NAPOLY: A Non-deterministic Automata Processor OverLaY

RASHA KARAKCHI and JASON D. BAKOS, University of South Carolina

Deterministic and Non-deterministic Finite Automata (DFA and NFA) comprise the core of many big data applications. Recent efforts to develop Domain-Specific Architectures (DSAs) for DFA/NFA have taken divergent approaches, but achieving consistent throughput for arbitrarily-large pattern sets, state activation rates, and pattern match rates remains a challenge. In this article, we present NAPOLY (Non-Deterministic Automata Processor OverLaY), an FPGA overlay and associated compiler. A common limitation of prior efforts is a limit on NFA size for achieving the advertised throughput. NAPOLY is optimized for fast re-programming to permit practical time-division multiplexing of the hardware and permit high asymptotic throughput for NFAs of unlimited size, unlimited state activation rate, and high pattern reporting rate. NAPOLY also allows for offline generation of configurations having tradeoffs between state capacity and transition capacity. In this article, we (1) evaluate NAPOLY using benchmarks packaged in the ANMLZoo benchmark suite, (2) evaluate the use of an SAT solver for allocating physical resources, and (3) compare NAPOLY's performance against existing solutions. NAPOLY performs most favorably on larger benchmarks, benchmarks with higher state activation frequency, and benchmarks with higher reporting frequency. NAPOLY outperforms the fastest of the CPU and GPU implementations in 10 out of 12 benchmarks.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; **Concurrent computing methodologies**; • **FPGA, automata processing, FPGA overlay, pattern matching**;

Additional Key Words and Phrases: FPGA, automata processing, FPGA overlay, pattern matching

ACM Reference format:
Rasha Karakchi and Jason D. Bakos. 2023. NAPOLY: A Non-deterministic Automata Processor OverLaY. *ACM Trans. Reconfig. Technol. Syst.* 00, JA, Article 00 (May 2023), 25 pages.
https://doi.org/10.1145/3593586

1 INTRODUCTION

Pattern-based datasets such as genomic sequences, item-sets, web data, and network packets are growing rapidly in size and complexity. Identifying complex patterns are often involved in applications such as motif discovery [28], de novo genomic assembly [24], web-search and ranking [4], question answering systems [8, 23], compression in NoSQL systems [18, 25], approximate string matching [13], calculating the edit distance between two genomic sequences [31], signature-based threat detection [6], association rule mining [12], and data-packet inspection [6]. Such pattern matching computations are often reducible to the simulation of either **Deterministic Finite Automata** (**DFA**) or **Non-deterministic Finite Automata**(**NFA**).

This material is based upon work supported by the National Science Foundation under Grant No. 1421059.
Authors' addresses: R. Karakchi and J. D. Bakos, University of South Carolina, Columbia, SC 29208; emails: {karakchi, jbakos}@cec.sc.edu.

1936-7406/2023/05-ART00 $15.00
https://doi.org/10.1145/3593586

DFA allows only one active state at any time, and the DFA structure must contain a state corresponding to every possible partial match of every pattern to be accepted. An NFA, on the other hand, allows an arbitrary number of active states and thus allows multiple next-state functions to operate concurrently.

Although pattern matching problems can be solved equivalently using either a DFA or NFA, neither can be efficiently performed on a CPU. DFAs require extremely large transition tables with an unpredictable access pattern, while NFAs require memory bandwidth that scales with state activation rate, often becoming memory bound. For this reason, there is widespread interest in **Domain-Specific Architectures** (**DSA**) that can efficiently exploit the NFA parallelism.

Automata processor DSAs are generally evaluated by their achieved symbol throughput but the reported values often assume that the entire NFA fits in on-chip memory. This is equivalent to evaluating a memory system using the bandwidth of only its on-chip cache. Such systems may require milliseconds [3] to seconds [2, 20, 22, 36] to reprogram, making rapid context switching impractical.

2 PRIOR WORK

Previous work in automata matching DSA architectures have been deployed on both **Field-Programmable Gate Arrays** (**FPGA**) and **Application-Specific Integrated Circuits** (**ASICs**). Most of these can achieve high traversal throughput of one or two input symbols per clock cycle but processing NFAs that are too large to fit in device memory requires multiple passes of the input stream. The latency of reprogramming between consecutive passes is often a significant performance limiter. This is the main challenge NAPOLY was designed to address.

When using FPGAs, the patterns are often synthesized directly onto the FPGA fabric as logic circuits, allowing high pattern density and high throughput (100*s* of MB/s). However, even if the logic is pre-synthesized offline, the fine granularity of reconfiguration requires long reconfiguration times of 10s of seconds making it impractical to time-multiplex the hardware [2, 20, 22, 36].

On the other hand, ASIC-based architectures such as the **Micron Automata Processor** (**Micron AP**), allow the input data to be streamed into multiple functional units where each functional unit tracks partial pattern matches. Such designs allow for faster re-configuration time than FPGA-based approaches since only a more compact, abstract form of the patterns need to be loaded. However, their fixed structure lacks the ability to leverage the patterns themselves to make design tradeoffs, i.e., trade between state density and transition density in the corresponding automata [3, 20].

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 [20]. 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, with a total capacity of 1 MB. The transitions are stored in a compact, efficient format but the design is limited to NFAs that can fit into the local memory.

Das et al. [5] proposed **Cache Automaton** (**CA**), in which a conventional cache is augmented to perform automata processing through the addition of two pipeline stages fed by each input symbol. The first stage finds the symbol match in the RAM and the second implements the state transitions through a hierarchical switching network. The achieved throughput degrades with the targeted NFA's edge density and number of states. Subsequent efforts have sought to address this problem. J et al. [10] use a time-division multiplexing approach by adding a multiplexer to pipeline the hierarchical switching network. This approach improved cache automata throughput by 2X.

Another approach for implementing DFAs and regular expressions is by using **Ternary Content-Addressable Memory** (**TCAM**). TCAM-based approaches have limited capacity and a–far as we are aware–have not been demonstrated to be amenable to time multiplexing [11].

Teubner et al. [30] implemented an FPGA-based automata engine for database systems by integrating the FPGA hardware as an XML projection (or pre-filtering) into a database system path. XML projection [17] extracts filtering expressions from query then pre-filters the data to reduce the dataset size and the compilation overhead. The hardware can be reconfigured in less than one microsecond, but is currently only capable of matching individual patterns and cannot be adapted to processing generalized automata descriptions such as the one defined in ANMLZoo.

CAMA-T [21], Impala [7], and Grapefruit [19] are based on multi-stride NFA to accept multiple input symbols per clock cycle. Impala is an ASIC-implementation that transforms the usual 8-bit symbol used by many automata processors into a 4-bit symbol to further improve throughput, while Grapefruit is an FPGA-implementation which has the ability to execute four 8-bit symbols at a time. Unfortunately, this article does not provide throughput values for each of the ANMLZoo benchmarks to provide a direct comparison.

Wang et al. recently proposed hAP, a spatial-von Neumann Automata Processor that consists of a hybrid DFA/NFA engine, where the DFA component is designed as state-match component that accepts one active state at a time and the NFA executes the transitions which are deployed directly to gates and registers [35]. The reported throughput includes the kernel throughput and the compressed reporting overhead, however, reconfiguration time is not considered. Additionally, hAP targets only one specific type of automata application, regular expression matching.

This work is comprised into three main contributions:

(1) a parameterizable overlay, NAPOLY, which is comprised of an array of hardware modules (called State Transition Elements or STEs), each sensitive to a specific pattern and reconfigured at run time in 21 to 74 μs depending on the overlay size selected,
(2) an open-source tool, NFATOOL, which maps logical pattern states onto the physical STEs using a SAT solver [27],
(3) analysis of the tradeoffs between state capacity, interconnect density, output buffer size, and
(4) comparison to state-of-the-art Intel's CPU-based NFA software (Hyperscan) and a well-known GPU-based implementation (iNFAnt) [14].

3 FINITE AUTOMATA

A finite automaton (FA) M is defined by [26].

$M = (Q, \sum, \delta, q_0, F)$, where

— Q is finite set of states,
— $\sum$ is a finite set of symbols called the input alphabet,
— $\delta : \begin{cases} Qx\sum \rightarrow Q, & \text{Transition Function for DFA,} \\ Qx(\sum \cup \lambda) \rightarrow 2^Q, & \text{Transition Function for NFA,} \end{cases}$
— $q_0 \in Q$ is the initial state,
— $F \subseteq Q$ is a set of reporting states.

At each clock cycle, the FA makes a transition based on (1) current state activation and (2) the match of the input symbol and the edge label. The FA must report whenever a "report" state is activated, meaning that a pattern defined in the pattern set was identified. In this case, both the report ID and the current symbol position (offset) in the input sequence are reported. An FA is classified either as DFA or NFA depending on how many states may be active at one time.

3.1 Deterministic Finite Automata (DFA)

During operation, a DFA may have only one active state and accesses only one entry of its state transition table. It must contain a state for every possible partial match of every possible pattern. This can lead to combinatorial growth of the state space.

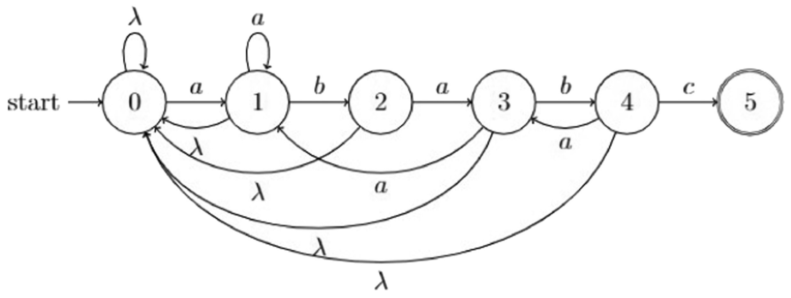


Fig. 1. DFA for regular expression pattern “ababc”.

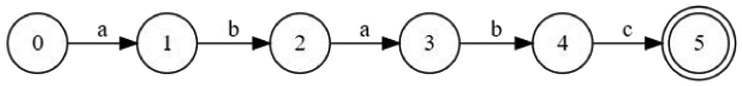


Fig. 2. NFA for regular expression pattern “ababc”.

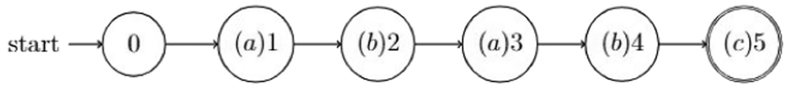


Fig. 3. ANML-NFA for regular expression pattern “ababc”.

Figure 1 shows an example of DFA consisting of six states (state 0 represents start-state and state 5 represents report-state) that recognizes a simple regular expression pattern “ababc”. The corresponding next state table is shown as Table 1.

3.2 Nondeterministic Automata

In an NFA, multiple states are active simultaneously. Each state needs only track the progress towards accepting one specific pattern instead of all possible patterns. This requires fewer states than an equivalent DFA.

Figure 2 shows an NFA that accepts the same pattern as in Figure 1. As it is shown in Table 2, the next state table for NFA is 2.6 times smaller than that of the DFA in Table 1.

An alternative form of NFA description called **Automata Network Markup Language** (**ANML**) was developed by Micron [3]. ANML-NFA is differentiated by associating the transition labels with the states instead of the edges. This adds an additional constraint that each state’s incoming transitions must have the same label set, but it allows an implementation to associate the next state table with the states instead of the edges and thus reducing the memory requirement.

Figure 3 shows the alternative form of NFA with symbols associated with states, for implementing the pattern “ababc”.

3.3 Intel FPGA

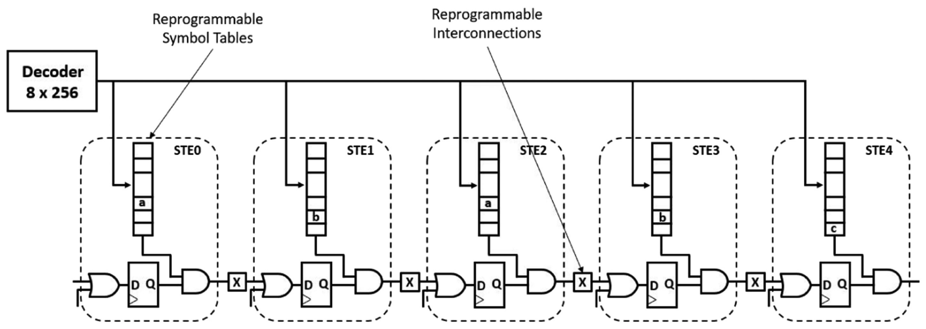
An Intel FPGA is comprised of a two-dimensional array of **Logic Array Blocks** (**LABs**). In the Stratix 5 family of FPGAs, each LAB consists of 10 basic reconfigurable **Adaptive Logic Modules** (**ALMs**) sharing local interconnections, control signals, and chain of connection lines. The ALM consists of two 6-input **Look-Up Tables** (**LUTs**), two-adders, four multiplexers, and four registers. Some LABs are variants called MLABs (Memory LAB), which contain LUTs-based SRAM capability

Table 1. Transition Table for DFA Description

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

Table 2. Transition Table for NFA Description

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



Q1

Fig. 4. Mapping “ababc” in Figure 2 directly to Micron AP.

to support simple dual-port SRAM. LABs connect to each other through global interconnections distributed horizontally and vertically on the device.

Prior work [2, 22] implemented NFA as circuits and flip-flops as illustrated in Figure 5, showing an automata that accepts pattern “ababc” implemented on an FPGA. This approach has fixed interconnection and fixed symbol tables, which requires that the FPGA be reconfigured to change the recognized patterns.

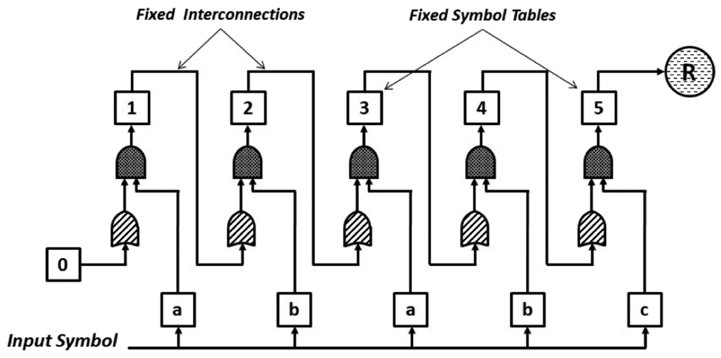


Fig. 5. Mapping NFA directly on FPGA of Figure 3.

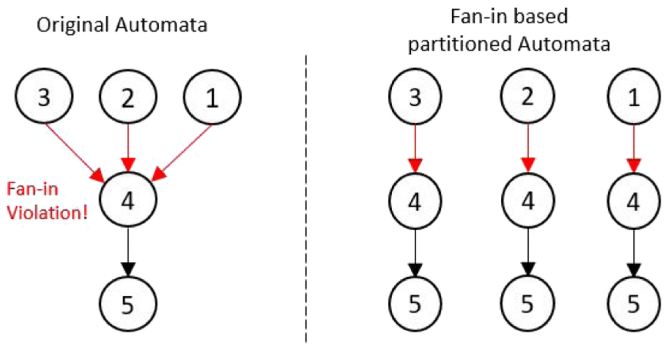


Fig. 6. An example of fan-in-based relaxation.

3.4 VASIM Relaxation

Mapping an arbitrary automata onto NAPOLY often requires transforming automata into another functionally equivalent automata but having different structure. One NFA transformation is fan-in and fan-out relaxation [33], replicates each state having a fan-in (incoming transitions) that exceed a given limit, resulting in the transformed NFA having a desired maximum fan-in in order to enforce constraints imposed by the hardware.

Figure 6 shows an original automaton of five vertices having a maximum fan-in of 3. If we wish to limit the maximum to, for example, 1, then state 4 would be a violation of this constraint. With fan-in relaxation, the violated state is replicated by $ceil(I/d)$, where I = the original fan-in and d = the fan-in constraint. The outputs of the original vertex are copied, while the inputs are divided among the new replicated vertices. Likewise, fan-out relaxation allows for constraining the application of a fan-out constraint.

Figure 7 shows an original automaton of five vertices, where its maximum logical fan-out is $O = 3$. Assuming the logical fan-out $d = 1$, state 2 violates the hardware fan-out constraint. During relaxation, the violated state is replicated by *ceil(O/d).* The outputs of the original vertex are divided among the new replicated vertices, and the inputs are copied.

3.5 ANMLZOO Benchmark Suite

ANMLZoo is a diverse benchmark suite of finite automata for evaluating automata processing engines [9]. It consists of 12 benchmarks representing various applications for automata processing.

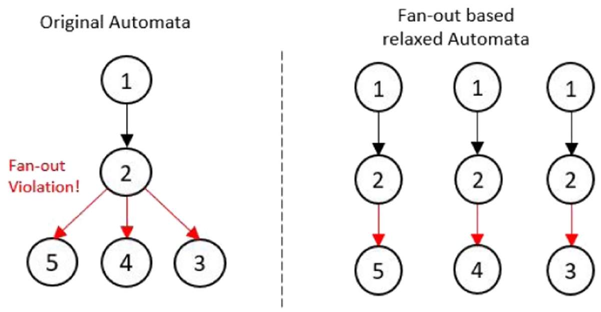


Fig. 7. An example of fan-out-based relaxation.

Table 3 shows ANMLZoo benchmarks which have up to 100,000 and up to 5,000 distinct subgraphs, which are connected to each other to form ANMLZoo graph. The first column lists the benchmark names, the second and third columns show the number of states in (1,000's) and the number of distinct subgraphs. The fourth column shows the maximum logical fan-in/ logical fan-out for each benchmark, which represents the maximum incoming and outgoing transitions of state. The Family column represents the family to which each benchmark belongs: regex (set of characters that define search pattern), mesh (regular structure with fan-in/fan-out), and widget (when automata represented as a tree). The last column, function, describes the function each benchmark performs.

4 NAPOLY DESIGN

NAPOLY is a parameterizable and reusable architecture for deploying an NFA description but imposes several constraints. First, the NFA topology must not violate a fan-in and fan-out constraint. However, the fan-in and fan-out constraint not only limits the number of incoming and outgoing edges but also limits the distance between the mapped location of states that are connected with an edge. We refer to this constraint as the "hardware fan-out", which determines the maximum number of outgoing transitions per STE as well as the maximum distance between a pair of connected STEs with respect to their location in the array. For example, with a hardware fan-out of 10, STE_n can only connect to STE_{n-4} to STE_{n+5} (including itself).

NAPOLY configurations are a tradeoff between hardware fan-out and state capacity, in terms of the number of STEs. We developed several Pareto optimal versions of the overlay with varying numbers of STEs and hardware fan-out [15].

4.1 STE Design

Figure 8 shows the NAPOLY STE design. To achieve maximum utilization of memory, the "current state table", which stores the set of input symbols associated with each STE, is generated as a *256 x M bit* RAM, where *M = the number of STEs*. Each STE accepts a one-bit input from its corresponding column in the current state table, indexed by the input symbol.

Each STE contains an OR-gate that combines activation signals from all its f possible predecessor STEs (f is the hardware fanout). Any cycle in which any of the incoming activation signals are asserted while simultaneously receiving a one-bit from the current state table will activate the STE's state bit in the following cycle. Unless the "start bit" is set, the state bit resets in any cycle in which this condition does not hold.

While the state bit is set, the STE will broadcast an activation signal to all its f outputs, each of which is AND'ed against a corresponding "interconnect configuration bit" before connecting to

Table 3. ANMLZoo Applications

Benchmark	**States** (K)	**Distinct Sub-graphs**	**Logical Fan-in / Fan-out**	**Family**	**Function**
Brill	26	1,962	4/4	Regex	brill tag patterns and correct tags
ClamAV	48	515	11/2	Regex	viruses signatures in files
Snort	69	2,585	19/5	Regex	particular snort rules
Protomata	42	2,340	3/106	Regex	particular motif signature
Dotstar	96	2,837	2/2	Regex	spy rules
Power En	40	2,857	4/3	Regex	complex rules
Levenshtein	27	24	8/5	Mesh	edit distance between DNA sequence
Hamming	11	93	4/2	Mesh	number of mismatches between sequences
SPM	100	5,025	3/2	Widget	groups of related items
Fermi	40	2,399	2/2	Widget	particular path
Entity Resolution	95	1,000	28/2	Widget	input sequences match encoded pattern
Random Forest	75	3,767	2/2	Widget	Recognize particular handwritten texts

the OR-gate of each of the potential successor STEs. The interconnect configuration bits are the mechanism by which edges are established between states mapped onto STEs. It forms a point-to-point programmable interconnect, in which each wire and corresponding configuration bit are associated with each pair of connectable STEs.

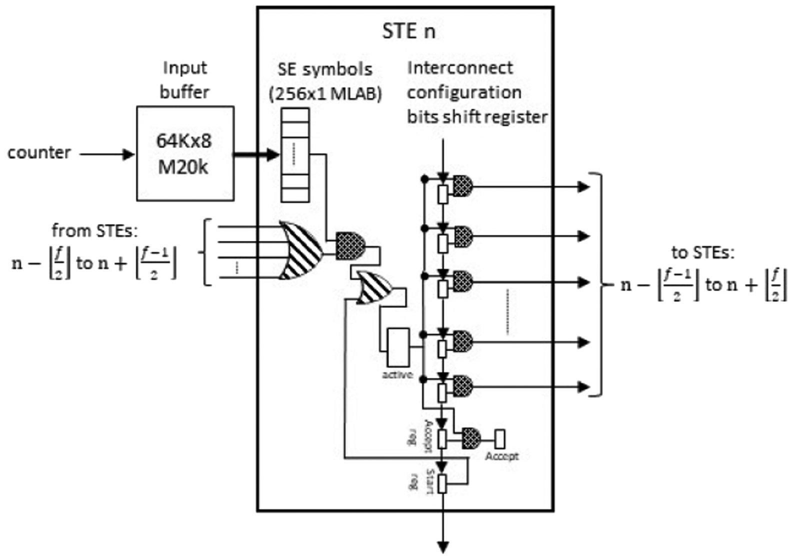

STE n
Input buffer
SE symbols (256x1 MLAB)
Interconnect configuration bits shift register
counter
64Kx8 M20k
from STEs:
$n-\lfloor\frac{f}{2}\rfloor$ to $n+\lfloor\frac{f-1}{2}\rfloor$
to STEs:
$n-\lfloor\frac{f-1}{2}\rfloor$ to $n+\lfloor\frac{f}{2}\rfloor$
active
Accept reg
Accept
Start reg

Fig. 8. STE design.

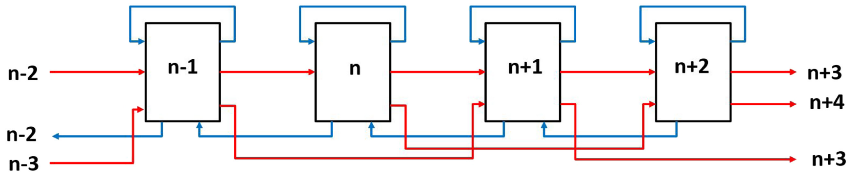


Fig. 9. NAPOLY interconnect.

The interconnect configuration bits and the start and reporting flags are stored in a set of flip-flops connected as one of several shift registers that collectively spans all STEs, similar to the JTAG boundary scan but a set of which are fed in parallel from DRAM as opposed to serially as in the case of JTAG.

4.2 Interconnection Design

The physical STEs on the FPGA are connected using point-to-point links between each STE and to itself and between each STE and *f-1* of its neighbors. The STEs adopt a one-dimensional addressing scheme, where each STE is associated with ID n and sends output signals to successor STEs $n - [(f - 1)/2]$ to $n + [f/2]$.

Figure 9 shows NAPOLY interconnects when $n = 4$, and $f = 4$. The blue and red wires represent the backward and forward interconnects respectively. This interconnect design is based on dedicated, non-shared point-to-point wires between each pair of connectable STEs. While it is less versatile than a switched interconnect consisting of shared wire tracks, it avoids the need to allocate and map interconnect resources, relying instead only on solving only the state-to-STE mapping problem.

4.3 Overlay Resource Constraints

STE capacity is limited by the LUTs required to implement the OR-gates that combine the incoming predecessor inputs into each STE. Our evaluation FPGA is an Intel Stratix 5 GX A7.

In terms of RAM capacity, there is a choice between using MLAB or M20K RAMs for the current state tables. The Stratix 5 GX A7 has roughly 7X the M20K capacity than it does MLAB capacity, but the current state tables have a depth of 256, 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. More importantly, M20Ks require synchronous reads, which if used for the current state table would reduce array throughput by 1/2, as each input symbol would require one cycle to access the current state table and another for updating the state flip-flops. Lastly, the M20K blocks are needed to buffer the input and output data for the AP fabric. The Stratix 5 GX 7A contains 7.16 Mb of MLAB memory, giving an upper bound of roughly 29K STEs.

4.4 I/O Interface

The input symbols stored initially in the DRAM are transferred into the input buffer. The outputs reported in NAPOLY are stored in output buffers before flushing out to the DRAM.

4.4.1 Input Buffer. A 64K x 8-bit M20K-based RAM serves as the input buffer. Once filled, it streams input data into the STE array at one symbol per cycle at $152MB/s$ for the 4K-STE overlay. Note that this is the upper bound for throughput, but in practice, the effective throughput is lower due to the overheads required for programming the array, filling the input buffer, and flushing the output buffer. Filling the input buffer from DRAM requires $8.6\mu s(7.1GB/s)$, performed $S/64K$ times, where S is the total number of input characters.

4.4.2 Output Buffer and Report Region. Any STE may be mapped to a particular reporting 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, which is readily achievable by setting the "start" and "reporting" flag on all STEs.

To obtain the reporting ID, NAPOLY's STEs are decomposed into output regions, where each region represents a group of consecutive STEs (M). The number of reporting regions in the design is equal to (N/M), where N is total number of STEs. To determine which STE is reporting in each group, we use a priority encoder. The number of encoders determines the maximum number of reports per clock cycle without stalling.

4.4.3 Output Buffer Implementation. The depth and width of the output buffers are design parameters. The buffer depth depends on overlay size, where smaller overlays can support deeper output buffers. The output buffer depth for overlay 4K, 8K, 12K, 16K, 20K, and 24K is 64K, 32K, 24K, 16K, 12K, and 8K, respectively.

For example, assume an 8K overlay comprised of eight 1024-STE reporting regions with four encoders per region, giving 32 encoders. STEs are enumerated within each reporting region, meaning that each STE ID is comprised of $log_2 1024 = 10$ bits, and the width of the encoder outputs is $32 \times 10 = 320$ bits.

The output buffer must therefore have a 320-bit port for reporting and a 512-bit port for DMA to DRAM. However, the dual RAM design is restricted by the set of ratios between port A and port B widths are 1, 2, 4, 8, 16, and 32. This prevents generating RAM with ratio 512/320, leading us to necessitate padding the input port width by extra 0s to the left to achieve the minimum valid ratio between the two ports. These padded bits are used to store the input offset as shown in Figure 11.

4.4.4 Priority Encoder. The priority encoders identify the active reporting STEs in each cycle. In each cycle, the encoding process starts by the right-most bit in the group, checking if the bit is set. If so, the bit will be encoded, and its ID sent to the reporting-ID register. If the bit is zero, the priority encoder moves to the next bit and repeats the process, until the final bit in the group.

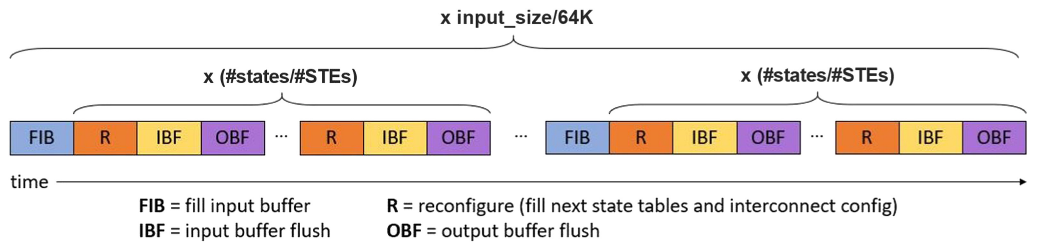


Fig. 10. NAPOLY timing diagram.

In order to limit the logic latency of this operation, the priority encoders in each reporting region are further divided into one or more *output regions*, within which it limits the number of reporting states. Figure 11 shows the design of the priority encoder with a reporting region size of 16 and a output region size of 8.

4.4.5 NAPOLY Performance Model. Historically, automata DSAs have evaluated performance in terms of symbol throughput, i.e., symbols per cycle or symbols per second. Practical workloads often require multiple passes through the pattern set, in which case reconfiguration time plays a substantial part in end-to-end performance. Performance metrics must therefore consider the number of reconfigurations and the time to read input set and flush the reports.

NAPOLY is reprogrammed in three steps. The current state tables, which are mapped onto MLAB blocks, are written through an exposed write port, while the registers (interconnect and state flags) are programmed using parallel shift register chains. There is one chain for each bit of width to the external memory interface. For the Stratix 5 board, there are 64 shift registers to allow for 64 bits to be shifted in every cycle to match the DRAM interface width. This width is scalable to utilize all available memory bandwidth. Finally, the input symbol buffer, which is mapped onto M20K blocks, is also configured through an exposed write port.

At runtime, NAPOLY follows the timing diagram shown in Figure 10. 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 STEs it must reconfigure its array ($time_{reconfig}$) DRAM (loading next_state tables and configuring gates), flush the input buffer through the array ($time_{IBF}$), and flush the output to DRAM ($time_{OBF}$).

Reconfiguring the interconnect bits scales with f and the number of STEs (N) and the time to load the current state tables scales with the number of STEs (note that f and N vary inversely).

The effective throughput is calculated according to Equation (1). The reconfiguration time $time_{reconfig}$ gives the time needed to reconfigure a new NFA onto the overlay. Thus, the execution time scales with $R \times time_{reconfig} \times \frac{IS}{64K}$, where $64KB$ = the size of the input buffer and IS is the size of the input data to be searched for patterns.

$$\text{Throughput} = \frac{size_{input\_buffer}}{\frac{size_{input\_buffer}}{bw_{DRAM}} + R \times (time_{reconfig} + time_{OBF} + time_{IBF})} \qquad (1)$$

5 MAPPING PROBLEM

Mapping an NFA graph to an overlay (NAPOLY) is performed by allocating each state into an STE and consequently mapping every edge to an STE-to-STE wire. This mapping must be performed without violating the hardware fan-out constraints; that is, without mapping any pair of connected states to a pair of STEs whose physical distance exceeds the reach of the STE interconnects [16].

Definition 5.1. For a given NFA $\{V, E\}$, where V is a set of states and E a set of edges (transitions), a **map** is an association between each of the NFA states of an NFA graph and a corresponding STE

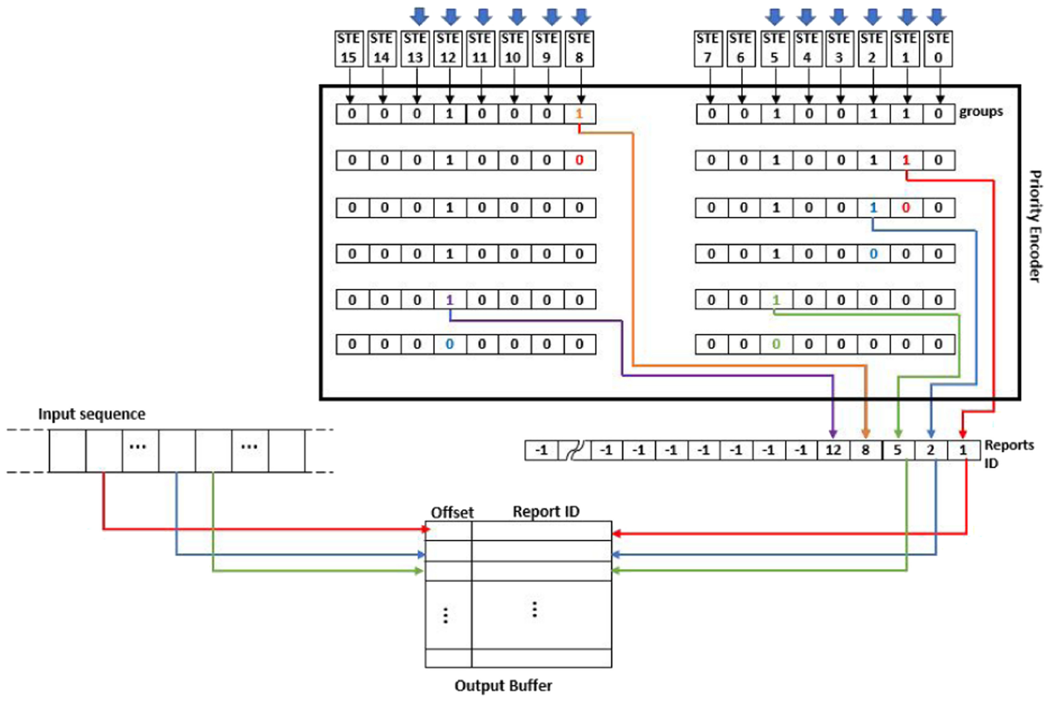


Fig. 11. NAPOLY output region.

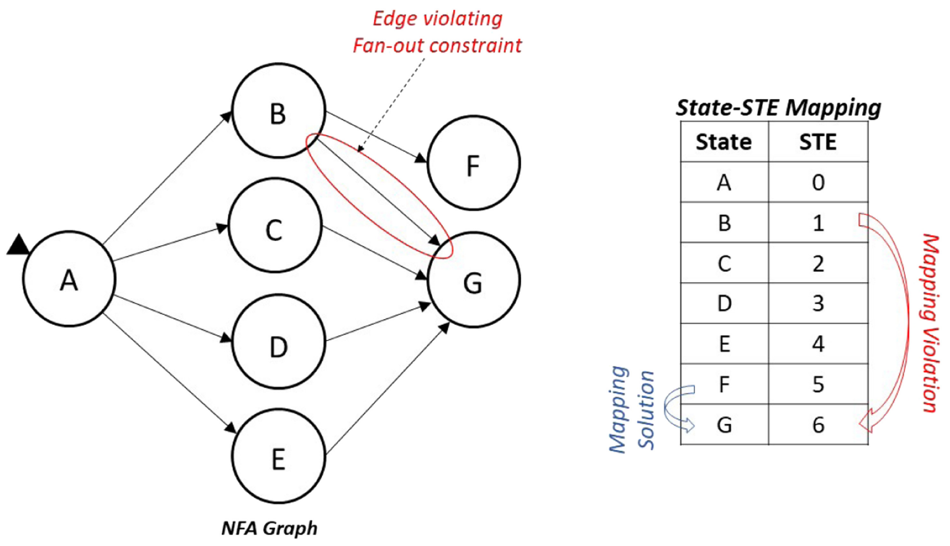


Fig. 12. Mapping problem.

index in the range of $[0, N-1]$, where N = number of STEs. There are thus $|V|!$ unique maps for a given NFA assuming $|V| = N$.

For example, assume we have an NFA graph consisting of seven states $[A, B, C, D, E, F, G]$ as shown in Figure 12, and we need to map this NFA onto an overlay consisting of seven STEs

$[0, 1, 2, 3, 4, 5, 6]$. Assume the hardware fan-out f is 9, meaning that the maximum forward connection distance is 4, maximum backward connection distance is 4, and self-loop is one connection.

If we map each state to an STE in order, as it is shown in Figure 12, the edge between B and G will require a connection to mapping state B onto STE_1, and mapping state G to STE_6. This will violate the fan-out constraint which is maximum backward or forward distance of 4. We refer to this as a *mapping violation*. One way to meet this constraint is to map state F to STE6, and G to STE5, as shown in the Figure.

In this example, there is 7! or $5,040$ possible ways to map the NFA onto the overlay. With $f = 5$, there is no mapping solution. With $f = 6, f = 7$, and $f = 8$, the number of mapping solutions grows to $24, 48$, and 372, respectively.

5.1 Greedy Mapping Heuristic

For each violation between a *predecessor* and *successor* state, the mapping score resulting from all possible violation resolutions is calculated and compared to the previous mapping score. The mapping score is the sum of mapped distances between the STEs holding each pair of connected states. To re-map a state from its original STE to a target STE, all states mapped to STEs between the original STE and target STE are shifted to accommodate the state's change in location.

The greedy heuristic considers every possible way to re-map either the predecessor or successor state that results in the physical distance between the predecessor and successor being less than or equal to the maximum reach as defined by hardware fan-out. Note that resolving a violation may create new violations caused by the shifting of states required in the re-mapping. The heuristic effectuates the re-mapping that gives the best overall score improvement relative to the original mapping state. If none of the re-mapping options improves the mapping score, then the algorithm moves on to the next violation without re-mapping any states.

The mapping heuristic will abort execution if it fails to achieve a reduction in mapping score after several iterations. When this occurs, the algorithm is performed again but using an overlay configuration with greater reach and less STEs.

5.2 SAT Solver Mapping Algorithm

Mapping states to STEs against the hardware interconnect constraints is reducible to SAT. In this work, we use CryptoMiniSat [29] as our SAT solver.

The hardware fan-out parameter f defines which subset of maps is valid for a given NFA. In order to find a valid map, a mapping algorithm must assign $map(s) \forall s \in V$ subject to the following constraints:

(1) **Maximum hardware fan-out**,
$\forall (s, d) \in E\text{: } ((map(s) - map(d)) \leq \lfloor (\frac{f-1}{2}) \rfloor) \text{ and } ((map(d) - map(s)) < \lfloor (\frac{f}{2}) \rfloor)$
(2) **Every state must be assigned to only one STE**
$\forall s \in V, \forall i, j \in N, i \neq j, \text{ if } map(s) = i, \text{ then } map(s) \neq j$
(3) **Every STE must be allocated one state**
$\forall s, d \in V, s \neq d, \forall i \in N, \text{ if } i = map(s), \text{ then } i \neq map(d)$
(4) **All states must be allocated**
$\forall s \in V, map(s) \in N$

In order to allocate the states into STEs, we describe the constraints above in **conjunctive normal form** (**CNF**), where each clause is formed as a disjunction of literals (i.e., a product of sums). We assign each possible mapping of a state to an STE as a Boolean variable whose state determines if the mapping is made, i.e., Let $L_i^s = TRUE$ when $map(s) = i$.

We describe *constraint 1* as shown in Equation (2) by constructing a set of clauses that collectively guard against every possible mapping violation.

$$\bigwedge_{\forall(s,d)\in E, \forall i\in N} (\overline{L_i^s} \vee \bigvee_{\forall m\in[-\lfloor\frac{f-1}{2}\rfloor\ldots,-1,1,\lfloor\frac{f}{2}\rfloor]} L_{i+m}^d). \tag{2}$$

In other words, if any edge (s, d) is mapped such that $map(s) == i$ and state d is not mapped to SEs $i - \lfloor\frac{f-1}{2}\rfloor$ to $i + \lfloor\frac{f}{2}\rfloor$, then the clause will be FALSE, invalidating the entire CNF expression.

We describe *constraint 2* similar to the previous constraint, but for each conjunction as the complemented variables corresponding to each state mapped to every pair of STEs, as shown in Equation (3).

$$\bigwedge_{\forall i_1\in N} \bigwedge_{\forall i_2\in N} \bigwedge_{\forall s\in V} (\overline{L_{i_1}^s} \vee \overline{L_{i_2}^s}). \tag{3}$$

We describe *constraint 3* by adding $|V|^2 \times |N|$ additional clauses, formed from the conjunction of the complemented variables corresponding to every pair of states mapped to every STE, as shown in Equation (4).

$$\bigwedge_{\forall s_1\in V} \bigwedge_{\forall s_2\in V} \bigwedge_{\forall i\in N} (\overline{L_i^{s_1}} \vee \overline{L_i^{s_2}}). \tag{4}$$

We describe *constraint 4* by adding an additional clause for each state, comprised of the conjunction of the literals representing every possible mapping of that state, as shown in Equation (5).

$$\bigwedge_{\forall s\in V} \bigvee_{forall i\in N} L_i^s. \tag{5}$$

Figure 13 depicts an example NFA, overlay, and corresponding CNF clauses that describe constraint 1. Graph G is composed of $V \in 0, 1, 2, 3$, $E \in (0, 1), (0, 2), (1, 3), (2, 3)$, and overlay M is composed of $N \in (0, 1, 2, 3)$ and $f = 3$.

Each potential mapping clause is shown as a matrix in Figure 13 where its rows represent the state end the columns represent the STEs to which the state can potentially be mapped. The cells in the matrix are the literals of the clauses, shown as T as the positive literal and F as the negative literal. The clause joins literals by OR, while clauses are joined by AND.

The $C(E_{01})$ in Figure 13 constrains the edge between state 0 and state 1 and shows four logical implications converted into four logical disjunctions. Any of these would evaluate to false if state 0 were mapped to any of the STEs without state 1 being mapped to another STE within the range $[-1, 2]$. In CNF, all clauses must be true to satisfy the expression.

5.3 NFA Transformation

NAPOLY overlay configurations exhibit a tradeoff between the number of STEs and the hardware fan-out, the number of available inputs and outputs in each STE. This tradeoff is caused by the resource constraints imposed by the ALMs required by the OR-gate that combines the inputs from all the possible predecessors into each STE.

As shown in Figure 19, larger overlays achieve higher throughput because they require less total runtime reconfigurations, but the maximum overlay size available to a given automata is determined by the minimum hardware fan-out on which the automata can be successfully mapped by the NAPOLY compiler. To maximize performance, each automata must be mapped onto an overlay having minimal hardware fan-out to allow for the use of a larger overlay. The minimum hardware fan-out depends on the transition density of the automata.

Our methodology for finding the minimal hardware fan-out for a given NFA is to perform a binary search. For some benchmarks, it is possible to map overlays with lower hardware fan-out

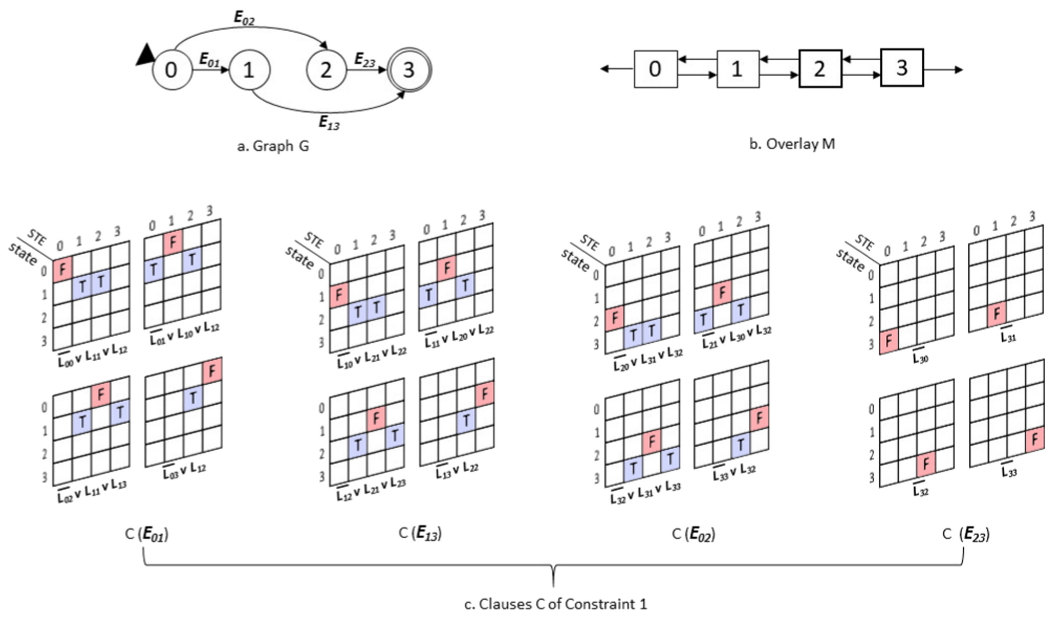


Fig. 13. An example for generating CNF clauses of literals based on *Constraint 1*.

by transforming the NFA into a functionally equivalent alternative form that limits the maximum number of incoming and/or outgoing transitions from each state at the cost of an increased number of states.

We use the fan-in/fan-out relaxation technique included in VASim [34] to decompose any states that have an in- or out-degree larger than the prescribed fan-in or fan-out limit. This type of transformation replicates all the states along all the paths from the start states to the accepting states that are part of any of the high fan-in or fan-out paths, as shown in the example in Figure 14. This approach is only practical when the performance gained from increasing the overlay size outweighs the performance loss caused by increasing the number of states and the resulting number of reconfigurations.

To explore this, Figures 15–17 show the achieved throughput of the Promomata, Snort, and PowerEn benchmarks on two different overlays: the baseline one, in which the compiler can successfully map all the states for the unaltered version of the benchmark, and the performance achieved on the next higher size overlay under the assumption that a given number of states must be replicated in order to reduce the transition density to the point where the compiler can map the automata (shown on the horizontal axis). In each of these benchmarks, less than 5% of its NFA sub-graphs failed to map to one of the overlay configurations, which in this case is considered to be the “next size up”. These results reveal the maximum number of state replications permissible before the overhead in workload outweighs the benefit of the larger overlay.

In the case of Protomata, the compiler was able to successfully map the automata to the 16K overlay (after having previously only been mappable to the 8K overlay) after the density improvements achieved through a 2% increase in states, giving way to an end-to-end speedup of 1.51, as shown in Figure 15.

In the case of Snort, the compiler was able to successfully map the automata to the 16K overlay (after having previously only been mappable to the 8K overlay) after the density improvements achieved through a 4% increase in states, giving way to an end-to-end speedup of 1.61, as shown in Figure 16.

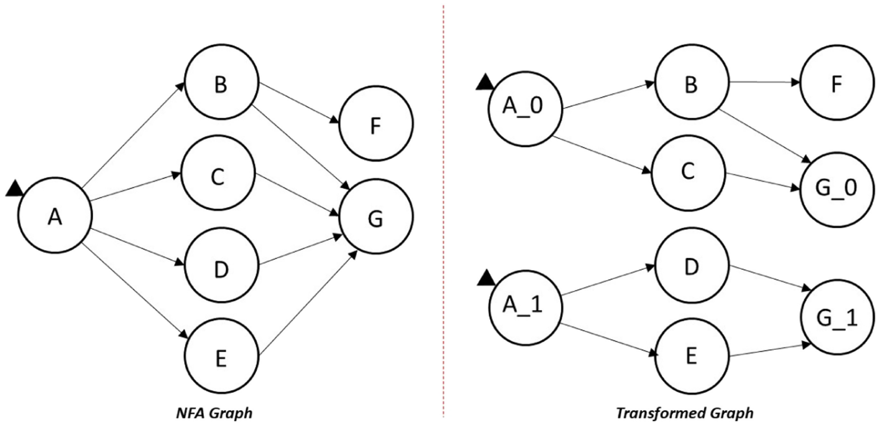


Fig. 14. Transformation of NFA graph in Figure 12.

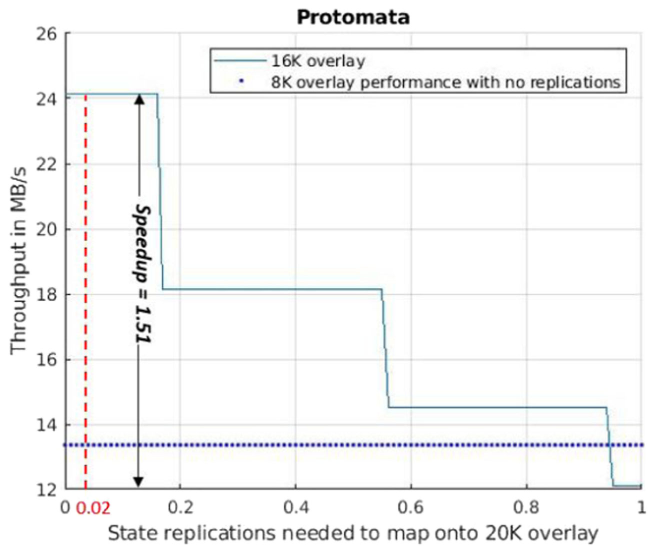


Fig. 15. Protomata: Performance on 16K STE overlay and 20K STE overlay vs. the assumed number of state replications needed to map the automata onto the 20K STE overlay with its fewer interconnections.

In the case of Power En, the compiler was able to successfully map the automata to the 20K overlay (after having previously only been mappable to the 16K overlay) after the density improvements achieved through a 0.3% increase in states, giving way to an end-to-end speedup of 1.29, as shown in Figure 17. As shown, Power En performance speeds up only within a very small region (number of replications ≤ 0.008) of 20K overlay performance plot.

5.4 Experimental Analysis

Comparing with the heuristic described in Section 5.1, the SAT solver-based mapper can map 75% of the ANMLZoo benchmarks to larger overlay configurations than when using the heuristic, which results in fewer number of reconfigurations at runtime. Figure 20 shows the effective speed up for these benchmarks when using the SAT-based mapper as compared to the heuristic mapper.

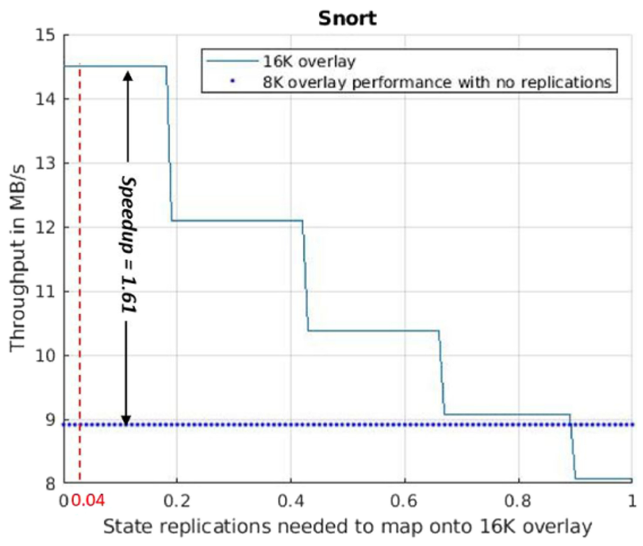


Fig. 16. Snort: Performance on 8K STE overlay and 16K STE overlay vs. the assumed number of state replications needed to map the automata onto the 16K STE overlay with its fewer interconnections.

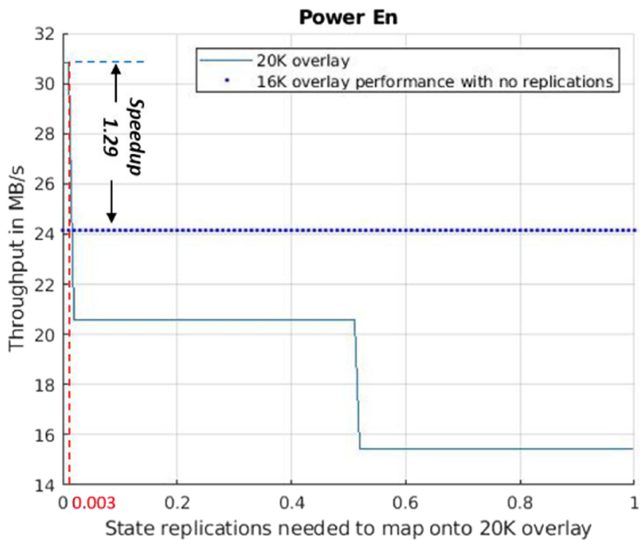


Fig. 17. Power En: Performance on 16K STE overlay and 20K STE overlay vs. the assumed number of state replications needed to map the automata onto the 20K STE overlay with its fewer interconnections.

5.4.1 Hardware Resources. Table 4 shows the hardware resources required to implement six overlay configurations. The column labeled **#STEs** gives overlay sizes (number of STEs), the column labeled **Maximum hardware fan-out** shows **f** the hardware fan-out of the overlay, and the column labeled **Fmax** shows the maximum clock frequency. The remaining columns show the hardware resources. Note that each of these overlay configurations is limited by ALM usage, which is driven by the overlay's STE fan-out.

Table 5 shows the total M20K used to implement the output buffer in the overlays. The column labeled **Buffer depth** ranges between $[64K, 32K, 24K, 16K, 12K, 8K]$ based on overlay size

Table 4. Hardware Resources Used in Different Overlay Configurations

# STEs	Max Hw. Fan-out (f)	Fmax (MHz)	MLABs	ALMs%	Reg.%	M20K%
4K	103	152	1,047,296	90	46	41
8K	44	136	2,096,384	91	41	41
12K	25	122	3,145,472	95	36	60
16K	12	121	4,193,024	94	26	41
20K	6	119	5,242,112	95	19	61
24K	3	112	6,291,200	96	15	41

Table 5. Total M20K Used for Output Buffer

# STEs	Buffer Depth	Buffer Width	Buffer Width After Padding	Total M20K (MB)
4K	64	192	256	16
8K	32	416	512	16
12K	24	624	1,024	24
16K	16	896	1,024	16
20K	12	1,144	2,048	24
24K	8	1,399	2,048	16

(**#STEs**). The column **Buffer Width** shows the width of the output buffer, which is determined by #encoders $\times$ #outputregions $\times \log_2$(STEs). As described in Section 4.4, the buffer width needs to be padded, as shown in column **Buffer Width after Padding**. Column **Total M20K** shows the total number of M20K needed in each overlay.

5.5 NAPOLY Run Time

Table 6 shows the Pareto optimal set of synthesized and place-and-routed overlay configurations with respect to STE capacity and hardware fanout. The column **#STEs** lists all the six NAPOLY configurations. The column labeled **Max BW for** $N_{\%active} = 0.25(GB/s)$ gives the effective on-chip memory bandwidth needed for 25% average active states. Exploitation of on-chip memory bandwidth is the principle performance advantage of NAPOLY over CPU- and GPU-based approaches.

The column labeled **Time_Reconfig (T)** lists the time needed to reconfigure a new NFA onto the overlay. The column labeled **Output Encoders** gives the number of output encoders, which determines the maximum number of “reports”, or accepting state activation, allowed per clock cycle.

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)**.

For a given NFA and input, the effective throughput is calculated according to Equation (1), which is shown in the last two columns (**Throughput for 24K** and **Throughput for 128K**) at 24K states and 128K states, respectively.

Figure 18 shows NAPOLY execution time is dominated by the time to flush input buffer and the time to flush the output buffer.

Figure 19 plots the achieved throughput of all NAPOLY overlays for 1 million input characters and for a total NFA workload from 4K to 128K states. The performance difference between the different overlays converges to their size, i.e., the 24K overlay is 6X faster than the 4K overlay for an automata of 128K states).

Table 6. Repertoire of the Achieved NAPOLY Configurations

# STEs	Max BW (GB/s)	Time Reconfig T (μs)	Output Encoders	Max Report Cycles	Max Report Rate (GHz)	Throughput 24K states (MB/s)	Throughput 128K states (MB/s)
4K	1,866	21	16	100%	2.4	14	3
8K	1,427	31	32	50%	2.2	27	5
12K	1,031	43	48	33%	2.0	32	6
16K	692	53	64	25%	1.9	36	9
20K	426	67	80	20%	1.9	31	9
24K	240	74	96	17%	1.8	67	11

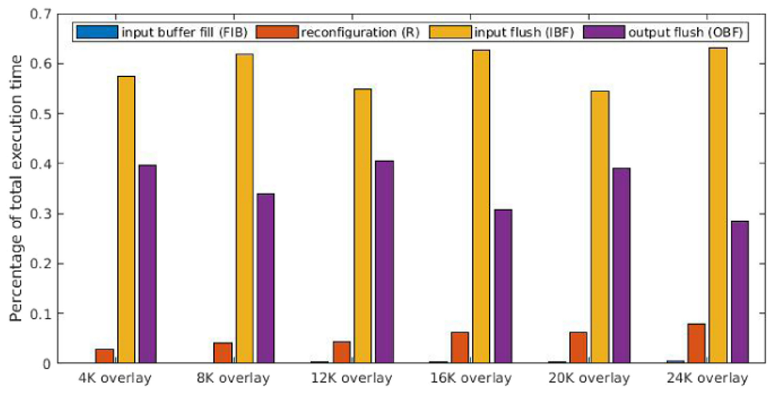


Fig. 18. Execution time makeup of NAPOLY.

5.6 Mapping Results

Table 7 shows the mapping result for each of the ANMLZoo benchmarks using SAT solver which achieved a significant improvement in hardware fan-out, targeting larger overlay and reducing the number of re-configurations in 75% of ANMLZoo benchmarks.

5.6.1 NFA Transformation Results. We applied the NFA transformation technique described in Section 5.3 on Protomata, Snort, and Power En benchmarks.

Tables 8, 9, and 10 show the state replications and the achieved hardware Fan-out after NFA transformation for three benchmarks Protomata, Snort, and Power En. The first column represents the **Fan-in/Fan-out limit** applied on the failing sub-graphs of each benchmark. The second column, **State Replications**, shows the number of state replications achieved when fan-in/out limits are applied. The third column shows the **Minimum Hardware Fan-out** achieved to map the sub-graphs onto larger overlays, and final column shows the **Target Overlay**. As shown in the three tables, the number of state replications significantly increases when limiting fan-in/fan-out to 1, while it lowers when moving the limits towards the maximum logical Fan-in/out for each benchmark.

5.7 Performance Comparison

For each of the ANMLZoo benchmarks, Table 11 shows the performance of competing CPU and GPU automata processing frameworks. The CPU implementation is Intel Hyperscan [1] measured

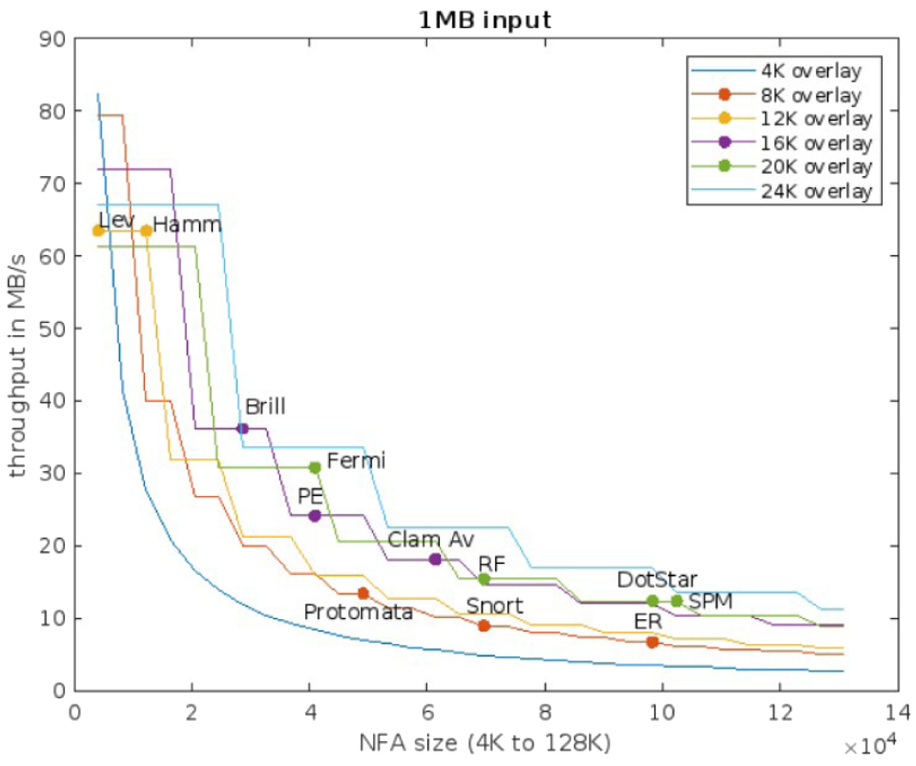


Fig. 19. NAPOLY performance vs. NFA size.

Table 7. NAPOLY Mapping Using SAT Solver

Benchmarks	# States (S)	Min f Achieved	Overlay Size (N)	# Reconf.	Throughput (MB/s)
Brill	26,668	8	16K	2	36
Clam AV	49,538	12	16K	3	16
Dot Star	96,438	4	20K	5	12
ER	95,136	41	8K	12	7
Fermi	40,783	5	20K	2	31
Hamming	11,346	14	12K	1	63
Levenshtein	2,784	16	12K	1	63
Power En	40,513	8	16K	3	25
Protomata	42,061	42	8K	6	15
Random Forest	75,340	6	20K	4	16
Snort	69,029	36	8K	9	9
SPM	100,500	6	20K	5	13

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 [9]. The second column of the table shows NAPOLY throughput for each benchmark. Comparing with Table 7, the throughput of Snort, Protomata, and Power En has increased because of applying the NFA transformation

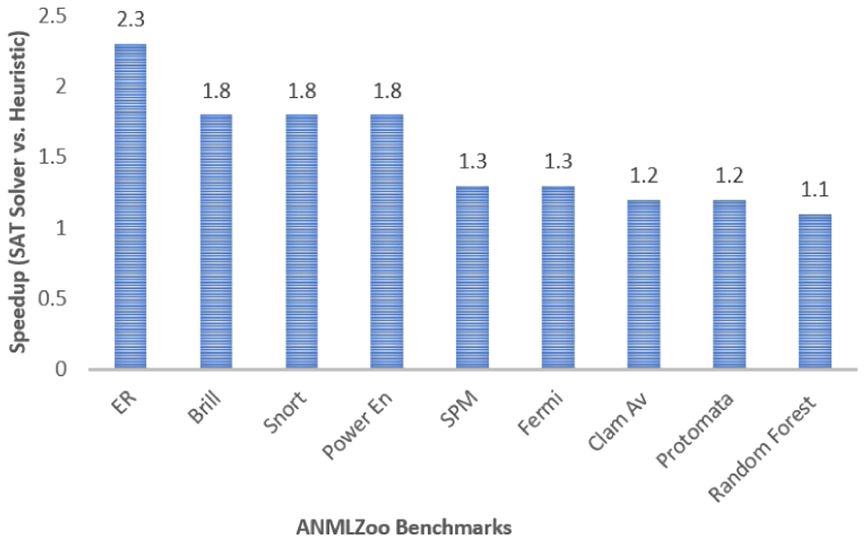


Fig. 20. Speedup achieved in 75% of benchmarks at SAT solver vs. Heuristic.

Table 8. Snort Transformation Results

Logical Fan-in/-out Limit	State Replications	Achieved HW Fan-out	Target Overlay
10/10	0	36	8K
8/8	1%	12	16K
6/6	3%	11	16K
4/4	4%	11	16K
2/2	4%	9	16K
1/1	40%	2	24K

Table 9. Protomata Transformation Results

Logical Fan-in/-out Limit	State Replications	Achieved HW Fan-out	Target Overlay
24/24	0	42	8K
16/16	0.07%	11	16K
8/8	0.2%	11	16K
2/2	2%	9	16K
1/1	12,415%	2	24K

technique described in 5.3 on these benchmarks. This technique allowed us to reduce the logical Fan-in/Fan-out limit 1/1 for each of the benchmarks and mapping the benchmarks on 24K overlay. 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 [9].

Table 10. Power En Transformation Results

Logical Fan-in/-out Limit	State Replications	Achieved HW Fan-out	Target Overlay
4/3	0	8	16K
2/2	0.3%	6	20K
1/1	10%	2	24K

Table 11. Performance Results

Benchmark	NAPOLY Throughput	Average Active States (AS)	R Per B	GPU Throughput (MB/s)	CPU Throughput (MB/s)	Speedup vs. Max(CPU,GPU)
Brill	36	14	4	7	1	5
Clam AV	16	4	5	4	14	1.14
Dot Star	12	3	5	40	10	0.3
Entity Resolution	7	10	19	4	1	1.75
Fermi	31	3,854	2	2	1	15.5
Hamming	63	240	1	18	10	3.5
Levenshtein	63	88	1	38	1	1.65
Power En	31	31	5	53	10	0.58
Protomata	24	19	6	5	1	4.8
Random Forest	16	968	5	2	0.5	8
Snort	15	98	17	14	0.4	1.07
SPM	14	6,631	5	0.5	0.1	28

NAPOLY performs best for larger benchmarks with more active states and is faster than both the GPU and CPU NFA implementations in 10 of the 12 benchmarks, while the GPU implementation is faster in only two benchmarks. DotStar and PowerEn have a small number of reports (0 for Dot Star and 4,304 for Power En [32]) and a relatively small number of active states (0.003% for Dot Star and 0.07% for Power En). C

$$\frac{1}{\frac{1}{2}+\frac{1}{2}\times\frac{1}{2}} \approx 1.33. \tag{6}$$

5.8 Overlay Scalability

As shown in Equation (1), 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 12 shows NAPOLY capability when scaled up to an Intel Stratix 10 GS. However, even if a larger FPGA can offer roughly double of overlay capacity, double of clock rate and double of DRAM bandwidth, the performance won't probably be doubled according to Equation (6).

6 CONCLUSION

In this article, we have presented a novel architecture for an automata processor overlay and its associated software. NAPOLY is parameterizable, allowing for tradeoffs in state capacity, interconnect density, and output buffer size. These tradeoffs allow for offline generation of a repertoire of

Table 12. Repertoire of Achieved Configurations on Stratix 10GS

#STEs	Hardware Fanout	Output Encoders	Max Reporting Cycles	Max Report rate (GHz)	Fmax (MHz)	Max BW for 25 %active (GB/s)
4K	254	16	100%	4.64	290	8,746
8K	126	32	50%	8	250	7,510
12K	83	48	33%	12	250	7,331
16K	62	64	25%	13.4	210	6,208
20K	49	80	20%	15.2	190	5,549
24K	40	96	17%	16.32	170	4,863
28K	34	112	14%	16.8	150	4,255
32K	30	128	12%	16.64	130	3,719
36K	26	144	11%	15.84	110	3,068
40K	23	160	10%	14.4	90	2,467
44K	21	176	9%	12.32	70	1,744
48K	19	192	8%	9.6	50	1,072

Table 13. Wire Utilization Achieved For ANMLZoo Benchmarks

Benchmark	Max Logical Fan-in/ Fan-out	Min Hardware Fan-in/ Fan-out	Average Fan-in degree	Average Fan-out degree	Fan-in Wire Utiliz	Fan-out Wire Utiliz
Brill	4/4	8/8	1.11	0.72	13.8%	9%
ClamAV	11/2	18/18	1.01	1.003	5.6%	5.6%
DotStar	2/2	4/4	1.00	0.48	25%	12%
Entity Resolution	28/5	41/41	1.89	1.15	4.6%	2.8%
Fermi	2/2	5/5	1.33	1.41	26.6%	28.2%
Hamming	4/2	14/14	1.69	1.69	12%	12%
Levenshtein	8/5	16/16	2.89	1.63	18%	10.2%
PowerEn	4/3	6/6	1.08	0.51	18%	8.5%
Protomata	3/106	9/9	1.02	0.49	11.3%	5.4%
Random Forest	2/2	6/6	1.05	0.5	17.5%	8.3%
Snort	19/19	9/9	1.22	0.6	13.5%	6.6%
SPM	3/2	6/6	2.1	1.05	35%	17.5%

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 arbitrary large NFAs. Automata-based benchmarks are mapped to NAPOLY processing elements based on the results of an SAT solver.

Our performance results included the time required to program the overlay from DRAM and are competitive with the state-of-the-art CPU- GPU-based implementations. Our performance results showed that NAPOLY's performance scales with on-chip memory capacity.

NAPOLY's main limitation is the hardware fan-out constraint, which determines the number of neighboring STEs to which any STE can connect and determines the maximum distance (or "reach") when establishing edges (NFA transitions) between mapped STEs. The fan-out constraint is imposed by the FPGA resources and must be traded off against STE capacity. As shown in Table 13, the utilization of STE-to-STE connections for each benchmark is less than 29%, meaning that a more efficient interconnect design would allow for more STEs, less reconfigurations for a given NFA, and higher throughput.

Additionally, NAPOLY spends over half of its execution time flushing the output buffer to DRAM, during which the STE array is idle. It is possible to perform these steps in parallel, but overlapping STE execution and output flushing would require splitting the STE array into two halves, resulting in more reconfigurations.

REFERENCES

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

Q3
[2] Michela Becchi and Crowley Patrick. 2009. *Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation.* Washington University, St. Louis, MO.

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

[4] A. Putnam et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. , In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture.*

[5] A. Subramaniyan et al. 2017. Cache automaton. In *Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture.*

[6] D. Guo et al. 2008. A scalable multithreaded l7-filter design for multi-core servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems.*

[7] E. Sadredini et al. 2020. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture.*

[8] J. Hauswald et al. 2015. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating System.*

[9] J. Wadden et al. 2016. ANMLzoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization.*

[10] J. Yu et al. 2019. Time-division multiplexing automata processor. In *Proceedings of the Design, Automatation and Test in Europe 2019 IEEE.*

[11] K. Peng et al. 2011. Chain-based DFA deflation for fast and scalable regular expression matching using TCAM. In *Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems.*

[12] K. Wang et al. 2015. Association rule mining with the micron automata processor. In *Proceedings of the IEEE 29th International Parallel and Distributed Processing Symposium.*

[13] M. Casias et al. 2019. Debugging support for pattern-matching languages and accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems.*

[14] N. Cascarano et al. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010).

[15] R. Karakchi et al. 2017. A dynamically reconfigurable automata processor overlay. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'17).*

[16] R. Karakchi et al. 2019. An overlay architecture for pattern matching. In *Proceedings of the IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP'19).*

[17] R. Moussalli et al. 2014. A study on parallelizing XML path filtering using accelerators. *ACM Transactions on Embedded Computing Systems* 13, 4 (2014), 1–28.

[18] R. Nishtala et al. 2013. Scaling memcache at facebook. .

[19] R. Rahimi et al. 2020. Grapefruit: An open-source, full-stack, and customizable automata processing on FPGAs. In *Proceedings of the IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'20).*

[20] Y. Fang et al. 2015. Fast support for unstructured data processing: The unified automata processor. In *Proceedings of the MICRO-48.*

[21] Y. Huang et al. 2022. CAMA: Energy and memory efficient automata processing in content-addressable memories. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'22).*

[22] Y. Yang et al. 2008. 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.*

Q4
[23] G. Li F. Seide and D. Yu. 2011. Conversational speech transcription using context-dependent deep neural networks. In *Proceedings of the 12th Annual Conference of the International Speech Communication Association.*

[24] P. Flicek and E. Birney. 2009. Sense from sequence reads: Methods for alignment and assembly. *Nature Methods* (2009).

[25] A. Harris. 2010. Distributed caching via memcached. *Pro ASP NET 4 CMS.*

[26] Peter Linz. 2006. *An Introduction to Formal Languages and Automata.* Jones and Bartlett Learning.

[27] Jason D. Bakos Rasha Karakchi. 2019. nfatool. Retrieved from https://github.com/HeRCLab/nfatool. Q5

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

[29] Mate Soos, Karsten Nohl, and Claude Castelluccia. 2009. Extending SAT solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing.* 244–257. DOI: https://doi.org/10.1007/978-3-642-02777-2_24

[30] Louis Woods Teubner, Jens, and Chongling Nie. 2012. Skeleton automata for FPGAs: Reconfiguring without reconstructing. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM.*

[31] T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins. 2015. Nondeterministic finite automata in hardware – the case of the Levenshtein automaton. In *Proceedings of the International Workshop on Architectures and Systems for Big Data (ASBD) in Conjunction with the 42nd International Symposium on Computer Architecture (ISCA'15).*

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

[33] J. Wadden, K. Samira Khan, and K. Skadron. 2017. 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.*

[34] J. Wadden and K. Shadron. 2016. *VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research.* Technical Report CS2016-03, University of Virginia.

[35] Xuan Wang, Lei Gong, Jing Cao, and Wenqi Lou. 2023. hAP: A spatial-von Neumann heterogeneous automata processor with optimized resource and IO overhead on FPGA. In *Proceedings of the 31st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'23).*

[36] Y. Yang and V. Prasanna. 2012. High-performance and compact architecture for regular expression matching on FPGA. *IEEE Transactions on Computers* 61, 7 (2012). Q6

Received 17 August 2022; revised 20 February 2023; accepted 27 March 2023

AUTHOR QUERIES

Q1: AU: Figure 4 is not cited in the text, please check.
Q2: AU: Please be sure to mention Figure 11 after mention of Figure 10 per ACM style.
Q3: AU: Please provide the issue number and volume number in references [1], [24], and [25].
Q4: AU: Please provide the names of all the authors in references [1] and [4]–[22].
Q5: AU: Please provide access date in reference [27].
Q6: AU: Please provide the page range in references [1], [3], [14], [24], [25], and [36].