2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)

An Overlay Architecture for Pattern Matching

Rasha Karakchi
University of South Carolina
Email: karakchi@email.sc.edu

Charles A. Daniels
University of South Carolina
Email: cad3@email.sc.edu

Jason D. Bakos
University of South Carolina
Email: jbakos@cse.sc.edu

***Abstract*—Deterministic and Non-deterministic Finite Automata (DFA and NFA) comprise the fundamental unit of work for many emerging big-data applications, motivating recent efforts to develop Domain-Specific architectures (DSAs) to exploit fine-grain parallelism available in automata workloads. In this paper we present NAPOLY (Non-Deterministic Automata Processor OverLaY), an overlay architecture and associated software that attempts to maximally exploit on-chip memory parallelism for NFA evaluation. In order to avoid an upper bound on NFA size that commonly affects prior efforts, NAPOLY is optimized for runtime reconfiguration, allowing for full reconfiguration in 10s of microseconds. NAPOLY is also parameterizable, allowing for offline generation of a repertoire of overlay configurations with various trade-offs between state capacity and transition capacity. In this paper we evaluate NAPOLY using our proposed state mapping heuristic and the ANMLZoo benchmark suite, and we compare NAPOLY's performance against existing CPU and GPU implementations. To the best of the authors' knowledge this is the first example of a runtime-reprogrammable FPGA-based automata processor overlay.**

***Index Terms*—overlay, automata processor, NFA, DFA, processor-in-memory, reconfigurable computing, pattern matching, heterogeneous computing, domain-specific processor, DSL, DSA, big data, data analytics, data processing, text processing, deep-packet inspection, state machine**

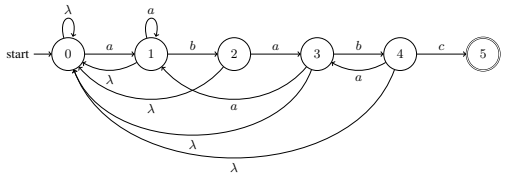
I. Introduction

Datasets comprised of symbolic data, such as genomic sequences, item sets, graph edges, web data, biological data, and network traces are growing rapidly in both size and practical utility. Applications that involve such data include motif discovery [1], approximate string matching [2], signature-based threat detection [3], association rule mining [4], and deep packet inspection [3]. Each of these is algorithmically reducible to the evaluation of a finite automata. On general-purpose architectures, throughput of automata are generally limited by cache performance, especially for datasets having a high rate of complete pattern matches ("reporting rate") or partial matches ("active set").

Evaluating automata on a domain-specific architecture typically comprises an NFA compilation step, reconfiguration step, and pattern matching step. Approaches that synthesize automata directly to an FPGA fabric have extremely long compilation times (hours) and long reconfiguration time (10s of seconds) [5] [6] [7] [8], but achieve very high pattern matching throughput (100s of MB/s or low GB/s).

GPU- and CPU-based approaches have fast compile time and trivial reconfiguration time but low pattern-matching throughput due to limitations in on-chip memory bandwidth and cache performance. These limitations especially affect large automata having a high rate of pattern match activity at runtime. Using specialized memories to avoid these bottlenecks can suffer from long reconfiguration times [9] [10] [11].

Multiple-Instruction Single-Data (MISD) architectures–where the data to be searched is streamed into multiple functional units, where each functional unit tracks partial pattern matches–have a faster reconfiguration time than FPGA-based approaches but lack the ability to make tradeoffs been state density and transition density [12] [8].



state	input	next
0	a	1
0	λ	0
1	b	2
1	a	1
1	λ	0
2	a	3
2	λ	0
3	b	4
3	a	1
3	λ	0
4	c	5
4	a	3
4	λ	0

Fig. 1. DFA for regular expression pattern "ababc".

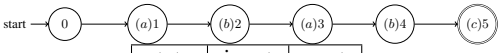
NAPOLY achieves a compromise between purely FPGA- and MISD-based approaches, allowing for rapid runtime reconfiguration while still having architectural customization. NAPOLY exploits as much on-chip memory bandwidth as allowed by the target automata while supporting arbitrarily-large automata workloads.

This paper describes three contributions: (1) an overlay comprised of an array of hardware modules called state elements (SEs), each sensitive to a specific pattern and reconfigured at runtime in 21 to 74 μs depending on the overlay size selected, (2) an allocation heuristic for mapping logical pattern states to SEs, and (3) an analysis of the tradeoffs between state capacity, interconnect density, output buffer size, and reconfiguration time, as well as a performance comparison to multithreaded Intel's CPU-based NFA software (Hyperscan) and a well-known GPU-based implementation (iNFAnt).

II. Background

Deterministic Finite Automata (DFA) are commonly used to implement regular languages and describe sequential logic. If the states are limited to a single true or false output, a DFA becomes a method to search for a set of patterns in a data stream. DFA are designed as a directed graph comprised of a set of states connected by labeled edges. During operation, a DFA can have only one active state and therefore must contain an instanced state for every possible partial match of every pattern, potentially leading to explosive state growth.

2160-052X/19/$31.00 ©2019 IEEE
DOI 10.1109/ASAP.2019.000-7



state	input	next
0	a	1
1	b	2
2	a	3
3	b	4
4	c	5

Fig. 2. Alternative form of NFA with symbols associated with states, for regular expression pattern "ababc"

Fig. 1 shows an example DFA that recognizes a simple regular expression pattern "ababc" along with its corresponding state transition table.

Non-deterministic Finite Automata (NFA) are also described by a set of states connected by labeled edges, but unlike a DFA any subset of states may be active at any time. This way, each state needs track only the progress toward accepting one pattern instead of all possible patterns, reducing the number of states as compared to that of an equivalent DFA.

As shown in Fig. 2, an alternative form of the NFA description is to associate the transition labels with the states instead of the edges, which requires that all incoming transitions of each state have the same symbol set. This offers the advantage of requiring only one symbol set per state, as opposed to one symbol set per transition.

III. NAPOLY State Element and Programmable Interconnect

The NAPOLY design is comprised of an array of state elements (SEs), the design is which is shown in Fig. 3. Each state in the NFA is allocated to one SE. Each potential outgoing transition from each SE is encoded using a single bit in an adjacency vector, such that each slot in the vector corresponds to either a transition or lack of transition to exactly one other SE. Activation passing is handled by AND-ing the active state of the current SE with each "slot" in the interconnect configuration vector to determine if the activation will be passed to the corresponding successor SE.

If the SE is active on a given cycle, it transmits its active state to all its adjacent connected successor SEs, which will become active on the next cycle if their input symbol matches one of the symbols the SE is configured to match, which are stored in a 256 x 1 RAM (implemented as an MLAB). Stated another way, an SE will activate if any of its predecessors pass it an activation (accomplished by OR-ing all incoming activation signals) AND the SE is configured to match the incoming symbol.

Because of this design, each SE can potentially be connected only to a fixed set of potential successor SEs, the number of which determines the size of the activation vector. The maximum number of successors (and predecessors) is limited by the overlay's "hardware fan-out". The hardware fan-out also determines the maximum physical distance between the SEs mapped to a predecessor-successor pair. Thus, each SE sends an output signal to itself and up to $f-1$ of its neighbors, where f = the hardware fan-out. The SEs adopt a one-dimensional addressing scheme, where each SE has an ID number n assigned contiguously across every SE in the overlay and sends output signals to SEs $n - \lfloor \frac{(f-1)}{2} \rfloor$ to $n + \lfloor \frac{f}{2} \rfloor$.

From this description we can derive the amount of on-chip memory bandwidth utilized during operation (as opposed to reconfiguration). Assume $freq$ = the overlay clock frequency, N_{SEs} = the number of SEs, N_{active} = the average number of active SEs, f = the number of possible outgoing connections from each SE (active or not), and $density$ = the average population count of all adjacency vectors as a percentage of f. The on-chip memory bandwidth during operation, as opposed to reconfiguration, can therefore be approximated as Equation 1. Note that this approximation does not include bandwidth used to report accepting state activation, nor does it consider the overlay's "down time" during reconfiguration.

$$bandwidth\left(\frac{\text{bits}}{\text{second}}\right) = freq \times N_{SEs} \times N_{\%active} \times f \times density \quad (1)$$

In short, NAPOLY's interconnect is based on a fixed set of gateable point-to-point connections between neighboring SEs. The interconnect configuration vector of each SE, as well as a "start state" flag and a "reporting" flag, are stored in a set of flip-flops connected in a global shift register. As each potential connection is pre-allocated, no special routing need be performed, only "placement" (mapping nodes and transitions in an automata to SEs and interconnect configuration vectors) which is discussed in detail in section IV.

A. Resource Constraints

SE capacity is limited by RAM capacity, while fan-out is limited by both register capacity and logic needed for OR-gate for each SE's predecessors. Our evaluation FPGA is an Intel Stratix 5 GX A7, which contains RAM in the form of M20K embedded RAMs as well as MLABs (LUT-based memory). Although the S5GXA7 has 7X as much memory available via M20Ks as via MLABs, there are several caveats to using M20K resources to store next-state tables.

The M20K blocks are available in only 20 out of the 209 columns on the FPGA, while the MLAB blocks are more uniformly distributed. Using MLABs avoids congestion around the M20K columns. Further, the tables must have a depth of 256 to accommodate a one-byte symbol alphabet, while the minimum depth required to fully utilize M20K resources is 512, meaning that only 50% of the M20K capacity is available for depth-256 tables. Additionally, the M20Ks have synchronous reads, which if used for the next state table would reduce throughput by 1/2, as each input symbol would require one cycle to access the next state table and another for updating the state flip-flop. Finally, the M20K blocks are needed for other purposes, specifically for the input and output buffers.

The Stratix 5 GX 7A contains 7.16 Mb of MLAB memory, giving a theoretical upper bound of 29K SEs, although since the MLAB RAM is shared with the resources used for the OR-gates there is a tradeoff between SE capacity and hardware fan-out. In this paper, we demonstrate experimental results for deployed designs of up to 24K SEs.

B. Reporting and Output Encoders

Any SE may be mapped to an accepting state, which causes it to generate a global output signal or "report" in all cycles in which it is active. Ideally the output buffer would accommodate a scenario where all states are configured as accepting states and all states are active in every cycle (easily achievable by setting the "start" and "reporting" flag on all SEs). However, this is not practical due to the bandwidth requirements needed ($freq \times N_{SEs} bits/sec$ using the variables from equation 1) (and this estimate doesn't include the additional volume of data needed to convey the encoded values of the accepting SE IDs, as described below).

NAPOLY must store the ID of any reporting SE, requiring an encoder for each potential report in a single cycle. We instance four 1024-to-10 reporting encoders per group of 1024 consecutive SEs,

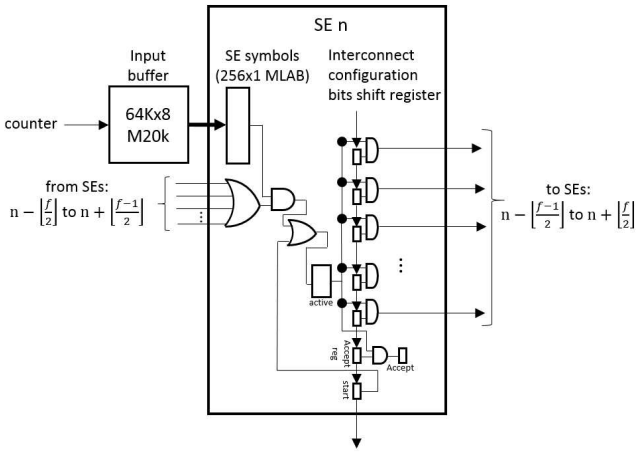


Fig. 3. SE Design.

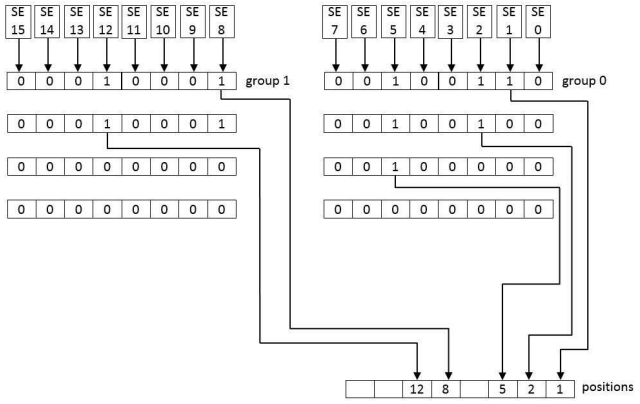


Fig. 4. Encoder Design.

limiting the number of simultaneous reports to $\frac{N}{1024} \times 4$ where N is the number of SEs.

As shown in Fig. 4. The encoders are priority encoders, which take the reporting bits from a group of 1024 SEs, encodes the right-most one-bit, zeros the bit, and then passes the remaining bits to the next encoder. Also, each array's output buffer has enough entries to store reports in $\frac{4\times1024}{N}$ % of cycles where N = the number of SEs.

C. Input Buffer

A 64K x 8-bit M20K-based RAM serves as the input buffer. Once filled, it streams input data into the SE array at one symbol per cycle (152 MB/s for the 4K-SE overlay). Filling the input buffer from DRAM requires $8.6\mu s$ (7.1 GB/s), performed $S/64K$ times, where S is the total number of input characters.

D. Overlay Configurations

Table I shows the Pareto optimal set of synthesized and placed-and-routed overlay configurations with respect to SE capacity and hardware fan-out. Hardware Fan-out (f) and $Fmax$ scale inversely with capacity. The column labeled **Max BW for** $N_{\%active} = 0.25(GB/s)$ gives the upper bound for on-chip memory bandwidth needed for 25% active states (assuming fully-populated adjacency vectors). Exploitation of on-chip memory bandwidth is the principle performance advantage of NAPOLY over CPU- and GPU-based approaches. The column labeled **Output Encoders** gives the number of output encoders, which determines the maximum number of "reports", or accepting state activations, allowed per clock cycle. Likewise, the column labeled **Max Reporting Cycles** gives the depth of the output buffer relative to the depth of the input buffer (64K). Together, these values and Fmax determine the maximum reporting rate of the overlay configuration, listed in the column labeled **Max Report Rate (GHz)**. The column labeled **Reconfig time (T)** lists the time needed to reconfigure a new NFA onto the overlay. Thus the execution time scales with $R \times T \times \frac{IS}{64K}$, where 64 KB = the size of the input buffer and IS is the size of the input data to be searched for patterns (note that this approximation of execution time does not include the time needed to flush the input or output buffer, which we incorporate into our thorughput Equation in Section III.E).

The columns labeled **Throughput for 24K states** and **Throughput for 128K states** lists the effective throughput for the benchmark, which includes the effect of the target overlays clock speed and reconfiguration time.

E. NAPOLY Runtime Behavior

NAPOLY follows the timing diagram shown in Fig. 5. For each block of input characters the array must fill the input buffer from DRAM ($\frac{size_{input\_buffer}}{bw_{DRAM}}$), and for each batch of SEs it must reconfigure its array ($time_{reconfig}$), flush the input buffer through the array ($time_{IBF}$), and flush the output to DRAM ($time_{OBF}$).

For a given NFA and input, the effective throughput is calculated according to Equation 2.

$$Throughput = \frac{size_{input\_buffer}}{\frac{size_{input\_buffer}}{bw_{DRAM}} + R \times (time_{reconfig} + time_{OBF} + time_{IBF})} \quad (2)$$

Fig.6 shows NAPOLY execution time is dominated by the time to flush input buffer and the time to flush the output buffer. In a future design we will use a double output buffer to overlap these times.

Fig. 7 plots the throughput of all NAPOLY overlays for 1 million input characters and for a total NFA workload from 4K to 128K states. Overlays with higher SE capacity perform better for larger NFAs, but for greater than 100,000 states the performance differential is only 10%, indicating that the choice of overlay configuration has an increasingly small impact for increasingly larger NFAs.

IV. SE Allocation Problem

Definition 1: For a given NFA $\{V, E\}$, where V is the set of states and E the set of edges (transitions), a **map** is an association between each of the NFA states of an NFA graph and a corresponding SE index in the range of $[0, N-1]$, where N = number of SEs. There are thus $|V|!$ unique maps for a given NFA assuming $|V| = N$.

The hardware fan-out determines the number of wire tracks to and from each SE, as well as the maximum "reach" of each SE in terms to maximum distance over which a connection can be made between two SEs: $i - j \leq \lfloor \frac{f-1}{2} \rfloor$ and $j - i \leq \lfloor \frac{f}{2} \rfloor$ for hardware fan-out f, for any edge in the NFA description $s \rightarrow d$ where state s is mapped to SE i and state d is mapped to SE j.

The hardware fan-out parameter is a constraint that defines which subset of maps are valid for a given NFA. In order to find a valid map, a mapping algorithm must solve the following problem.

Given a set of NFA edges $\{e : \forall (p, s) \in E\}$, find:

$$\left\{ map(p), map(s) : \begin{array}{cc} (map(p), map(s) \text{ are unique}) & \text{and} \\ (map(p) - map(s) \leq \lfloor (f-1)/2 \rfloor) & \text{and} \\ (map(s) - map(p) \leq \lfloor f/2 \rfloor) & \end{array} \right\}$$

TABLE I
REPERTOIRE OF ACHIEVED NAPOLY CONFIGURATIONS AND RESOURCE COST ON STRATIX 5 GX A7

# SEs	Hardware Fan-out (f)	Fmax (MHz)	Max BW for $N_{\%active} = 0.25(GB/s)$	Output Encoders	Max Reporting Cycles	Max Report Rate (GHz)	Reconfig time (T) (μs)	Throughput for 24K states (MB/s)	Throughput for 128K states (MB/s)
4K	103	152	1866	16	100%	2.4	21	14	3
8K	44	136	1427	32	50%	2.2	31	27	5
12K	25	122	1091	48	33%	2.0	43	32	6
16K	12	121	692	64	25%	1.9	53	36	9
20K	6	119	426	80	20%	1.9	67	31	9
24K	3	112	240	96	17%	1.8	74	67	11

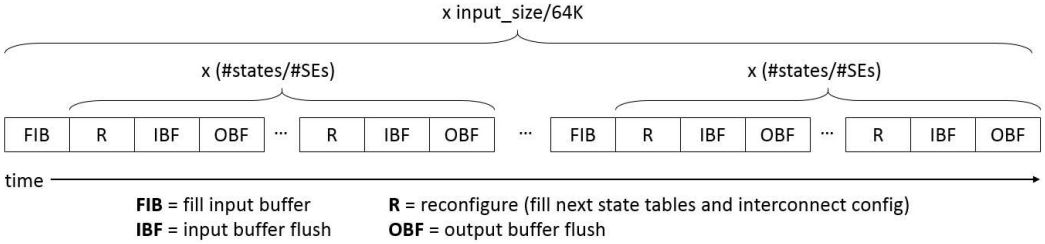


Fig. 5. NAPOLY Timing Diagram

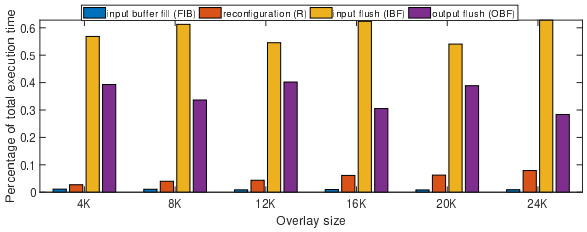


Fig. 6. Execution time makeup of NAPOLY.

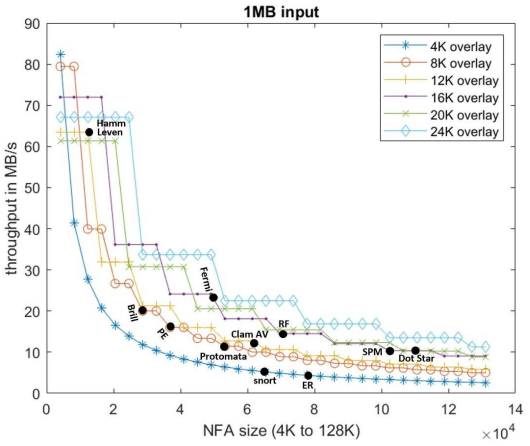


Fig. 7. NAPOLY Performance vs. NFA size.

Definition 2: For a given NFA graph $\{V, E\}$ and a given map, a **mapping violation** is any edge $(p, s) \in E$ where $(map(p) - map(s) > \lfloor (f-1)/2 \rfloor)$ or $(map(s) - map(p) > \lfloor f/2 \rfloor)$. In other words, a mapping violation occurs for each NFA edge whose predecessor and successor states are mapped to SEs whose indices are too far apart given the hardware fan-out of the target NAPOLY interconnect.

For a given map, our heuristic greedily finds and resolves each mapping violation. Our heuristic resolves each violation in order of ascending predecessor SE index by remapping either the predecessor or successor state in a way that minimizes the resulting mapping score.

The score function is computed as shown in Equation 3.

$$\sum_{(p,s)\in E} |\text{map}(p) - \text{map}(s)| \tag{3}$$

The score function is the accumulated mapped distance of the mappings of each predecessor-successor pair, where the distance is defined as the difference in SE index. The score is not directly affected by mapping violations, meaning that mapping A could have a lower score than mapping B when mapping A has more violations than mapping B.

We found that this approach gives the mapper flexibility to make decisions that potentially increase the number of mapping violations in order to achieve longer-term optimization. A consequence is that violations are likely to still exist after each pass through the SEs, in which case the heuristic will make additional passes as needed to resolve all violations.

The mapping heuristic is comprised of four subroutines: **validate_edges**, which returns the number of violations in a given mapping, **check_move**, which evaluates the mapping score difference of a proposed resolution, **move_state**, which modifies the placement of state, and **calculate_score**, which evaluates the mapping score over the set of a set of edges affected by proposed SE remapping.

validate_edges contains the top-level do-while loop, which iterates until there are no mapping violations. On each iteration, it validates the placement of each pair of states associated with each NFA edge.

For every mapping violation, **validate_edges** will evaluate the difference in score given by each of the $2 \times (f-1)$ potential resolutions, where f is the hardware fan-out. In other words, for every edge comprised of predecessor state p and successor state s, the

routine can fix the violation by either remapping s within the range of reachable SEs to p or remapping p within range of reachable SEs to s, where "within range" refers to any SE in the $f-1$ positions from $\lfloor \frac{f-1}{2} \rfloor$ locations less and $\lfloor \frac{f}{2} \rfloor$ greater than the target SE location. **validate_edges** eventually chooses one move that results in most positive or least negative impact on the score.

```
1  Function validate_edges():
       Input: NFA edges
       Output: NFA edges
2      do
3          for edge p → s in current SE assignment do
4              // check for a mapping violation
5              if ((p − s) < (−f−1)/2) || ((s − p) > f/2) then
6                  max_differential_score = - INT_MAX
7                  // evaluate each potential
                    solution to the violation...
8                  for k = −⌊(f−1)/2⌋ ... ⌊f/2⌋ do
9                      from = p
10                     to = s + k
11                     // ... by moving the
                        predecessor closer to the
                        successor
12                     max_differential_score =
                        check_move(from, to,
                        max_differential_score, best_from,
                        best_to)
13                 end
14                 for k = −⌊(f−1)/2⌋ ... ⌊f/2⌋ do
15                     from = s
16                     to = p + k
17                     // ... by moving the
                        successor closer to the
                        predecessor
18                     max_differential_score =
                        check_move(from, to,
                        max_differential_score, best_from,
                        best_to)
19                 end
20                 move_se(best_from,best_to)
21             end
22         end
23         // avoid getting suck in a local
            minema
24         if # of violations unchanged for 10 iterations then
25             make 10000 random moves
26         end
27     while fan-out constraint violations exist;
```

The **check_move** routine evaluates the effect of re-mapping a state in terms of its impact on the mapping score. Re-mapping a state from its original location in SE n to new location in SE m where $n < m$ (i.e. moving a state to a larger SE index) will affect any edge whose predecessor or successor state is mapped to SE $l : n \leq l \leq m$, or where $m < n$ (i.e. moving a state to a lower SE index) will affect any edge whose predecessor or successor is mapped to SE $l : m \leq l \leq n$.

move_SE performs a remapping operation on the graph by reassigning the state in SE index $from$ to SE index to. Moving a state in this way causes the states mapped in the range of SEs between $from$ and to to be shifted by one in order to fill the gap left by the state being moved.

```
1  Function check_move():
       Input: from, to
       Output: max_differential_score, best_to, best_from
2      // score for edges affected by the
        remapping
3      score = calculate_score(from, to);
4      // perform the remapping
5      move_SE(from, to)
6      // re-calculate score
7      differential_score = score - calculate_score(from,
        to)
8      // revert mapping to previous state
9      move_SE(from, to); // undo move
10     // check if the new score is better
        than the best found so far
11     if differential_score > max_differential_score then
12         max_differential_score = differential_score
13         best_to = to
14         best_from = from
15     end
```

This operation is depicted in Fig. 8. In this example, there is an edge connecting states "fifth" and "second" that are mapped to SEs n and m, respectively. Since $n > m$, the edge is oriented in the upward direction in the figure, in which higher-numbered SEs are lower as compared to lower-numbered SEs. The left side shows the original mapping state. Moving the state "fifth" from SE n to SE m causes all the states between them to shift down, as shown on the right side. This affects the mapping score contribution of any edges having successors or predecessors in the range of n to m.

```
1  Function move_SE():
       Input: from, to
       Output: NFA edges
2      if from < to then
3          for edge i → j do
4              if j == from then
5                  replace i → j with i → to
6              else if j > from && j ≤ to then
7                  replace i → j with i → j − 1
8              end
9          end
10     else
11         for edge i → j do
12             if j == from then
13                 replace i → j with i → to
14             else if j > to && j < from then
15                 replace i → j with i → j + 1
16             end
17         end
18     end
```

calculate_score accumulates the "distance" of all edges having successors or predecessors mapped to any of the SEs in a given SE range, where the distance is defined as the absolute difference in SE numbers corresponding to the states that comprise the edge. The mapping heuristic's objective is to minimize this score by mapping connected SEs into localized regions in the SE array.

A. Results

To evaluate the suitability of the mapping heuristic for realistic workloads, we mapped each of the NFA benchmarks in the ANM-

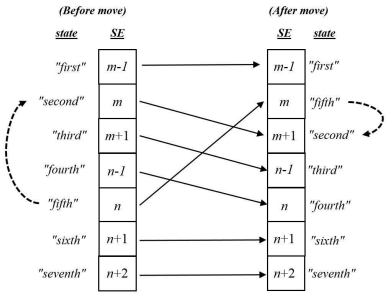


Fig. 8. Remapping SEs. Edge between state "fifth" and "second" is reassigned from SEs n and m, where $n > m$, to m and $m+1$ (after an operation "move SE n to m"). In this case, a movement from a higher-numbered SE to a lower-numbered SE causes all other SEs assignments between the two values to shift up, requiring an update to all other edges involving these SEs.

```
1 Function calculate_score():
      Input: from, to
2     sum = 0
3     for edge i → j such that (from ≤ i ≤ to || to ≤ i ≤ from
        ) || (from ≤ j ≤ to || to ≤ j ≤ from) do
4         sum = sum + |i − j|
5     end
6     return sum
```

LZoo benchmark suite [13], shown in Table II. A key goal of this work is to find the minimal hardware fan-out under which we can map each benchmark.

The mapping heuristic used will run infinitely when it cannot find a valid mapping, so it will abort execution when the derivative of the mapping score remains zero after several iterations of **validate_edges**, and try again with a larger hardware fan-out value. After finding a valid mapping, a suitable NAPOLY overlay is chosen based on the needed hardware fan-out. The overlay always has less SEs than states, but enough SEs to hold the largest distinct graph in the benchmark (all ANMLZoo benchmarks contain multiple distinct graphs).

Table II shows the mapping result for each of the ANMLZoo benchmarks. The **Minimum f Achieved** column lists the minimum hardware fan-out required for each benchmark based on our heuristic mapping algorithm. The **Target Overlay** is the largest overlay that can support the needed fan-out. **# NAPOLY reconfigurations** is computed $\lceil \frac{S}{N} \rceil$. The columns labeled **Throughput** lists the effective throughput for the benchmark, which includes the target overlay's clock speed and reconfiguration time.

For each of the ANMLZoo benchmarks, Table III shows the performance of competing CPU and GPU automata processing frameworks. The CPU implementation is Intel Hyperscan [14] measured independently by the authors using a 3.1 GHz Intel i5-4440 CPU with 32 GB RAM. The GPU implementation is iNFAnt2 executed on an Nvidia Titan Xp as reported in [13].

In order to understand the relationship between the NFA and its corresponding performance on the CPU and GPU implementations, the table also lists runtime data for each benchmark: the average number of active states (active set) and total number of reports as reported in [13].

The rows of the table are sorted in descending order according to NAPOLY speedup relative to the best of the GPU and CPU results. NAPOLY performs best for larger benchmarks with more active states and is faster than both the GPU and CPU NFA implementations in 7 of the 12 benchmarks, while the GPU implementation is fastest in 4 and the CPU implementation is fastest in 1. NAPOLY's average speedup is 4.4.

B. Overlay Scalability

As shown in Equation 2, NAPOLY throughput depends on (1) the number of reconfigurations needed, which may be reduced by having a larger overlay with more interconnect density, (2) the time to flush the input buffer, which depends on clock speed, and (3) reconfiguration time, which depends on DRAM bandwidth.

Table IV shows the NAPOLY capability when scaled up to an Intel Stratix 10 GS. In general, the Stratix 10 offers roughly a doubling of overlay capacity, clock rate, and DRAM bandwidth, which according to Equation 2 would result in a speedup of approximately $\frac{1}{\frac{1}{2}+\frac{1}{2}\times\frac{1}{2}} = 1.33$

V. Prior Work

This section summarizes prior work in four related areas: (1) methods for synthesizing automata-type architectures onto an FPGA fabric, (2) applications that benefit from such architectures, (3) open source automata models and architectures, and (4) tools and methods for optimizing automata descriptions.

A. Synthesis of NFAs and Regular Expressions

FPGA implementation of regular expression matchers are often inspired by networking applications, and some of these are based on automata-based architectures [15]. A challenge for these approaches is the high cost of reconfiguring the FPGA to change or update the target NFA. Prasanna et al developed early methods for synthesizing regular expressions onto both FPGAs and a conceptual Self-Reconfigurable Gate Array (SRGA) device [16]. Their original approach bypassed logic synthesis and directly targeted the low-level FPGA fabric. However, as FPGA architecture evolved in complexity, this approach became infeasible. Their second design targeted HDL but introduced additional optimization methods for both the NFA descriptions and generated architecture [5], [6]. Similar efforts have produced more dense designs but still suffer from long reconfiguration times [17]. Becchi et al developed a set of techniques for optimizing both NFA and DFA-based architectures [18], [19], including several approaches to identify and explore design parameters that have the most significant impact on the performance and cost of the corresponding NFA and DFA implementation. Examples of these include alphabet size, number of inputs read per cycle (stride), and storage of next state tables in logic and/or RAM. There are previous efforts to overcome the high cost of synthesizing automata into an FPGA fabric. Like NAPOLY, they allow a user to quickly change the NFA description for an in-place FPGA configuration. However, while NAPOLY allows updates of both the NFA topology and edge labels, these earlier efforts are limited to only edge labels, leaving the NFA toplogy fixed [20], [21].

B. Open Source Automata Processor Architectures, Simulators, and Benchmarks

Wadden et al. developed a VPR-derived [22] place-and-route tool that targets a conceptual Automata Processor fabric [23]. This tool serves as an experimental framework with which to explore the impact of routing algorithms and interconnect design on performance and efficiency. Using this tool they compared the hierarchical design of the AP routing matrix to a non-hierarchical mesh-based network-on-chip and concluded that the ideal interconnect architecture depends on the input NFA topology. The same group compiled a suite

TABLE II
ANMLZOO BENCHMARKS AND THEIR NAPOLY MAPPINGS

Benchmark	# States (S)	Minimum f Achieved	Overlay Size (N)	# Reconfigurations	Reconfiguration Time Throughput (MB/s)
Hamming	11346	21	12K	1	63
Levenshtein	2784	17	12K	1	63
Fermi	40783	8	16K	2	24
Brill	26668	40	8K	4	20
ClamAV	49538	18	12K	5	13
DotStar	96438	4	20K	5	12
PowerEN	40513	29	8K	5	16
RandomForest	75340	12	16K	5	15
SPM	100500	8	16K	5	10
Protomata	42061	42	8K	6	13
Snort	69029	60	4K	17	5
ER	95136	62	4K	19	3

TABLE III
PERFORMANCE RESULTS

Benchmark	States (S)	NAPOLY Throughput (MB/s)	Ave. Active States (AS)	GPU Throughput (MB/s)	CPU Throughput (MB/s)	Speedup vs max(GPU,CPU)
SPM	100500	10	6331	0.5	0.1	20.0
Fermi	40783	24	3854	2	1	12.0
RandomForest	75340	15	968	2	0.5	7.5
Hamming	11346	63	240	18	10	3.5
Brill	26668	20	14	7	1	2.9
Protomata	42061	13	19	5	1	2.6
Levenshtein	2784	63	88	38	1	1.7
ClamAV	49538	13	4	4	14	0.9
EntityResolution	95136	3	10	4	1	0.8
Snort	69029	5	98	14	0.4	0.4
DotStar	96438	12	3	40	10	0.3
PowerEN	40513	16	31	53	10	0.3

TABLE IV
REPERTOIRE OF ACHIEVED NAPOLY CONFIGURATIONS AND RESOURCE COST ON STRATIX 10 GS

# SEs	Hardware Fan-out	Output Encoders	Max Reporting Cycles	Max Report rate (GHz)	Fmax (MHz)	Max BW for $N_{\%active} = 0.25(GB/s)$
4K	254	16	100%	4.64	290	8746
8K	126	32	50%	8	250	7510
12K	83	48	33%	12	250	7331
16K	62	64	25%	13.4	210	6208
20K	49	80	20%	15.2	190	5549
24K	40	96	17%	16.32	170	4863
28K	34	112	14%	16.8	150	4255
32K	30	128	12%	16.64	130	3719
36K	26	144	11%	15.84	110	3068
40K	23	160	10%	14.4	90	2467
44K	21	176	9%	12.32	70	1744
48K	19	192	8%	9.6	50	1072

of NFA benchmarks called ANMLZoo containing a representative example of an NFA description, sample input, and expected outputs for every publicly-released application for the AP as well as two synthetic benchmarks [13]. They also developed open source tool that can simulate the evaluation of arbitrary ANML descriptions and perform basic transformations to NFA such as elimination of counters and Boolean elements and use of state replication to limit the maximum in-degree (fan in) and out-degree (fan-out) of the NFA [24]. Fang et al. designed the Unified Automata Processor (UAP), a set of vector extensions added to a traditional von Neuman CPU optimized for implementing a variety of NFA-based programming models [8]. The UAP exploits parallelism by concurrently traversing one edge per cycle for each of its 64 lanes. The design stores NFA transitions in local memory attached to each lane, comprising 1 MB in total. The transitions are stored in a compact, efficient format but the design is limited to NFAs that can fit into the local memory. Wadden and al. proposed a modified Micron AP Reporting Architecture to reduce AP overhead and stall cycles during dense reporting activity [25]. The modified AP reporting region consists of 64 16-bit sub-RA (Reporting Aggregation) equivalent to one 1024-bit RA in Micron AP, all gathered in an arbitration unit. Along with reporting aggregation, there is a shared 64-bit mega tag component to report

the symbol offset. This architecture improves the reporting sparsity of some ANMLZoo benchmarks and keeps same performance for other benchmarks which have dense of reporting.

C. Comparative Studies of NFA Implementations

GPU-, FPGA-, and ASIC-based Automata Processors require preprocessing overhead when processing a new NFA description. Depending on the application, this overhead may be performed offline or at runtime. CPU- and GPU-based approaches are able to process NFAs stored in DRAM and are generally less affected by preprocessing time, but their traversal time–especially for larger NFAs–is limited by their cache performance. Since the behavior of automata processors is dependent on both the NFA structure and input stream, performance comparisons between competing architectures is difficult. Becchi et al. characterized the performance of GPU, AP, and FPGA-based automata processing approaches, finding that FPGAs offer a traversal throughput of 2 to 3 times that of the AP and 80 to 1000X that of a GPU at the cost of extremely high preprocessing time. In this analysis, the preprocessing time including a pass through the FPGA synthesis and place-and-route design flow [26]. MNCart is a recently-proposed comprehensive central ecosystem for automata tools to simplify the comparisons between the CPU, GPU, and AP platforms [27] proposed. MNCart system includes a new JSON-based network language MNRL for representing NFA.

VI. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1421059.

VII. Conclusion and Future Work

This paper describes NAPOLY, an automata processor overlay. NAPOLY is parameterized, allowing for tradeoffs in state capacity, interconnect density, and output buffer size. These tradeoffs allow for offline generation of a repertoire of overlays that allow for the overlay to be customized for specific types of NFAs. Once an overlay is deployed, the user can rapidly program the NFA at runtime, supporting arbitrarily large NFAs. The performance results include the time required to program the overlay from DRAM and are competitive with the state-of-the-art CPU implementation from Intel and the state-of-the-art GPU implementation. Further, they show that NAPOLY's performance scales with on-chip memory capacity, and in future work NAPOLY's scale ability on larger FPGAs or multi-FPGA platforms, such as those available in the cloud will be evaluated. NAPOLY spends over half its time flushing its input buffer into the SE array and nearly half its time flushing its output buffer to DRAM. It is possible to perform these steps in parallel if reports are written to DRAM immediately after being generated from the SE array. This change is planned for the next version of NAPOLY.

References

[1] I. Roy and S. Aluru, "Finding motifs in biological sequences using the micron automata processor," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, 2014.

[2] M. C. et al, "Debugging support for pattern-matching languages and accelerators," in *ASPLOS*, 2019.

[3] D. G. et al., "A scalable multithreaded l7-filter design for multi-core servers," in *ANCS'08*, 2008.

[4] K. Wang, M. Stan, and K. Skadron, "Association rule mining with the micron automata processor," in *Proceedings of the IEEE 29th International Parallel and Distributed Processing Symposium*, 2015.

[5] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on fpga," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[6] Y.-H. Yang and V. Prasanna, "High-performance and compact architecture for regular expression matching on fpga," in *IEEE Transactions on Computers*, vol. 61, no. 7, 2012.

[7] M. Becchi and C. Patrick, "Data structures, algorithms and architectures for efficient regular expression evaluation," in *Washington University, St. Louis, MO*, 2009.

[8] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *Proceedings of MICRO-48*, 2015.

[9] K. Peng, S. Tang, M. Chen, and Q. Dong, "Chain-based dfa deflation for fast and scalable regular expression matching using tcam," in *Seventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2011.

[10] A. S. et al., "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. ACM,*, 2017.

[11] J. Y. et al., "Time-division multiplexing automata processor," in *In Design, Automatation and Test in Europe 2019 IEEE*, 2019.

[12] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, 2014.

[13] J. W. et al, "Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[14] K. Angstadt and et al., "Mncart: An open-source, multi-architecture automata-processing research and execution ecosystem," in *IEEE Computer Architecture Letters*, 2017.

[15] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, 2006.

[16] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[17] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "Reapr: Reconfigurable engine for automata processing," in *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[18] M. Becchi and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[19] Chen and e. a. Xinming, "Picking pesky parameters: Optimizing regular expression matching in practice," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, 2016.

[20] J. Teubner, L. Woods, and C. Nie, "Skeleton automata for fpgas: reconfiguring without reconstructing," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 229–240.

[21] R. Moussalli, M. Salloum, R. Halstead, W. Najjar, and V. J. Tsotras, "A study on parallelizing xml path filtering using accelerators," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, pp. 93:1–93:28, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560040

[22] V. Betz and J. Rose, "Vpr: a new packing, placement and routing tool for fpga research," in *Proceedings of the 1997 International Workshop on Field Programmable Logic and Applications*, 1997.

[23] J. Wadden, K. S. Khan, and K. Skadron, "Automata-to-routing: An open-source toolchain for design-space exploration of spatial automata processing architectures," in *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2017.

[24] J. Wadden and K. Shadron, "Vasim: An open virtual automata simulator for automata processing application and architecture research," in *Technical Report CS2016-03, University of Virginia, 2016*, 2016.

[25] J. Wadden, K. Angstadt, and K. Skadron, "Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures," in *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA'18)*, 2018.

[26] M. Nourian, X. Wang, W. F. X. Yu, and M. Becchi, "Demistifying automata processing: Gpus, fpgas or micron's ap?" in *Proceedings of the International Conference on Supercomputing*, 2017.

[27] K. A. et al., "Mncart: An open-source, multi-architecture automata-processing research and execution ecosystem," in *IEEE Computer Architecture Letters 17.1*, 2018.