A Dynamically Reconfigurable Automata Processor Overlay

Rasha Karakchi
Dept. of Computer Science and Engineering
Univ. of South Carolina
Columbia, SC USA
karakchi@email.sc.edu

Lothrop O. Richards III
Dept. of Computer Science and Engineering
Univ. of South Carolina
Columbia, SC USA
lothropr@email.sc.edu

Jason D. Bakos
Dept. of Computer Science and Engineering
Univ. of South Carolina
Columbia, SC USA
jbakos@cse.sc.edu

***Abstract*—This paper describes a design for a parameterizable automata processor overlay and a placement algorithm required for its support software. The resulting framework serves as both an open-source alternative to Micron's Automata Processor (AP) and as an experimental testbed for exploration of architectural tradeoffs. An automata processor is a processor-in-memory architecture designed to recognize patterns in streaming data. Our framework takes a description of a nondeterministic finite automata (NFA) described in Micron's ANML language and uses instantiated JTAG sources to configure the on-chip RAM and programmable interconnect of the overlay programmed onto an FPGA. Like the Micron AP, our design is comprised of an array of interconnected state transition elements (STEs). While our STE design is equivalent to that of the Micron AP, our overlay uses a simpler, non-switched interconnect based on pairwise gated connections. This interconnect design creates a constraint satisfaction problem when mapping logical states to the physical STEs. In this paper, we explore the impact of tradeoffs in the interconnect architecture as it relates to a Stratix 5 GX target device and we describe and evaluate an algorithm for STE placement with respect to the ANMLZoo benchmark suite. As far as the authors know, this is the first example of an FPGA-based automata processor overlay.**

Keywords—automata processor, Micron AP, NFA, processor-in-memory, reconfigurable computing, pattern matching, ANML, FPGA, heterogeneous computing, accelerator, high-performance computing, big data, data analytics

I. Introduction

Many tasks in big data analytics are built upon pattern matching and classification operations. Examples include approximate string matching, calculating the distance between two genomic sequences, signature-based threat detection, feature extraction, and association rule mining. Such pattern matching tasks are reducible to instances of deterministic finite automata (DFA) or nondeterministic finite automata (NFA). When evaluated on a CPU, the computational throughput of DFA and NFA are generally limited by cache performance, which (for large pattern sets) is itself limited by the inherently unpredictable memory access pattern of state transition tables. Achieving transformative performance improvements for these types of tasks require specialized processors having customized memory structures, such as the Micron Automata Processor (AP) [1].

The Micron AP is a programmable, spatial architecture for implementing arbitrary NFAs. The current generation Micron AP stores state activation in a 48 kilobit register and stores the state transition table in two separate memory structures: an array of 48K 256x1 asynchronous RAMs that store the input symbols and a programmable interconnect that stores the logical topology of the NFA's transitions. During operation, the interconnect serves as a function that maps the output of the RAMs to the input of the state registers.

The AP's basic unit of configuration is the state transition element (STE), which contains a 256x1 RAM and one state register. Like an FPGA, the AP's programmable interconnect is hierarchical, in which localized groups of STEs are more density-connected than more distant STEs. Since the interconnect requires a significant amount of real estate, there is fundamental tradeoff between the number of STEs and the interconnect density. For the Micron AP, this tradeoff is fixed, while an FPGA-based automata processor can offer a repertoire of design space alternatives while still offering the ability to re-use any one design for multiple NFA descriptions.

In this paper, we describe and evaluate a reusable FPGA-based overlay architecture that exhibits most of the functionality of the Micron AP and can rapidly adapt its behavior to match that of an arbitrary NFA description without requiring a full FPGA synthesis or reconfiguration. Unlike previous work that requires the FPGA design be re-synthesized for each NFA, our design is an abstraction that is synthesized once but is reconfigurable in-place on the FPGA. We target an Intel Stratix 5 GX A7 FPGA, which contains 7 Mb of asynchronous (LUT-based) RAM and 938,880 registers. This would potentially allow for the design of an AP array of up to 28,672 STEs or approximately one-half the capacity of a Micron AP. In this paper, we demonstrate experimental results for deployed designs of up to 24,576 STEs, roughly 86% of the theoretical capacity bound.

II. Background

Deterministic Finite Automata (DFA) are commonly used to implement regular expressions and to design sequential digital logic. DFA are comprised of a set of states connected by labeled-edges. During operation, a DFA may have one active state at any time and accesses only one entry of its state transition table and thus must contain a state for every possible partial

978-1-5386-3797-5/17/$31.00 ©2017 IEEE

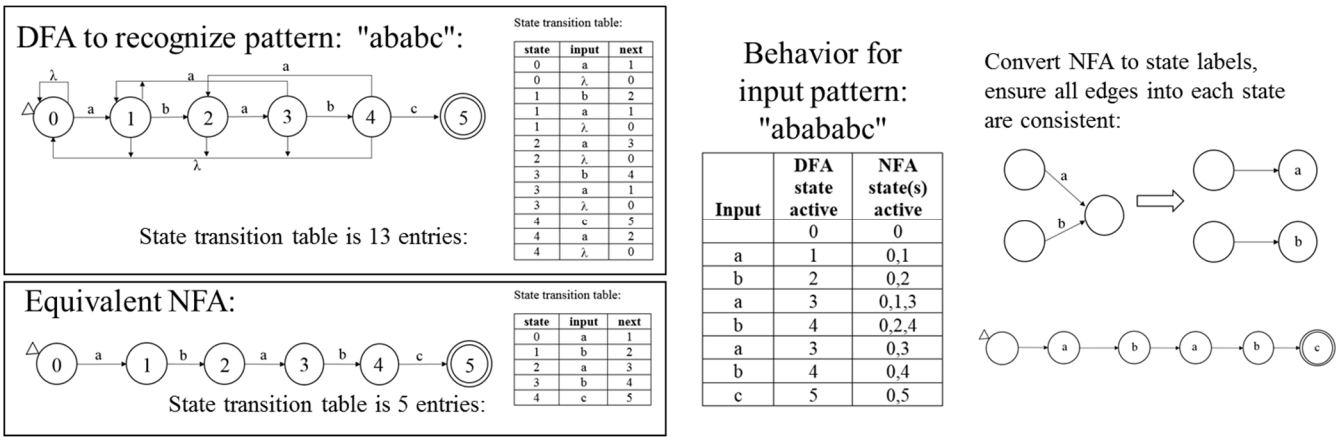

Fig. 1. DFA vs. NFA and conversion of NFA with edge labels to state labels.

match of every possible pattern. This can lead to combinatorial growth of the state space and next-state table.

Nondeterministic Finite Automata (NFA) differ from DFA in that multiple states may be simultaneously active, allowing for the tracking of multiple partial matches in parallel. As a result, an NFA generally requires a substantially smaller state transition table as compared to an equivalent DFA.

Fig. 1 a and b show an example DFA and NFA that recognize a simple pattern "ababc" along with their corresponding state transition tables. The next state table for the NFA is 2.6 X smaller than that of the DFA. Fig. 1c shows the activation sequence for the input pattern "abababc".

The AP's architecture inherently requires an alternative form of the NFA as shown in Fig 1d, where the transition labels are associated with the states as opposed to the edges. In this form, all transitions into each state (from all immediate predecessor states) must activate on the same set of input symbols. Using this form allows the STEs to store the symbols associated with the incoming transitions to each state.

The State Transition Element (STE) is the Automata Processor's basic element of reconfiguration. Each STE contains an activation flip-flop that holds the STE's active state equivalent to that of one-hot encoding. Each STE is interconnected with other STEs by accepting activation inputs from predecessor STEs and outputting activation signals to successor STEs. The STE's activation inputs are fanned into an OR-gate. Each STE also contains a 256-to-1 asynchronous RAM that stores a 1-bit corresponding to each of the input symbols associated with the state. The address input to this table is connected to the global, top-level input symbol. The OR-gate and the RAM work together to activate the state; internal logic sets the STE's activation flip-flop when at least one of the STE's activation inputs are asserted and there is a one-bit stored in the currently-indexed location in the RAM. Any other condition resets the flip-flop to zero.

The AP's routing matrix is built using tri-state switches that can form arbitrary connections between different pairs of STEs using a pool of shared physical wires. The switches and wires are arranged hierarchically, where the interconnectivity between STEs is highest at lower levels of the hierarchy. Starting from the bottom of the hierarchy, there are two STEs to a Group of Two (GoT), eight GoTs to a row (16 STEs), sixteen rows to a block (256 STEs), 96 blocks to a half-chip (24K STEs), and two half-chips to an AP (48K STEs). As is the case for FPGAs, the design of the routing matrix places physical constraints on the logical topology of the implemented NFA.

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

III. Previous 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 NFAs and Regular Expressions

FPGA implementation of regular expression matchers are often inspired by networking applications [2] and many of these are based on automata-based architectures. For these approaches a significant challenge is the high cost of logic synthesis and place-and-route for each set of regular expressions to be implemented.

Yang and Prasanna developed early methods for synthesizing regular expressions into logic mapped onto two specific FPGA devices, a Xilinx Virtex XCV100 (20x30 array of configurable logic blocks) and a conceptual Self-Reconfigurable Gate Array (SRGA) device [3]. Their original approach bypassed the synthesis flow and directly targeted the low-level FPGA fabric. However, as FPGA technology matured this approach became infeasible, and their second design targeted HDL but introduced additional optimization methods for both the NFA descriptions and generated architecture [4,5].

Becchi et al developed a set of techniques for optimizing both NFA and DFA-based architectures [6,7,8], 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.

B. Mapping Applications to AP Execution Model

Automata-based architectures are most commonly associated with regular expression evaluation, but the introduction of the Automata Processor has generated interest in identifying other applications that map to NFA-type architectures, or so-called "pattern recognition processors." Examples include association rule mining [9], brill tagging [10,11], and Levenshtein and Hamming distance calculation [12]. More specific examples include Protomata and Motomata [13], which search for motifs--or common approximate DNA subsequences among a group of genomes--in which each motif is identified by NFA-based pattern loaded onto the AP during runtime. For these, the performance of the AP depends on its ability to quickly synthesize and load patterns onto the AP. There are also efforts to develop general-purpose programming languages for NFA-type architectures, such as RAPID, a proposed high-level programming language for pattern recognition processors [14].

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

Wadden et al. developed a place and route tool built on VPR [15] that targets a conceptual design for a theoretical Automata Processor fabric [16]. 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 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 [17]. 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 [18].

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 [19]. 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, equally 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.

D. Optimization Methods for NFA Descriptions

Recent work has contributed new methods for transforming NFA descriptions into alterative but functionally-equivalent forms, such as eliminating redundant sub-substructures within the NFA logical topology [18].

Becchi et al developed an algorithm for partitioning NFA descriptions into roughly equal-sized sub-automata while minimizing the number of state replications and balancing the sizes of the resulting sub-automata [20]. This technique facilitates the usage of NFA descriptions on architectures that are limited by on chip memory capacity to NFAs having less transitions or states.

E. Comparative Studies of NFA Implementations on CPUs, GPUs, and FPGAs

Once configured with an NFA description, the Micron Automata Processor, the Unified Automata Processor, and all FPGA-based automata processors generally achieve high traversal throughput of one or two input symbols per clock cycle. Processing NFAs that are too large to fit on a particular device requires multiple passes of the input stream. Preprocessing time, which potentially includes synthesis and place-and-route, is often an important performance consideration. 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 [20].

IV. Overlay Architecture

Our AP overlay architecture is reusable (without re-synthesis and place-and-route) across different NFA descriptions having arbitrary state labels and arbitrary logical NFA topologies, provided that the logical topology does not violate resource constraints inherent in the overlay's structure. The most important constraint is a parameter of the interconnect that we refer to as "**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 STEs $n-4$ to $n+5$ (including to itself).

We developed several Pareto optimal versions of the overlay with varying numbers of STEs and hardware fan-out. The overlay architecture and software infrastructure is compatible with Micron's ANML NFA format, except that it currently lacks support for Boolean and counting elements, features we plan to incorporate into the next version of the design.

Our AP architecture design differs from the Micron AP in two important ways. First, it has a non-switched interconnect based on gate-able point-to-point connections between STEs. Second, the STEs contain programmable flags that can specify any STE as being a start state or a reporting state.

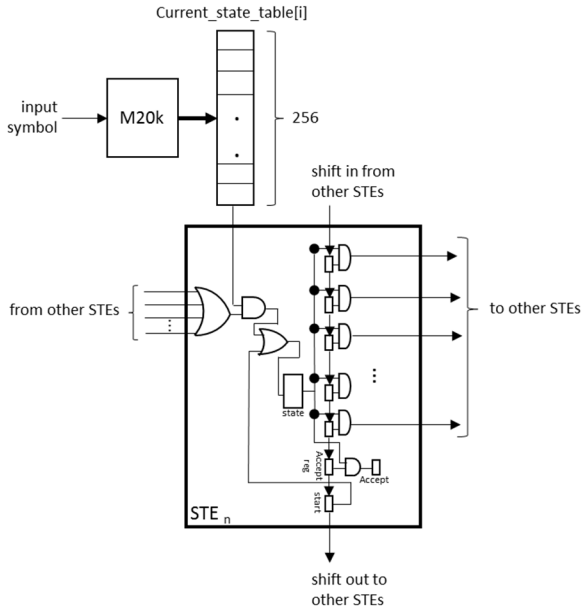

Fig. 2. STE Design.

A. STE Design

Without considering the resource usage of the interconnect, the number of STEs is limited by the on-chip RAM available to store the input symbols associated with each STE. We refer to this table as the **current state table** to remain consistent with Micron's terminology.

Our evaluation FPGA is an Intel Stratix 5 GX A7. This device has roughly 7X the on-chip memory capacity in M20K resources than it does in its MLAB (LUT-based) resources, but there are several practical problems with using M20K resources for the current-state tables.

First, 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. Second, 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. Third, the M20K requires synchronous reads, which if used for the current state table would potentially reduce the 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-flop. Finally, the M20K blocks are needed for other purposes, such as 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, as compared to 48K STEs on the Micron AP.

Fig. 2 shows the design of our STE. In order to achieve maximum utilization of the MLAB memory, the current state tables are generated as 256-deep x M bit RAMs, where M = the number of STEs in each placement-constrained region (described later). Each STE accepts a one-bit input from its corresponding column in the current state table, indexed by the input symbol. The current state MLABs are initialized through JTAG.

Each STE contains an OR-gate accepting activation signals from its connected predecessor STEs. Any cycle in which any of the incoming activation signals are asserted while 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 of its outputs, which are each AND-gated against a corresponding interconnect configuration bits before being sent out to its logical successor STEs. The interconnect configuration bits and the start and reporting flags are stored in a set of flip-flops connected in a shift register both internally and across all the STEs in the array. The shift register input is driven by JTAG through the JTAG source interface. As such, the number of available registers defines an upper bound on the level of interconnectivity. Our current FPGA contains 938,880 registers, giving an upper bound to the hardware fan-out of 32 under the theoretical limit of 29K STEs. The STE design is parameterized, allowing the synthesizer to customize it with a specified hardware fan-out.

B. Interconnect Design

The physical STEs on the FPGA are connected using point-to-point links, where each STE sends an output signal to itself and $f-1$ of its neighbors, where f = the hardware fan-out. The STEs adopt a one-dimensional addressing scheme, where each STE has an ID number assigned contiguously and sends output signals to STEs $n - \left\lfloor \frac{f-1}{2} \right\rfloor$ to $n + \left\lfloor \frac{f}{2} \right\rfloor$ where n = the STE ID.

Our proposed top-level design is different from that of the Micron AP, which uses a hierarchical switched interconnect that gives each STE the ability to send signals to a larger pool of potential successor STEs. However, a switched interconnect complicates NFA preprocessing, as the synthesis tools must place and route the states onto the fabric while managing shared interconnect resources. On the other hand, our design requires only consideration of state-to-STE mapping, since there are dedicated, non-shared wires between each pair of connectable STEs.

C. STE Mapping Algorithm

Existing tools such as VASim are capable of optimizing and transforming NFA descriptions. Examples such transformations include removing repeated NFA substructures or using state replication to restrict the maximum number of incoming or outgoing transitions into or out of a single state. These properties are often called maximum fan-in or fan-out, but we refer to them as **logical fan-in and fan-out** to differentiate properties of the NFA description from the underlying hardware.

The basic element defined in a Micron ANML file is the state transition element (STE), which shares its name with the element of reconfiguration in our overlay (and the Micron AP). In order to avoid confusion, we refer to the STEs in the ANML file as "logical STEs" and the STEs in the overlay as "physical STEs".

In other words, the logical fan-in and fan-out define the maximum number of incoming and outgoing transitions per state in the NFA topology, while the hardware fan-out, in our overlay, determines the number of physical outputs per STE in the hardware. Despite the similarity in names, there is no concrete relationship between these values, since each STE's outputs in the hardware connect to a shared set of neighboring STEs. The hardware fan-out also constrains the reach of an STE's outputs as $i - j \leq \left\lceil \frac{f-1}{2} \right\rceil$ and $j - i \leq \left\lceil \frac{f}{2} \right\rceil$ for hardware fan-out *f*, for any edge in the NFA description $s \rightarrow d$ where logical STE *s* is mapped to physical STE *i* and logical STE *d* is mapped to physical STE *j*.

As shown in Fig. 3, logical STEs in the ANML file are identified by strings but are mapped to physical STE addresses in the overlay array. In the figure, the logical STE names are shown as "first", "second", etc. while their corresponding physical STE IDs are shown in the adjoining box. Each physical STE can only activate another physical STE whose address is reachable, as determined by the hardware fan-out parameter.

Our mapping algorithm is shown as Algorithm 1 and is comprised of four routines. The **validate_edges** routine checks each edge to determine if it violates a placement constraint. The **check_move** routine evaluates the quality of a proposed solution for an edge constraint violate with respect to the set of possible alternative solutions. The **move_ste** routine modifies the placement of a given edge and adjusts other edges accordingly. Finally, the **calculate_score** routine evaluates the score of a set of edges.

Algorithm 1: Edge Validation and Re-Mapping

```
validate_edges (IN: NFA edges; OUT: NFA edges)
1    FOR each edge x→ y in the current physical STE assignment
2     IF (y-x) < ((-f-1)/2) OR (y-x)> (f/2) THEN
3      SET max_differential_score = -INT_MAX
4      FOR k FROM 0 TO f-1
5       from = x, to = y-f/2+k
6       check_move(from,to,
          max_differential_score,best_from,best_to)
7      END FOR
8      FOR k FROM 0 TO f-1
9       from = y, to = x-(f-1)/2+k
10      check_move(from,to,
          max_differential_score,best_from,best_to)
11     END FOR
12    END IF
13   END FOR

check_move (IN: from, to; INOUT:
max_differential_score, best_to, best_from)
14   score = calculate_score(from,to)
15   Move_STE (from,to)
16   SET score =score - calculate_score(from,to)
17   Move_STE (to,from)  // "undo" move
18   IF score > max_differential_score THEN
19    best_to = to, best_from = from
20   END IF

move_STE (IN: from, to; OUT: NFA edges)
21   IF from < to THEN
22    FOR each edge i →j
23     IF j == from THEN
24      replace i →j with i →to ELSE
25      IF j > from AND j <= to THEN
26       replace i →j with i →j-1
27      END IF
28     END IF
29    END FOR
30   ELSE
31    FOR each edge i →j
32     IF j == from THEN
33      replace i →j with i →to ELSE
34      IF j >= to AND j < from THEN
35       replace i →j with i →j+1
36      END IF
37     END IF
38    END FOR
39   END IF

calculate_score (IN: from, to)
40   SET sum = 0
41    FOR each edge i →j WHERE:
       (from <= i <= to OR to <= i <=from) OR
       (from <= j <= to OR to <= j <=from)
42     sum = sum + abs(i-j)
43    END FOR
44   RETURN sum
```

1) validate_edges

Our algorithm initially maps each logical STE to a physical STE according to the order in which the STE appears in the ANML file. Then, the algorithm iteratively scans each edge to detect mapping errors. The validate_edges routine performs one scan. Line 2 checks for a placement constraint violation. The loops on lines 4 to 12 evaluate each possible solution to the violation, either by changing the mapping of the source STE

(lines 4 to 7) or by changing the mapping of the destination STE (lines 8 to 12).

2) check_move

The check_move routine evaluates the effect of re-mapping the logical STE mapped to physical STE *from* to physical STE *to*. The routine first calculates the score of the subset of edges effected by the remapping (line 14), performs the remapping (line 15), recalculates the score (line 16), and then reverts the remapping to restore the original state of the graph (line 17). The routine checks the resulting difference in score and updates the "best found" remapping if the impact in score exceeds that of any previously-tested remapping (lines 18 to 19).

3) move_STE

The move_ste routine performs a remapping operation on the graph by reassigning a physical STE from location *from* to location *to*.

This operation is depicted in Fig. 3. On the left side, there is an edge connecting logical STEs "fifth" and "second", which are mapped to physical STEs n and m where $n > m$. This remapping requires that the physical mapping of all logical STEs from m to n-1 be moved down to m+1 to n to make room for the insertion of original STE n into new position m.

4) calculate_score

The calculate_score routine accumulates the distance of all incoming and outgoing edges into the given STE range. The intuition is of this score function is that remapping operations should result in an overall reduction in total edge distance.

D. I/O Interface

A 128K x 8-bit M20K-based RAM serves as the input buffer, allowing the streaming of up to 128 KB of input data into the STE array at one symbol per cycle. The runtime system configures the input buffer over JTAG using the Intel in-system memory content editor through the JTAG interface.

The user may configure any STE as a reporting state, where the user configures the reporting attribute as a flag in the 'report' flip-flop within each STE. This value is AND'ed with the active bit to drive the report output. The reporting output from each group of 1024 STEs are combined in a group. In any cycle in which any of these bits is set, the 1024-bit value, along with the value of the counter that tracks the offset in the input stream, are written into a 1024 x 1041 M20K-based RAM, allowing up to 1K of the 128K inputs to generate a report for each group of 1024 contiguous STEs.

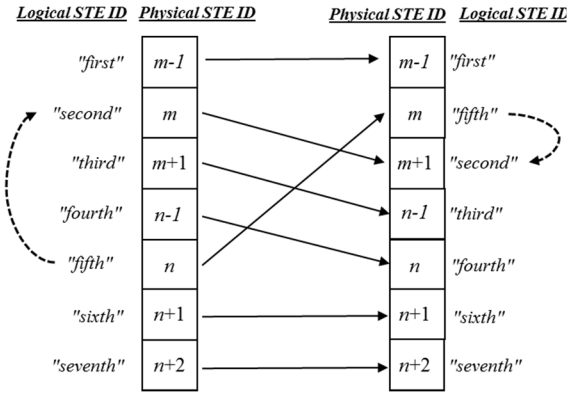

Fig. 3. Remapping physical STEs. Edge between logical STE "fifth" and "second" is reassigned from physical STEs n and m, where $n > m$, to m and m+1 (after an operation "move STE n to m"). In this case, a movement from a higher-numbered STE to a lower-numbered STE causes all other STEs assignments between the two values to shift up, requiring an update to all other edges involving these physical STEs.

E. FPGA Floorplanning

The Stratix 5 GX A7 contains 128 rows and 209 columns of equal-sized blocks , where each block can be one of a LAB, MLAB, M20K RAM, or DSP block. Each LAB/MLAB contains 10 ALMs (Adaptive Logic Module), each of which containing 64x1 bits of LUT RAM and four flip-flops, although the user may only may use the LUTs as RAM in the MLABs blocks (not the LAB blocks).

As shown in Fig. 4, FPGA is logically divided into a left and right half by two large phase locked loops (PLL) on the top and bottom of the chip, each comprising an area that would otherwise contain of 21 rows by 8 columns of blocks. These PLL areas are not programmable but they are traversed by horizontal routing tracks. Using location constraints (logic lock regions) we divided the FPGA into 256 areas, two per block row. We map each set of N / 256 STEs to each of these regions, starting from the upper-left and proceeding using a zig-zag pattern to the right, down, left, down, right, etc. Because the interconnection pattern flows to the right, this assignment pattern minimizes wire length.

V. Evaluation

In this section we give results that characterize both our hardware implementation and the effectiveness of our placement algorithm.

A. Overlay Configurations

To characterize the cost of the interconnect we synthesized overlays of various sizes and searched for the corresponding maximum hardware fan-out for each array size.

Table 1 summarizes the Pareto optimal frontier for our overlay designs. The first column shows the number of STEs divided by 1024. The second column shows the maximum hardware fan-out achieved with the FPGA design flow without exceeding the FPGA's resources. The third column shows the maximum clock frequency. As a comparison, the Micron AP's clock operates at 133 MHz [21]. The fourth column shows the FPGA resource usage. The last three columns show the on-chip memory required by the corresponding design. The total memory includes the 128 KB input buffer and the 1024-deep output buffer. Our maximum array size is 24K STEs with a hardware fan-out of 5, while our maximum hardware fan-out is 24 with 8K STEs.

B. Interconnect Scaling

Table 2 shows the impact of the interconnection network on the utilization of routing resources in the FPGA for the 16K STE design with hardware fan-out from 6 to its maximum of 14.

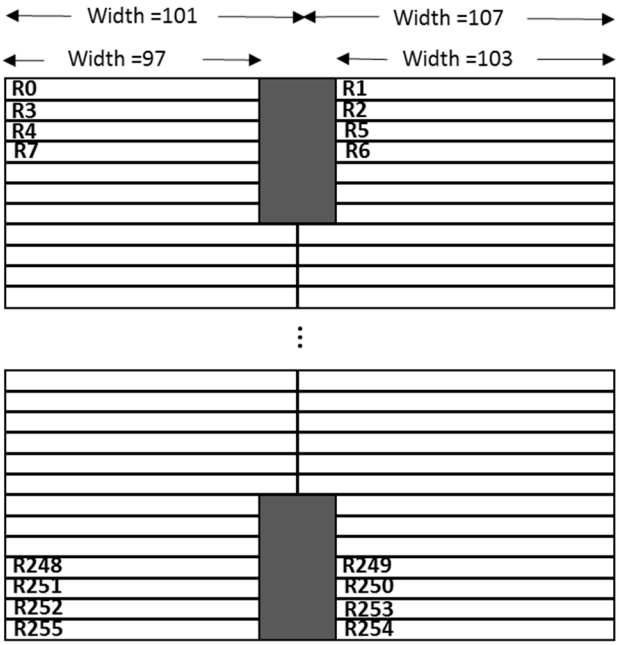

Fig. 4. FPGA floorplan. The dark rectangles on the top and bottom of the chip are non-programmable areas reserved for the PLLs, although some horizontal routing tracks run over them. Our design consists of 256 placement regions, two per row, across the 128 rows. Each region contains *N*/256 STEs, where *N* = the number of instanciated STEs. The STEs connect left-to-right, so the design mostly relies on horizontal routes for STE-to-STE connections, while it uses vertical routes for input and output.

Each column shows the utilization of different types of routing resources in the FPGA fabric.

In Intel's technology, **block interconnect** (column 2) connects pairs of ALMs from adjacent blocks. **Local interconnect** (column 3) connects ALMs within a LAB or MLAB. **R24**, **R3**, and **R6** are longer-reach wires that run horizontally along a row of blocks. The utilization of each of these interconnect types scale with the hardware fan-out. The last column gives total LAB utilization. As shown in the table, the hardware fan-out is limited by LABs--possibly required by the *f*-input OR gates within each STE--as opposed to fabric-level interconnect utilization required by the additional routes between STEs.

C. Mapping ANMLZoo Benchmark Suite to the Overlay

In order to evaluate the suitability of the overlay for realistic workloads, we performed an analysis of the NFA benchmarks in the ANMLZoo benchmark suite [17]. The NFA optimization tool VASim [18] allows for trading off logical STEs and the maximum number of incoming and outgoing transitions per STE (logical fan-in and fan-out). Note that the logical fan-in and fan-out values do not equate to the hardware fan-out in the overlay, since the hardware fan-out defines the maximum "reach" of each STE-to-STE connection. Also, each STE connects to a set of STEs that are shared among multiple STEs, potentially causing placement contention. As such, one of the challenges when implementing these benchmarks onto the overlay is find a valid mapping for a given benchmark.

TABLE 1: OVERLAY CONFIGURATIONS

STEs (K)	Max. H/W Fan-out	Fmax (MHz)	ALMs	MLAB mem. (Mbits)	Reg. (Kbits)	Total mem. (MB)
4	24	152	42%	1	104	0.6
8	24	136	77%	2	208	1.6
12	23	122	95%	3	300	2.3
16	14	121	96%	4	256	2.9
20	8	119	93%	5	200	3.4
24	5	112	95%	6	168	4.0

TABLE 2: IMPACT OF INTERCONNECT SCALING

H/W Fan-out	Block intrcon't	Local interconnect	R24	R3	R6	# LABs utilized
6	25%	21%	25%	14%	25%	81%
10	34%	25%	33%	18%	35%	94%
11	35%	27%	31%	20%	36%	95%
12	42%	29%	38%	29%	46%	97%
14	44%	32%	41%	30%	47%	98%

Table 3 shows the results of our analysis. The second column shows the number of STEs required by the original, unoptimized version of the benchmark design with Boolean elements removed, except for Protomata which we must optimize to restrict maximum fan-out due to its unusually high native fan-out of 109. Columns three and four show the maximum fan-in and fan-out and column four shows the minimum hardware fan-out needed to find a valid mapping solution.

Our algorithm fails to map PowerEN, Snort, DotStar, and Protomata with a reasonable hardware fan-out due to their logical interconnect complexity. In these cases, the greedy nature of our placement algorithm leads to infinite repeating remapping sequences. We plan to address this problem in a future version of our mapping algorithm.

In its current form, the algorithm is not practical due to the difference between its minimal hardware fan-out constraints, shown in the last column of Table 3, and the achieved hardware fan-out in the synthesized overlay architecture, shown in the second column of Table 1. We will continue to develop both the mapping algorithm and hardware design to close this gap.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we described a parameterizable overlay architecture for automata processing on an FPGA. The current version of the overlay supports up to 24K state transition elements (STEs) on a previous-generation 28 nm Intel Stratix 5 GX A7 FPGA, as compared to 48K of the Micron Automata Processor. The overlay uses a point-to-point non-switched programmable interconnect, simplifying the FPGA implementation at the cost of increased constraints.

In future work we will work to develop a more effective placement algorithm. One possible approach is to partition the target NFA topology into smaller, independent sub-automata that that placer can place more effectively. This will have the

added benefit of allowing NFAs that would not other fit on the overlay.

We also plan to explore more flexible more flexible designs for our programmable interconnect, including switched designs.

Finally, we plan to change our configuration strategy from JTAG to using a combination of PCI-express and DDR3 DRAM to improve the rate at which the array is configured.

TABLE 3: ANMLZOO RESULTS

ANML Benchmarks	#STEs	Maximum Logical Fan-in	Maximum Logical Fan-out	Minimum Hardware Fan-out Achieved
Brill	26668	4	4	42
Clam AM	49538	11	2	22
Levenshtein	2784	8	5	22
Hamming	11346	4	2	85
SPM	100500	3	2	22
EntityResolution	95136	28	5	200
RandomForest (300f_15t_tree)	75340	2	2	7
PowerEN (01000_00123)	40513	4	3	cannot place
Snort (after removing special elements)	69029	19	19	cannot place
Fermi	40783	2	2	27
DotStar (after removing special elements)	96438	2	2	cannot place
Protomota (after removing special elements)	42061	3	9 (optimized)	cannot place

REFERENCES

[1] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," IEEE Transactions on Parallel and Distributed Systems 25.12 (2014): 3088-3098.

[2] 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, San Jose, California, USA, 2006, pp. 93-102.

[3] R. Sidhu, and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in IEEE Symposium on Field-Programmable Custom Computing Machines, 2001.

[4] 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, San Jose, California, 2008, pp. 30-39.

[5] Y.-H.E. Yang and V.K. Prasanna, ''High-Performance and Compact Architecture for Regular Expression Matching on FPGA,'' IEEE Trans. Comput., vol. 61, no. 7, pp. 1013-1025, July 2012.

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

[7] Becchi, Michela, and Patrick J. Crowley. "Data structures, algorithms and architectures for efficient regular expression evaluation, Washington University, St." Louis, MO (2009).

[8] Chen, Xinming, et al. "Picking pesky parameters: Optimizing regular expression matching in practice." IEEE Transactions on Parallel and Distributed Systems 27.5 (2016): 1430-1442.

[9] K. Wang, M. Stan, K. Skadron, "Association Rule Mining with the Micron Automata Processor," in IEEE 29th International Parallel and Distributed Processing Symposium, May 2015

[10] K. Zhou, J.J. Fox, K. Wang, D.E. Brown, K. Skadron, "Brill tagging on the Micron automata processor," in IEEE 9th International Conference on Semantic Computing (ICSC), pp. 236–239, 2015.

[11] K. Zhou, J. Wadden, J.J. Fox, K. Wang, D.E. Brown, K. Skadron, "Regular expression acceleration on the Micron automata processor: Brill tagging as a case study," IEEE International Conference on Big Data (Big Data 2015).

[12] T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, G. Robins, "Nondeterministic Finite Automata in Hardware – the Case of the Levenshtein Automaton," in 5th International Workshop on Architectures and Systems for Big Data (ASBD), in conjunction with the 42nd International Symposium on Computer Architecture (ISCA 2015).

[13] I. Roy, S. Aluru, "Finding Motifs in Biological Sequences Using the Micron Automata Processor," in IEEE 28th International Parallel and Distributed Processing Symposium, pp. 415–424, May 2014.

[14] K. Angstadt, W. Weimer, and K. Skadron, "RAPID programming of pattern-recognition processors," in Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, pp. 593–605, 2016.

[15] Vaughn Betz, Jonathan Rose, "VPR: a new packing, placement and routing tool for FPGA research," Proc. 1997 International Workshop on Field Programmable Logic and Applications, pp 213-222.

[16] Wadden, Jack, Samira Khan, and Kevin Skadron. "Automata-to-Routing: An Open-Source Toolchain for Design-Space Exploration of Spatial Automata Processing Architectures." Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on. IEEE, 2017.

[17] J. Wadden, et al. "ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," Workload Characterization (IISWC), 2016 IEEE International Symposium on. IEEE, 2016.

[18] J. Wadden, K. Skadron. "VASim: An open virtual automata simulator for automata processing application and architecture research," Technical Report CS2016-03, University of Virginia, 2016.

[19] Yuanwei Fang, Tung T. Hoang, Michela Becchi, Andrew A. Chien, "Fast Support for Unstructured Data Processing: the Unified Automata Processor," Proc. MICRO-48, 2015.

[20] M. Nourian, X. Wang, X. Yu, W. Feng and M. Becchi. 2017. Demistifying Automata Processing: GPUs, FPGAs or Micron's AP? In Proceedings of ACM International Conference on Sup

[21] Wang, Ke, et al. "An overview of micron's automata processor," Proc. Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. ACM, 2016.ercomputing, Chicago, Illinois USA, June 2017 (ICS'17), 11 pages.