Enabling In-SRAM Pattern Processing With Low-Overhead Reporting Architecture

Elaheh Sadredini, Reza Rahimi, and Kevin Skadron

Abstract—The demand for accelerated pattern matching has motivated several recent in-memory accelerator architectures for automata processing, which is an efficient computation model for sophisticated pattern matching. Existing in-memory pattern matching architectures focus on accelerating the pattern matching kernel, but either fail to support a practical reporting solution or overlook the reporting stage. However, gathering and processing the reports can be the main bottleneck, especially for applications with high reporting frequency. Moreover, all the existing in-memory architectures work with a fixed processing rate (mostly 8 bits per 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-award transformations. The key insight of our work is that transforming the commonly-used 8-bit processing to nibble-processing reduces required hardware resources (i.e., number of used memory rows) exponentially and achieves higher information density. This frees up space for storing reporting data in-place, which significantly eliminates host communication and reporting overhead. As a result, Sunder enables a low-overhead, energy-efficient, and high-performance in-memory pattern matching solution. Our results confirm that Sunder reporting architecture has *zero* performance overhead for 95% of the applications and incurs only 2% additional hardware overhead compared to the state-of-the-art solutions with no support for the reporting stage.

Index Terms—In-SRAM processing, near-data processing, pattern matching, automata processing

1 Introduction

The performance gap between processor and memory, also known as the memory wall, has been a primary performance concern for many years. This problem is aggravated in memory-bound applications, such as pattern matching kernels in big-data domains, where millions of patterns should be processed at once, mostly with real-time and high-throughput processing requirements. Pattern matching is involved in many applications such as network security, bioinformatics, data mining, and natural language processing. These patterns are generally massive in number, complex in structure, and dynamic in behavior. These properties, combined with an increase in the volume and velocity of data, make high-throughput pattern matching ever more challenging.

One leading methodology for inexact pattern matching is to use regular expressions or equivalent finite automata to identify these complex patterns. To address the memory-wall challenges, in-memory architectures for automata processing have been introduced to benefit from the massive internal memory bandwidth by performing symbol matching using memory arrays [3], [6], [8]. They all support the execution of Non-deterministic Finite Automata (NFA) in memory arrays by providing a reconfigurable infrastructure to implement finite automata in hardware.

- *Elaheh Sadredini is with the Department of Computer Science, UC Riverside, CA 92507 USA. E-mail: el.sadredini@gmail.com.*
- *Reza Rahimi and Kevin Skadron are with the Department of Computer Science, University of Virginia, Charlottesville, VA 22904 USA. E-mail: {rahimi, ks7h}@virginia.edu.*

Manuscript received 5 Oct. 2020; revised 21 Nov. 2020; accepted 21 Nov. 2020. Date of publication 3 Dec. 2020; date of current version 23 Dec. 2020.
(Corresponding author: Elaheh Sadredini.)
Digital Object Identifier no. 10.1109/LCA.2020.3042194

In-memory automata processing model has three processing stages; state-matching, state-transition, and report-gathering, and can be combined in a pipeline 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 a memory row. In the state-transition stage, successors of active states are determined by propagating signals via an interconnect. In the report-gathering phase, the report data is accumulated and eventually analyzed for the final action or decision.

Prior work, including our own, has mostly neglected the real cost of providing a reporting architecture and assumed that reporting is not a bottleneck [6], [7], [8] (and evaluated only the first two stages). However, reporting incurs a significant cost when it is considered accurately. For example, the reporting architecture in the Automata Processor [3] has 40% area overhead [4] and up to 46× performance overhead due to stalls and host communications [9].

Moreover, existing in-memory automata accelerators have a fixed processing rate set at design time—typically 8 bits. This can limit the capacity and throughput benefits of in-memory solutions for running a diverse set of applications. For example, if the hardware capacity is more than what an application needs, then the automata could be transformed to process multiple symbols per cycle (i.e., higher throughput) at the expense of a higher number of states, which translates to utilizing the unused hardware resources. On the other hand, if the application has a small number of symbols (e.g., genome sequencing with an alphabet of only four unique symbols), then the automata hardware could use smaller structures, or the automata could again be transformed to process four input symbols per 8-bit input. In short, a fixed 8-bit processing architecture wastes memory resources.

To address these issues, we propose Sunder, a highly reconfigurable in-SRAM automata processing design with a flexible, compact, and low-overhead memory-mapped reporting architecture. Our main observation is that transforming 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 multiple 2^4 memory rows) can greatly reduce the required memory elements in memory-based automata processing solutions. We opportunistically utilize the saved memory rows in a subarray to store the reporting data locally near the state-matching data. This greatly eliminates data movement and stalls due to reporting, and provides an easy and flexible mechanism for the host to analyze any portion of reporting data at any time. In addition, the reconfigurable processing rate enables throughput and density benefits for a diverse set of applications.

Our reporting architecture only incurs a negligible hardware overhead (less than 2%), as it repurposes existing components for state matching subarrays. Our experimental results show that more than 95% of the real-world automata benchmarks [1], [10] have almost no performance degradation with the Sunder reporting architecture (thanks to our compact and localized reporting architecture and our optimized 3-stage pipeline) compared to prior solutions that have overlooked the reporting stage. Sunder can be realized by repurposing the available on-chip SRAM memory or custom-designed memory arrays.

To support a larger automaton, Liu *et al*. [5] propose a hybrid solution to split an automaton between the CPU and the AP [3]. This approach generates more intermediate results (or reports), which needs to be transferred to the CPU, and thus, can significantly benefit from our compact and efficient reporting architecture.

In summary, this paper makes the following *contributions*.

- We present Sunder, a highly reconfigurable and flexible in-SRAM pattern processing architecture using a well-designed algorithm/architecture design methodology.

- We thoroughly analyze the behavior of a diverse set of applications to understand the reporting pattern.
- We introduce a compact, simple, and localized memory-mapped reporting architecture to reduce data movement and host communication, which results in significantly higher performance and area efficiency.
- We provide an open-source framework for the algorithm transformations, mapping, and placement.

2 Background and Motivation

All the previous hardware implementations of automata processing [3], [7], [8] suffer from three problems. First, they all have a fixed symbol processing rate decided at design time. Second, they all have failed in realizing an efficient reporting architecture design. Third, their rudimentary reporting architecture does not provide any support to summarize reporting data in hardware.

Fixed Processing Rate. Two critical design parameters for automata engines are the symbol size and the number of symbols being processed per cycle, which determines system throughput. All previous in-memory automata processing architectures have a fixed processing rate (usually 8 bits per cycle) [3], [7], [8]. This limitation causes density and throughput disadvantages for some applications. For example, if the application size is smaller than the hardware capacity, we can increase the processing rate (i.e., throughput) by utilizing the unused hardware resources. On the other hand, if the application has a small number of unique symbols (e.g., only four in genome sequencing), we can reduce the symbol size to improve capacity, and be able to accommodate more patterns in a unit of hardware.

Reporting Architecture Issues. The reporting architecture module is responsible for collecting per-cycle report information and storing them in a buffer temporarily, to be transferred to the host whenever the host program needs to check for matches. Realizing such a hardware module is not straightforward because there are a few concerns that need to be considered. First, report states are generated in different memory arrays and need to be routed toward the global reporting buffers, potentially with high latency. Second, choosing the right buffer bit-width is challenging due to its effect on area cost. A wide buffer solution (e.g., [3]) 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., [9]) works effectively for applications with sparse reporting behavior and can physically be placed near to 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 is not negligible 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 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 currently first read all the reporting data of the buffer associated with that state and calculate the row-wise logical OR of the reporting cycles. But this is much more efficiently done in the report buffer.

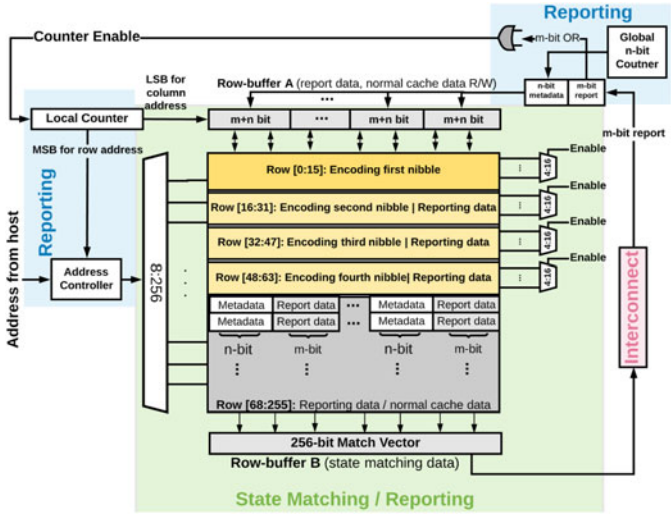

Fig. 1. Sunder architecture for state matching, state transition, and reporting.

3 Sunder Architecture

Sunder leverages the fact that 4-bit automata consume exponentially fewer memory rows for state encoding than 8-bit automata (2^4 versus 2^8). The unused memory rows in a standard memory or cache subarray can be used to locally store the reporting data at a minimal cost. Fig. 1 shows the overall architecture of Sunder for one processing unit (PU). Each PU can process up to 256 automata states, and allows up to 12 reporting states (i.e., final states) and full connectivity among the states using memory-mapped full-crossbar local and global switches.

3.1 State Matching

The state matching stage is realized by the one-hot encoding of the state symbols in columns of a memory subarray. The green box in Fig. 1 depicts one memory subarray of size 256×256 (conventional subarray size in an L3 Cache [2]), and Sunder stores both state matching and reporting data in the same subarray by employing a precise and optimized algorithm/architecture methodology to compress the state matching data and make room for reporting information. This is unlike prior work [3], [8] that uses the whole subarray to store state matching data. To be able to perform state matching and store reporting data in the same subarrays in one cycle, the PU subarrays are dual-ported (i.e., two sets of sense amplifiers (SA) and two sets of decoders).

Algorithmic Transformation. We utilize our prior work, Impala [6], and transform 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, which we call it *nibble processing*. 4-bit symbols only require 2^4 memory rows for one-hot symbol encoding. We then use the temporal striding algorithm presented in Impala [6] to stride the nibbles to our desired processing rate, and configure the Sunder processing rate accordingly. This transformation avoids any false positive reports by splitting the states when needed (details in [6]).

Reconfigurable Nibble Processing. different from all previous work, Sunder allows for a reconfigurable processing rate, i.e., 4-bit, 8-bit, and 16-bit processing. This can be specified at compilation time (usually based on the properties of the target application). In the *State Matching/Reporting* subarray in Fig. 1, 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 (1 nibble), only the first 16 rows are used to encode the 4-bit symbols, and thus, only the associated decoder to the first 16 rows will be enabled. This means the remaining rows (i.e., Row[16:]) can be

TABLE 1
Reporting Statistics for Four Nibble Processing

	Brill	Bro217	Dotstar03	PowerEN	Protomata	Ranges05	Snort	TCP	Hamming	Levenshtein	Fermi	RandomForest	SPM	EntityResolution
#States	42658	2312	12144	40513	42009	12621	66466	19704	11346	2784	40783	33220	100500	95136
#Subarrays	167	16	47	288	197	49	347	77	44	11	200	138	419	372
#Report Cycles	118814	17210	1	4303	105722	38	995011	103198	2	4	13444	3322	33933	28612
#Reports/Report Cycle	9.2	1	1	1	1.2	1	1.7	1	1	1	7.2	6.4	1394	1.3
Report Overhead (%)	0	0	0	0	0	0	0	0	0	0	0	0	2.3	0

used for storing the reporting data or normal cache data. The same analogy applies to 8-bit and 16-bit processing.

The partial state-matching results from nibbles are combined using bitwise operations with multi-row activation of 8T SRAM arrays. For the 16-bit processing, four memory rows are activated (with the four 4:16 decoders), and their matching results are bitwise ANDed to generate the final matching results.

3.2 Reporting Architecture

Many previously proposed architectures for automata processing have neglected the reporting stage in automata processing, and their performance and cost results are only based on state matching and interconnect stages [6], [8]. Sunder improves upon this by localizing the reporting data within the very 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 stage 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, Sunder is able to store the reporting data in each cycle at the bottom rows of the state-matching subarrays (i.e., row 16 onward in Fig. 1).

Report Storing Mechanism. Assume the processing rate is 16-bit; therefore, the first 64 rows in the memory subarrays in Fig. 1 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. At run-time, in the automata-mode, after the current active states have been calculated, we check if there is any reporting data is generated. This is done by O-Ring the m-bit reporting states driven from the *active state vector*. If at least one report is generated, we then need to store this information along with the current cycle in which the report has happened. The cycle count is generated from a global counter in the hardware. Therefore, we concatenate the cycle number as the metadata to the reporting data, and write it in the reporting region of the subarray in a very compact way.

As 8T cells have different ports for read and write, the state matching phase and reporting phase (from the previous cycle) can be performed in pipeline. 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. Sunder introduces several unique features, which can greatly reduce the overhead of reporting.

- *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). 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 OR operation among report rows. This feature is very beneficial for applications that have a very frequent reporting behavior, where the existence of a report in a specific duration matters to the user. In other words, the user does not care about the specific cycle that the report has happened.
- *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 does not usually fill, and thus, there will be no stall during execution to flush the reporting data. On the other hand, when the application has a sparse but frequent reporting behavior, the reports are compacted in the report-storing subarrays, and thus, reducing the need to stall the application.
- *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 number execution cycles. This implies that 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. When the report buffer is full from the end, the reports will be written from the head of the buffer. If the report generation rate is higher than consumption and the report buffer is full, the execution is stalled.

4 Evaluation Methodology

We evaluate our proposed claims and architectures using ANML-Zoo [10] and Regex [1] benchmark suites. They represent a set of diverse applications, including machine learning, data mining, and network security. We use our open-source tool to perform automata simulation, automata transformation, and report analysis.

5 Results

Table 1 shows a summary of the automata reporting statics and behavior for a wide set of benchmarks [1], [10], which significantly varies from application to application. #Subarrays is the number of required memory subarray for each application. *#Report Cycle* shows the number of cycles in which at least one report is generated, and *#Reports/Report Cycle* demonstrates the average number of generated reports in each report cycle.

Reporting Behavior. Some applications report very infrequently (i.e., Dotstar03 and Ranges05). This is mainly because the automata in these applications are either a set of virus scanning signature or detecting a bad behavior in a network, and this reporting behaviour is expected. In SPM, in each report 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 be able to handle the bursty and dense reporting behaviour of such applications to avoid significant performance loss. These analyses have motivated Sunder reporting architecture.

Performance Overhead. We employ the FIFO strategy to read out the reporting data for the report buffer during the execution of an

application. We calculate the total number of cycles for executing each application including the report gathering and transferring to the host. The last row of the Table 1 shows the overhead (number of execution cycles) of Sunder architecture with a complete reporting solution compared to the prior work [6], [8]. Thanks to our pipeline architecture and optimized design, the reporting stage does not incur nearly any performance penalty compared to the prior work, which only considers the execution cycles for the matching kernel. SPM has 2.3% reporting overhead, and this is because of the extremely high-frequency reporting behavior. Interestingly, SPM application mostly requires to know if a single report has happened for specific input intervals with no interest in knowing the exact cycle that report events have occurred. Our report summarizing technique can further reduce the reporting overhead close to zero.

6 Conclusion

We introduce Sunder, a fully reconfigurable, efficient, and low overhead in-SRAM pattern processing accelerator. Our main observation is that prior work sparsely uses memory subarrays to encode the matching data. Moreover, they do not provide efficient and practical reporting architecture. Sunder leverages these observations and presents a concise and efficient algorithm/architecture methodology to compress the matching data, which frees up space and enables a localized, simple, and compact reporting architecture with less than 2% hardware overhead compared to prior work that overlooks the reporting stage. We also present a fully configurable architecture to adjust the processing rate, reporting buffer size, and normal cache data size based on the properties of the applications. Our performance results reveal that Sunder's reporting architecture does not add performance overhead for more than 95% of the applications due to our optimized pipeline design and simple and localized data controlling mechanism.

Acknowledgments

This work was supported, in part by the NSF under Grant CCF-1629450 and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA. Elaheh Sadredini and Reza Rahimi contributed Equally to this work.

References

[1] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, 2008, pp. 79–89.
[2] W. Chen *et al.*, "A 22 nm 2.5 MB slice on-die L3 cache for the next generation Xeon processor," in *Proc. Symp. VLSI Circuits*, 2013, pp. C132–C133.
[3] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3088–3098, dec. 2014.
[4] L. Gwennap, "New chip speeds NFA processing using DRAM architectures," *Microprocessor Rep.*, 2014.
[5] H. Liu, M. Ibrahim, O. Kayiran, S. Pai, and A. Jog, "Architectural support for efficient large-scale automata processing," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 908–920.
[6] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Impala: Algorithm/Architecture co-design for in-memory multi-stride pattern matching," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 86–98.
[7] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "eAP: A scalable and efficient in-memory accelerator for automata processing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 87–99.
[8] A. Subramaniyan *et al.*, "Cache automaton," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 259–272.
[9] J. Wadden, K. Angstadt, and K. Skadron, "Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 749–761.
[10] J. Wadden *et al.*, "ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, 2016, pp. 1–12.