching in FPGA hardware for bioinformatics research. *BMC bioinformatics* 9, 1 (2008), 197.
[17] Paul Dlugosch, Dean 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).
[18] 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), 48th Annual IEEE/ACM International Symposium on*.
[19] Victor Mikhaylovich Glushkov. 1961. The abstract theory of automata. *Russian Mathematical Surveys* (1961).
[20] 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)*.
[21] Linley Gwennap. 2014. New Chip Speeds NFA Processing Using DRAM Architectures. In *In Microprocessor Report*.
[22] John E Hopcroft. 2008. *Introduction to automata theory, languages, and computation*. Pearson Education India.

[23] Nen-Fu Huang, Hsien-Wei Hung, Sheng-Hung Lai, Yen-Ming Chu, and Wen-Yen Tsai. 2008. A GPU-based multiple-pattern matching algorithm for network intrusion detection systems. In *22nd International Conference on Advanced Information Networking and Applications-Workshops (aina workshops)*. IEEE.

[24] Intel. [n. d.]. https://github.com/01org/hyperscan.

[25] Satoshi Ishikura, Marefusa Kurumada, Toshio Terano, Yoshinobu Yamagami, Naoki Kotani, Katsuji Satomi, Koji Nii, Makoto Yabuuchi, Yasumasa Tsukamoto, Shigeki Ohbayashi, et al. 2008. A 45nm 2-port 8T-SRAM using hierarchical replica bitline technique with immunity from simultaneous R/W access issues. *IEEE Journal of Solid-State Circuits* 43, 4 (2008).

[26] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. 2016. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits* 51, 4 (2016).

[27] Vlastimil Košar and Jan Korenek. 2014. Multi-stride nfa-split architecture for regular expression matching using FPGA. In *9th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*.

[28] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. 2020. Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical in-situ Accelerators. *The 26th IEEE International Symposium on High-Performance Computer Architecture* (2020).

[29] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, M Arif Rahman, and Mircea R. Stan. 2019. An Overflow-free Quantized Memory Hierarchy in General-purpose Processors. *IEEE International Symposium on Workload Characterization* (2019).

[30] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *53rd Annual Design Automation Conference*. ACM.

[31] Alan Wee-Chung Liew, Hong Yan, and Mengsu Yang. 2005. Pattern recognition techniques for the emerging field of bioinformatics: A review. *Pattern Recognition* 38, 11 (2005).

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

[33] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why GPUs are slow at executing NFAs and how to make them faster. In *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[34] 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)*.

[35] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse engineering Intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*. Springer.

[36] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. 2020. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs. In *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

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

[38] 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).

[39] 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 *24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[40] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. FlexAmata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[41] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[42] Elaheh Sadredini, Reza Rahimi, and Kevin Skadron. 2020. Enabling In-SRAM Pattern Processing with Low-Overhead Reporting Architecture Accelerators. *IEEE Computer Architecture Letters* (2020).

[43] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A scalable and efficient in-memory accelerator for automata processing. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[44] 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).

[45] 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.

[46] Tian Song, Wei Zhang, Dongsheng Wang, and Yibo Xue. 2008. A memory efficient multiple pattern matching architecture for network security. In *IEEE INFOCOM -The 27th Conference on Computer Communications*.

[47] Arun Subramaniyan and Reetuparna Das. 2017. Parallel automata processor. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*.

[48] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache Automaton. In *50th Annual IEEE/ACM International Symposium on Microarchitecture*.

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

[50] II Tommy Tracy, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabe Robins. 2015. Nondeterministic finite automata in hardware—the case of the Levenshtein automaton. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[51] 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.

[52] Tommy Tracy II, Lucas M Tabajara, Moshe Vardi, Kevin Skadron, et al. 2020. Runtime Verification on FPGAs with LTLf Specifications. In *20th Conference on Formal Methods in Computer-Aided Design –FMCAD*, Vol. 1. TU Wien Academic Press.

[53] Lucas Vespa, Ning Weng, and Ramaswamy Ramaswamy. 2010. MS-DFA: Multiple-stride pattern matching for scalable deep packet inspection. *Comput. J.* 54, 2 (2010).

[54] Jack Wadden. [n. d.]. Virtual Automata Simulator - VASim. https://github.com/jackwadden/vasim.

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

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

[57] Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, and Kevin Skadron. 2018. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*.

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

[59] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2017. Hierarchical Pattern Mining with the Micron Automata Processor. In *International Journal of Parallel Programming (IJPP)*.

[60] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: a fast multi-pattern regex matcher for modern CPUs. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*.

[61] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (1995).

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

[63] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. 2008. High-speed regular expression matching engine using multi-character NFA. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

[64] Yi-Hua Yang and Viktor Prasanna. 2012. High-performance and compact architecture for regular expression matching on FPGA. *IEEE Trans. Comput.* 61, 7 (2012).

[65] 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 *ACM/IEEE symposium on Architecture for Networking and Communications Systems*.

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