Appearing in *IEEE International Conference on Field-Programmable Technology* (FPT 2019), December 11–13, 2019

Pipelined Parallel Finite Automata Evaluation

Vipula Sateesh, Connor Mckeon, Jared Winograd, and André DeHon
Dept. of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA
Email: vipula@seas.upenn.edu, connor@mckeon6.com, jarwino@gmail.com, andre@ieee.org

***Abstract*—Finite automata are key compute models in modern computational theory and important building blocks for digital logic used for regular expression and protocol parsing, filtering, and control. Finite automata evaluation would seem to be a sequential operation, since we need to complete the evaluation of one state to know the next state in which to evaluate the logic. Nonetheless, parallel theory provides strategies for parallel finite automata evaluation. We show how to exploit this parallel evaluation strategy in practice on today's high capacity FPGAs, including a novel formulation for spatially pipelined evaluation. For non-deterministic finite automata (NFA) with S states, we can evaluate N inputs in a single cycle with $O(N \cdot S^2)$ BRAMs and $O(N \cdot S^3)$ LUTs. This allows us, for example, to consume 64 inputs on a 16 state NFA in a single cycle on the Xilinx XZCU9EG-ffvb1156-2-i SoC FPGA, achieving 47 GB/s (377 Gb/s) single stream throughput for 8b inputs. For a 40 Gb/s network link, we can support 28 state NFAs.**

I. Introduction

The Finite Automata (FA) model is an important, restricted compute model. FA are less powerful than Turing Machines or Push Down Automata that have unlimited state, but capture all computations that operate with limited (finite) state. They directly capture the complexity of regular expression parsing which shows up in lexing, protocol processing, filtering (including intrusion detection [1]) and compression.

FA computations are state-dependent, meaning we must know their current state to compute the output, including the next state. This suggests we must compute each state transition in sequence. However, early theory work [2], reviewed in Sec. III, has shown that we can reformulate state transitions as associative operations so that we can apply associative reduce techniques, similar to the ones used for parallel-prefix addition [3], to perform state evaluation in parallel. For an S-state FA, these parallel computations do perform $O(S)$ more work than their sequential counterparts, so the parallel transformation represents a work-delay tradeoff. In the past this made the parallel FA solution viable only on large parallel supercomputers, like the Connection Machine [4].

After decades of Moore's Law scaling [5], modern FPGAs now have adequate capacity to realize these parallel designs. Furthermore, as the end of Dennard Scaling [6], [7] has slowed frequency scaling [8], [9], there is increasing demand to turn the exponential growth of gates into performance without scaling clock frequencies. Here, we show practically how to perform this parallelism transformation for modern FPGAs. Specifically, we formulate the parallel FA evaluation problem for spatial computation on FPGAs exploiting embedded memories and LUTs and using the direct mapping of NFAs to FPGAs [10] (Sec. II-B). Our pipelined solutions (Sec. IV) go beyond the original parallel-prefix FA designs, showing that the designs can be fully pipelined such that N inputs can be consumed on every cycle. These pipelined parallel solutions are particularly valuable for regular expression parsing that are largely insensitive to evaluation latency. While the area in our solutions scales quickly in the number of states—$O\left(S^3\right)$ scaling for a FA with S states—the area scales linearly with the speedup, N. We detail FPGA designs that are specific to the number of states, S, but only require reprogramming of memories to support any FA up to a given S.

Conventional FA implementations that consume a single input per cycle can operate at the top frequency of the FPGA or the embedded RAMs used for state lookup (e.g., 738 MHz on speed-grade -2 Xilinx Zynq Ultrascale+ devices). Consuming 8b inputs, this limits the designs to 738 MB/s throughput on a single input stream. With our techniques, we can multiply this throughput by a factor of N. Depending on the number of states in the NFA, we can fit designs with up to N=256 on Xilinx MPSoC Zynq FPGAs, meaning the same FPGAs can process a single data stream at guaranteed, real-time data rates of 196 GB/s. The real-time processing guarantees make these designs suitable for use in network switch line cards or network interface cards.

Our novel contributions include:

1) extension of parallel FA evaluation to direct NFA implementations (Sec. III)
2) spatial pipeline design to evaluate N inputs to an NFA per cycle suitable for FPGA implementation (Sec. IV)
3) concrete mappings to Xilinx UltraScale+ FPGAs quantifying resources and latency as a function of the number of states, S, and the inputs per cycle, N, for NFAs (Sec. V)

II. Background

A. Deterministic Finite Automata

A deterministic finite automaton (DFA) is a compute model parameterized by an input alphabet, A, a set of states Q, a start state q_0, a set of accepting states, T, and a transfer function, F, that takes a current state, q_i and input symbol, I_i, and determines the next state, q_{i+1}.

$$q_{i+1} = F(q_i, I_i) \quad (1)$$

The transfer function F is a pure function, returning a single next state q_{i+1} for each state, input pair. The DFA accepts an input string I if the DFA ends in an accepting state after processing all the characters I_i in I. For simplicity in defining asymptotic results, we define S to be the number of states in Q or $|Q|$. Fig. 1 shows a simple DFA.

© 2019 IEEE

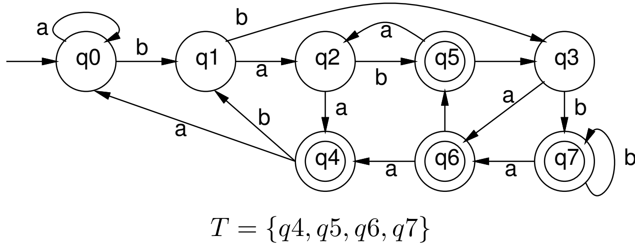


Fig. 1. Sample DFA

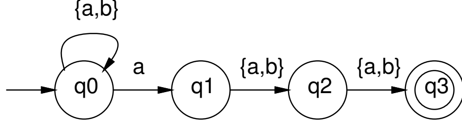


Fig. 2. Sample NFA for Regular Expression `(a|b)*a(a|b)(a|b)`

B. Non-Deterministic Finite Automata

Non-deterministic finite automata (NFA) have similar parameters to DFA, but the transfer function F is allowed to specify multiple next states. Furthermore, the NFA allows transitions to occur without consuming an input (ϵ-transitions). An NFA accepts a string if any of the allowed transitions leads to an accepting state. Since the NFA can be in multiple states at any point in time, it is more useful to formulate its transfer function as mapping from sets of states, R, to sets of states:

$$R_{i+1} = F(R_i, I_i) \tag{2}$$

Any NFA can be converted into a DFA [11]. The basic construction is to consider simulating the NFA being in all the possible states and compute the next state as the set of possible next states from the set of possible current states. This means the states in the DFA-converted NFA may represent the power-set of states in the original NFA. As a result, it is possible the DFA-converted NFA may have exponentially more states than the NFA. Fig. 2 shows a simple NFA. Fig. 1 above is a conversion of this NFA into a DFA.

In register-rich FPGAs, it is possible, and often more efficient, to directly track the NFA state set [10]. That is, we can represent the NFA state as a bit vector representing the set of states the NFA is currently in. On each input, the NFA next state evaluation computes the set of states that the NFA may transition into based on the input and the set of previously occupied states. Fig. 3 shows the logic directly implementing the NFA from Fig. 2. [10] and subsequent work shows that this direct NFA implementation is often a better implementation for FPGAs than performing a DFA conversion and implementing the DFA. Micron's Automata Processor directly builds on this construction [12].

Any regular expression can be recognized by an NFA [13]. The NFA example in Fig. 2 recognizes the regular expression `(a|b)*a(a|b)(a|b)` from [14].

In a sense, the NFA evaluation is a parallel evaluation, since it evaluates the transition functions on all the potential states

$$\begin{aligned} R_i &= [q0_i, q1_i, q2_i, q3_i] \\ q0_{i+1} &= \mathbf{true} \\ q1_{i+1} &= q0_i \wedge a_i \\ q2_{i+1} &= q1_i \\ q3_{i+1} &= q2_i \end{aligned}$$

a_i indicate that character i is an 'a'

Fig. 3. Direct NFA Implementation of Fig. 2

simultaneously. Furthermore, the NFA can be expanded in parallel with more states to simultaneously compute matches against different regular expressions. However, even with this form of parallelism, the NFA is still processing only a single input character on each cycle. In contrast, we show how to process multiple input characters on each cycle.

III. Parallel NFA Evaluation

Ladner and Fischer observe that we can use a parallel-prefix computation to evaluate the DFA state transformation across a sequence of inputs in logarithmic time [2]. The key observation is to treat the evaluation of the DFA across multiple inputs as function composition and exploit the associativity of function composition to compute the composite function across N inputs as an associative reduction in $\log(N)$ steps. We extend this idea to direct NFA evaluation, but the basic strategy remains the same.

A. Strategy

We start by observing that we can *specialize* the next state transition function, F, with respect to an input symbol a as $F_a(R) = F(R, a)$. The specialized function, $F_a(R)$, is a functional *state transform* mapping from state sets to state sets that tells us how input a (or I_j) transforms from state set R_j to R_{j+1}. Now, if we want to know how a pair of inputs I_j and I_{j+1} transform the state from set R_j to R_{j+2}, we can compute the composite function $F_{j,j+1}(R) = F_{j+1}(F_j(R))$. This composite function is also a state set transform, just like the single input state transform function. Similarly, if we want to know how a sequence of 4 inputs I_j, I_{j+1}, I_{j+2}, I_{j+3} transform the state, we can compose functions $F_{j,j+3}(R) = F_{j+2,j+3}(F_{j,j+1}(R))$. $F_{j,j+3}(R)$ is also a state set transform. We can continue in this manner to compute the state set transform $F_{j,j+N-1}(R)$ in $\log_2(N)$ state composition steps. Note that this is a tree reduce so only requires $N-1$ total state set transform composition computations.

B. Matrix Formulation

For NFAs, it is useful to formulate F and the state transformers as binary matrices. We start by representing F as a three-dimensional matrix $FSM[a, q_i, q_{i+1}]$ of dimension $|A| \times S \times S$, which holds a one at each position (a,q_i,q_{i+1}) when the FA in state q_i on input a can transition (including all ϵ-transitions) to state q_{i+1}; equivalently, $FSM[a, q_i, q_{i+1}]$

a	$q0_i$				$q1_i$				$q2_i$				$q3_i$			
	$q0_{i+1}$	$q1_{i+1}$	$q2_{i+1}$	$q3_{i+1}$	$q0_{i+1}$	$q1_{i+1}$	$q2_{i+1}$	$q3_{i+1}$	$q0_{i+1}$	$q1_{i+1}$	$q2_{i+1}$	$q3_{i+1}$	$q0_{i+1}$	$q1_{i+1}$	$q2_{i+1}$	$q3_{i+1}$
'a'	1	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
'b'	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0

Fig. 4. Matrix Representation for Sample NFA from Fig. 2

$$ST_a = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, ST_b = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$ST_{ab} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, ST_{ba} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 5. Specialized and Compose State Transforms for NFA from Fig. 2

is one only when $q_{i+1} \in R_{i+1}$ for $R_{i+1} = F(\{q_i\}, a)$. Fig. 4 shows the FSM matrix for the Sample NFA in Fig. 2.

A state transform, ST, is a two-dimensional matrix, $S \times S$, that has a one in each position $ST[s,t]$, if an input state set that includes state s will transform to an output state set that has state t. The specialized state transformer, ST_a, is then easily computed by selecting the $S \times S$ submatrix in FSM associated with input a; that is: for each (s,t), $ST_a[s,t] = FSM[a,s,t]$. Fig. 5 shows the state transforms that result from specializing the NFA in Fig. 2 to each of the potential input symbols.

Similarly, composition of state transformers is simply a binary matrix-matrix multiplication on the state transformers. For each entry, (s,t), in the state composer matrix, we compute the dot product:

$$NST_{compose}[s][t] = \bigvee_{j=0}^{S-1} NST_{1st}[s][j] \wedge NST_{2nd}[j][t] \quad (3)$$

That is, if there is any j such that the first transform can activate state j on input state s and the second transform activates state t on state j, then the composed transform will activate state t on input state s. Fig. 5 also shows the composite state transforms for example sequences of input symbols.

Once we have the state transform $F_{j,j+N-1}(R)$, we can apply it to state set R_j to compute R_{j+N} in one evaluation of the composite function $F_{j,j+N-1}(R)$. Given the binary matrix representation of the NFA state transformer, NST, we can evaluate the transformation it applies to a state by performing a binary vector-matrix multiplication between the input state vector and the transformer matrix. For each state, we compute the dot product:

$$R_{out}[t] = \bigvee_{j=0}^{S-1} R_{in}[j] \wedge NST[j][t] \quad (4)$$

$$R \times ST_{ab} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$$

Fig. 6. State Transformation Example using Composite State Transformer ST_{an} from Fig. 5

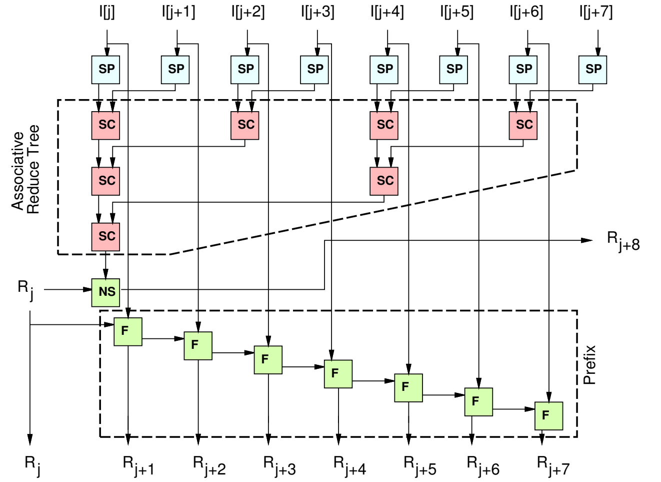


Fig. 7. Parallel-Prefix NFA Evaluation for $N = 8$ Inputs

Fig. 6 shows the application of a composite state transformer to a sample current state.

To fully evaluate the FA, we will often want to know all the intermediate states, R_{j+k} for each k from 1 to $N-1$, that is the *prefix* of states; this allows us to identify any accepting states visited in the NFA. While there are clever ways to compute the prefix in logarithmic depth, a simple way to compute the prefix is with a sequence of F calculations. Once pipelined, this is a good implementation for the size of designs we can currently build.

C. Putting it Together

Putting this together, Fig. 7 shows the parallel-prefix computation for $N = 8$. SP is the specializer that computes $F_i(R)$ from $F(R, I_i)$. SC is the matrix-matrix multiplication state transform composer that computes $F_{j,j+m}(q)$ from $F_{j,j+k-1}(q)$ and $F_{j+k,j+m}(q)$ (Eq. 3). NS computes the next state by performing the vector-matrix multiplication to evaluate a composed ST for a particular state input (Eq. 4), and F is the original next state computation based on a character input.

IV. Spatially Pipelined Parallel NFA Evaluation

The parallel-prefix formulation shows that it is possible to compute the NFA next state for N characters in logarithmic delay. However, the serial prefix still takes $O(N)$ time for evaluation, so offers no speedup. The trick is that this configuration (Fig. 7) is now highly pipelineable, which we now show in this section. We also look concretely at how this maps to modern FPGAs.

The first thing to note from the parallel-prefix formulation, perhaps best seen in Fig. 7, is that the critical path from current state to the N'th next state is a *single next state evaluation*, NS. The critical path for this next state evaluation can be a single S-input OR (Eq. 4). Everything that occurs before the next state evaluation, the specializers and the associative reduce tree, are not in the critical path to advancing the state. Similarly, the sequential state evaluation that follows the next state evaluation is not necessary to evaluating the *next* N'th next state. That means, when we pipeline this hardware to consume N inputs per cycle, the only dependence between iterations is the NS evaluation of R_{j+N} from R_j at the bottom of the associative reduce tree. Once pipelined, R_{j+N} becomes the R_j input for the next state computation (See Fig. 10), creating the only cycle limiting throughput (the Initiation Interval (II)).[1] We can pipeline specialization, associative reduce, the prefix, and the final state evaluations to the point where they meet the throughput of this central next state evaluation. This means the throughput for an N-input parallel evaluation should be N times the rate at which a single next state evaluation can be performed, or roughly a factor of N speedup.

The latency from $I[j]$ to R_j will still be $O(\log(N))$. This will make the spatially pipelined version most attractive for parsing where there is no latency constraint.

A. Next State (NS)

For the highest throughput pipelining, we will want to be able to perform each of the operations in the pipeline spatially, ideally in a single cycle. As previously noted, the next state evaluation can be performed as vector-matrix multiplication (Eq. 4), which is a set of S parallel binary dot products on pairs of S-input vectors. A single 6-LUT can take in 3 inputs from each of the vectors and compute the OR3 of the AND of the pairs (See Fig. 8). After this, we can build an OR-reduce out of 6-LUTs, for a total of:

$$NS_{LUTs} = \frac{2S}{6}\left(1+\frac{1}{6}+\frac{1}{6^2}+\cdots\right) \leq \left(\frac{S/3}{1-\frac{1}{6}}\right) = \frac{2S}{5} \quad (5)$$

This means that each dot product will require $\frac{2S}{5}$ 6-LUTs. There are S dot products for a total of $\frac{2S^2}{5}$ 6-LUTs. The central next state evaluator is the only cycle in the graph, and its asymptotic delay, ignoring 2D-VLSI interconnect delay, is $O(\log(S))$.

[1]With additional transformation this II can be reduced as well, but page limitations prevent elaboration of those transforms.

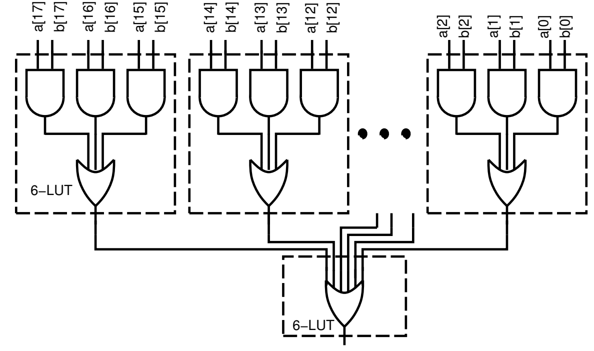


Fig. 8. Binary Dot-Product Mapped to 6-LUTs (shown 18b vectors)

B. Specializer (SP)

The simplest specializer is a memory lookup (Fig. 9). Given the input character $I[j]$, a memory produces the S^2 output bits for a specialized state transformer matrix. Each Virtex 36Kb BRAM can produce up to 72b of output for up to 512 address inputs. For input alphabets, A, with no more than 512 symbols ($|A| \leq 512$), we will need $\frac{S^2}{72}$ such BRAMs per specializer. To operate fully spatially, a configuration consuming N inputs will need N specializers. So, the total BRAMs needed for specialization will be:

$$SP_{BRAMs} = N\left\lceil\frac{S^2}{72}\right\rceil \quad (6)$$

C. State Composer (SC)

To compose states, we perform the binary matrix-matrix multiplication on the state transformer matrices (Eq. 3). The basic computation is a dot product just like the binary vector-matrix multiplication for the next state. Here, as with any matrix-matrix multiply, we must perform one dot product for each element in the output matrix, for a total of S^2 dot products, meaning the whole state composer requires $\frac{2S^3}{5}$ 6-LUTs. Logically, this is the same latency as the next state evaluator since all the S^2 dot products can occur in parallel. Since this has high wiring complexity, $O(S^3)$ gates, its layout may be more complicated and require additional pipelining to meet the rate of the NS unit. The entire associative reduce tree requires $N-1$ state composers.

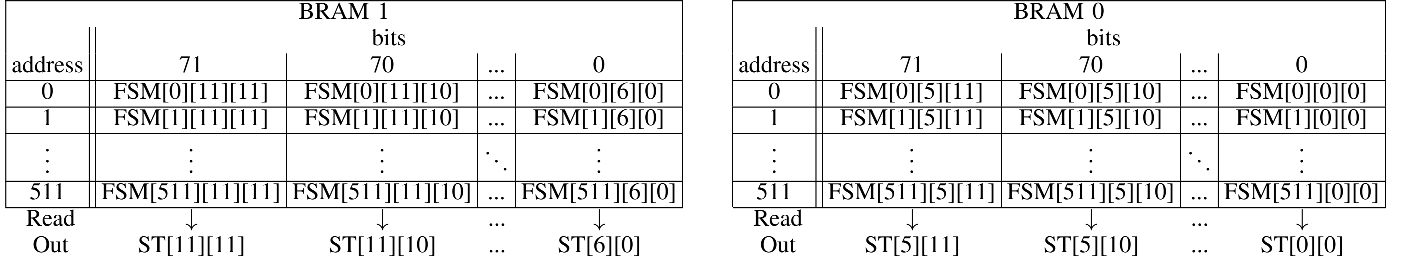
D. Prefix

The prefix portion of the computation uses only F units that can be built as pairs of SP and NS units. We can reduce the length of the pipeline by using the $ST_{j,j+\frac{N}{2}}$ transformer to compute the next state plus $N/2$. This leaves us with two prefix pipelines of length $N/2$, and we call this design 2-pipe prefix. The 2-pipe prefix requires N NS units and N SP units. Fig. 10 shows the complete pipelined, 2-pipe parallel-prefix computation.

E. Pipeline Registers

The pipeline registers grow large. Pipelining the reduce requires $O(N \cdot S^2)$ registers. The need to carry characters

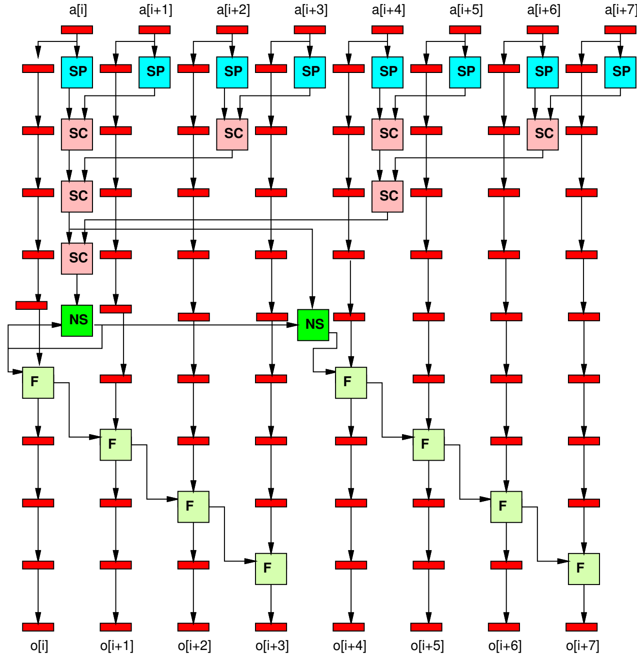
Symbol $I[j]$ input is used as the lookup address to all the BRAMs in the SP:



BRAM 1				
	bits			
address	71	70	...	0
0	FSM[0][11][11]	FSM[0][11][10]	...	FSM[0][6][0]
1	FSM[1][11][11]	FSM[1][11][10]	...	FSM[1][6][0]
⋮	⋮	⋮	⋱	⋮
511	FSM[511][11][11]	FSM[511][11][10]	...	FSM[511][6][0]
Read Out	↓ ST[11][11]	↓ ST[11][10]	...	↓ ST[6][0]

BRAM 0				
	bits			
address	71	70	...	0
0	FSM[0][5][11]	FSM[0][5][10]	...	FSM[0][0][0]
1	FSM[1][5][11]	FSM[1][5][10]	...	FSM[1][0][0]
⋮	⋮	⋮	⋱	⋮
511	FSM[511][5][11]	FSM[511][5][10]	...	FSM[511][0][0]
Read Out	↓ ST[5][11]	↓ ST[5][10]	...	↓ ST[0][0]

Fig. 9. Specializer from 36Kb BRAM [512×72 organization] (shown $S = 12$, $\log(|A|) \leq 9$)



Red boxes denote pipeline registers.

Fig. 10. Spatially Pipelined, Parallel-Prefix FA Evaluation (shown $N = 8$)

forward until used in the pipelined sequential prefix means registers grow as $O(N^2C)$, where C is the number of bits in each character. In SLR32 mode on the Virtex architecture, the structure supporting the 6-LUTs can also be used as a shift register of length up to 32. As a result, many of the long register runs can be implemented compactly. Register requirements do not dominate for the size of designs we can place on today's FPGAs.

F. Putting it Together

Assuming $\log_2(|A|) \leq 9$, design requirements are summarized in Tab. I. For simplicity, Tab. I assumes all units (SP, SC, NS, F) operate on the same unit clock. In practice, the NS cycle limit grows as $O(\log(S))$ gate delay. The other units may require more pipeline stages to match the NS cycle delay.

TABLE I
RESOURCE ESTIMATES ($\log(|A|) \leq 9$, UNIFORM UNIT LATENCY)

Parallel Prefix NFA	BRAMs	$2N\left\lceil\frac{S^2}{72}\right\rceil$
	6-LUTs	$2N\left(\frac{S^3}{5}\right)+2N\frac{S^2}{5}$
	Regs.	$O\left(N\cdot S^2+N^2\cdot C\right)$
	Latency	$\log_2(N)+N$

TABLE II
XILINX ZYNQ PART CHARACTERISTICS

Xilinx Family	Artix-7	UltraScale+
Tech. Node	28 nm	16 nm
Board	ZedBoard	ZCU102
Part	XC7Z020-clg484-1	XCZU9EG-ffvb1156-2-i
6-LUTs	53,200	274,000
36Kb BRAMs	140	913
Transceivers	0	24×16.3Gbps
Memory Freq.	388 MHz [15, Tab. 65]	738 MHz [16, Tab. 80]

V. FPGA MAPPING

The previous sections have shown how we can formulate parallel and pipelined NFA evaluation. In this section, we look concretely at how that maps to current FPGA technology. We specifically characterize how many NFA states are currently viable with what level of parallelism. We also characterize the performance they provide and the pipelining required.

A. Key Technology Characteristics

Concretely, we map to Xilinx Virtex SoC FPGAs from the Zynq and MPSoC families. These SoCs include embedded ARM processors as well as the FPGA fabric. Our designs only exploit the LUTs and BRAMs in the FPGA fabric, but we use the embedded processors to manage at-speed validation. Tab. II summarizes the technologies and capabilities of the parts used for characterization.

The peak frequency in our design is set by the maximum operating frequency of the BRAMs implementing the SP (Sec. IV-B, Fig. 9). We use the BRAMs in dual port, NO_CHANGE mode where the peak frequency is 738 MHz as summarized in Tab. II.

Xilinx parts in these families also come with a range of high-speed serial I/O links. Tab. II shows the XCZU9EG has

24, 16.3 Gbps transceivers providing 390 Gbps of I/O bandwidth. The XCZU19EG has 4, 100 Gbps ethernet transceivers.

B. Methodology

We developed a parameterized circuit generator in Python to produce Verilog descriptions of all the building blocks (SP, SC, NS) and connect them together into suitable associative reduce and prefix trees. The generator is parameterized by the number of simultaneous characters to process, N, the maximum number of states, S, and the number of registers to add between stages, r. We tune r to increase pipelining for larger designs to approach the peak operating frequency of the BRAMs as identified above. In some cases, we also pipeline the dot products (Fig. 8) and input fanout in the SC at the LUT level. We map the produced Verilog designs to the FPGA through synthesis, placement, and routing with Vivado 2017.1. We set `-shreg_min_size` to 8 to prevent the added pipeline shift registers between SCs in adjacent tree levels from being turned into SRL32s while allowing the SRL32s to be used for the long pipeline chains that cross tree levels. We enable retiming by setting the `STEPS.SYNTH_DESIGN.ARGS.RETIMING` property to `true`.

We validated function and functionality at speed using a bank of BRAMs to supply input characters to the pipelined, parallel NFA evaluator and to receive the outputs. This allowed us to provide the high throughput input and output capture necessary to fully exercise these designs. We used the embedded ARM processor on the Zynq SoC FPGAs to load the dual-port input BRAMs and offload the dual-port output BRAMs on a slow speed clock, then ran the full throughput tests on a high-speed clock to validate high frequency operation. We loaded a few designs on a ZedBoard to validate correct operation at frequency. While we note (Sec. V-A) that these FPGAs have the I/O bandwidth to feed the high throughput implementations we report, we did not directly connect the pipelined parallel-prefix NFA to off-chip data streams, dividing that as a separate engineering task largely orthogonal to the result we are demonstrating here.

For the bulk experiments that follow, we performed Out-of-Context design flow mapping [17] that only synthesized, placed, and routed the pipelined, parallel-prefix NFA from input registers through SPs, through the SC reduce and prefix, to the final NS computations (Fig. 10). This models the creation of a hard macro block or IP-core that could then be integrated into a larger design. The select designs mapped to the ZedBoard verified that the Out-of-Context frequency was achievable for full designs.

C. Experiments

1) Feasible Parallelism: In Tab. III we generate designs for a variety of NFA states, S, and parallel input characters, N, and report the LUT and BRAM resource usage. This shows it is practical to support designs with tens of NFA states, processing tens to hundreds of state inputs at a time on today's MPSoC Zynq devices. With LUT usage growing as $O(N \cdot S^3)$, for a fixed-capacity device, the number of inputs that can be supported in parallel drops as states grow. LUTs rather than BRAMs tends to be the limiting resource, which matches the asymptotic trend of BRAM needs growing as $O(N \cdot S^2)$, while LUT needs growing as $O(N \cdot S^3)$. The BRAM count in Tab. III exactly matches the equations in Tab. I.

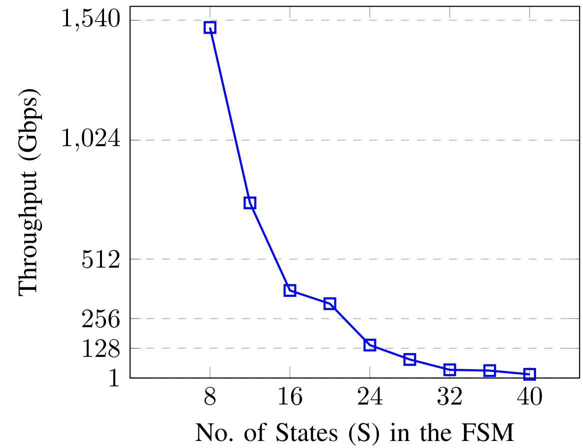


Fig. 11. Feasible Throughput vs. States (S) for Parallel-Prefix NFA Evaluation assuming 8b Characters on XCZU9EG

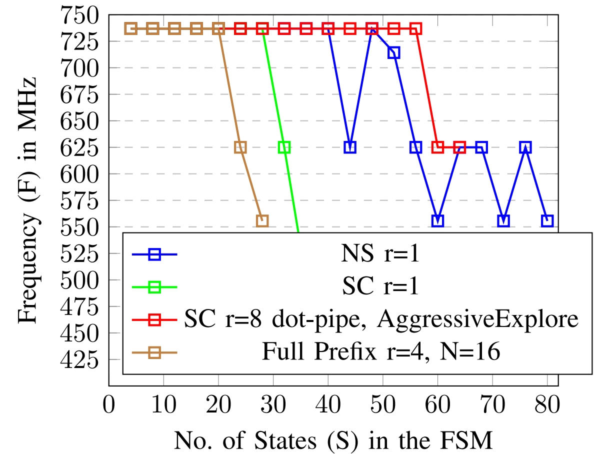


Fig. 12. NS and SC Operating Frequency vs. S

2) Feasible Operating Frequencies: As noted (Sec. IV) the only critical cycle that cannot be pipelined is the next state (NS) calculation. Fig. 12 shows how NS frequency scales with S. For the XCZU9EG, the cycle does not limit frequency until $S > 48$. The curve show small non-monotonic frequency results due to the heuristic nature of the CAD tools.

The state combiner, SC, should be pipelineable to achieve the maximum frequency target. We expect to need larger pipelining, r, as S grows in order to achieve the highest frequency operation. Fig. 12 also shows that a single SC unit can achieve up to $S = 28$ with simple pipelining. By further pipelining the SC at the LUT level, and, in some cases, pipelining the fanout from the inputs of the SC module to the S points of consumption within the SC, we can achieve full frequency operation up $S = 52$ for the XCZU9EG. Beyond $S = 52$, we see operating frequency drop with S. As noted,

TABLE III
REDUCE 2-PIPE INPUT CHARACTERS PIPELINED: RESOURCES VS STATES (S) AND PARALLELISM (N), CELLS SHOW: LUT COUNT (BRAM COUNT)

S	N							
–	2	4	8	16	32	64	128	256
4	67 (4)	147 (8)	307 (16)	700 (32)	1565 (64)	3279 (128)	6691 (256)	13492 (512)
8	312 (4)	744 (8)	1591 (16)	3392 (32)	7137 (64)	14612 (128)	29542 (256)	59383 (512)
12	995 (8)	2555 (16)	5652 (32)	11963 (64)	24822 (128)	50471 (256)	101754 (512)	
16	2536 (16)	6852 (32)	15483 (64)	32780 (128)	67709 (256)	137526 (512)		
20	3980 (24)	10700 (48)	24019 (96)	50968 (192)	105213 (384)	213678 (768)		
24	6888 (32)	18742 (64)	42741 (128)	90812 (256)	187361 (512)			
28	13020 (44)	39507 (88)	90914 (176)	193800 (352)				
32	21151 (60)	60896 (120)	140382 (240)					
36	28743 (72)	84553 (144)	194899 (288)					
40	41350 (92)	119409 (184)						

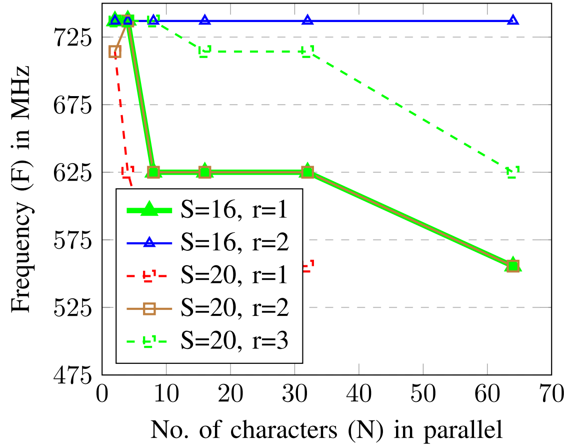


Fig. 13. Frequency vs. N and Pipelining for $S = 16$, $S = 20$

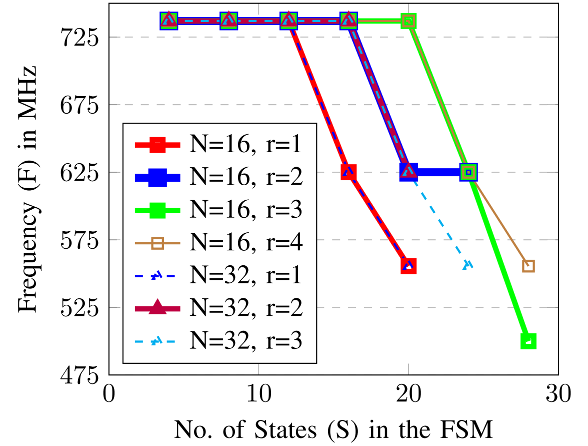


Fig. 14. Frequency vs. S and Pipelining for $N = 16$ and $N = 32$

there is no logical restriction preventing further pipelining, but it will require more careful control of register insertion and placement for the additional pipelining to be effective.

3) Frequency and Pipelining: As the design grows with N, it will need pipelining to accommodate the long wire connections between SC units to prevent the reduce tree from limiting operating frequency. To see how these effects manifest on today's FPGAs, we mapped designs varying r, the number of registers inserted between each SC. Fig. 13 shows how frequency drops as N increases for fixed $S = 16$ and $S = 20$ designs. It includes separate curves for different r, to illustrate how r must grow to maintain high frequency for large N.

Fig. 14 shows how frequency and pipelining requirements vary with S for fixed $N = 16$ and $N = 32$. Up to the $S = 24$ design for $N = 16$, the XCZU9EG is able to achieve near full frequency, but drops at $S = 28$. We know from the NS and SC experiments in Fig. 12 that these state sizes should be able to achieve full frequency. The larger designs may require more aggressive physical optimization options and additional care to achieve peak throughput.

D. Discussion

Our results show that it is feasible to run NFA evaluation at 190 GB/s ($S = 4$) for the MPSoC Zynq. Using these techniques, we can turn growing FPGA resources into throughput without increasing clock frequencies. That means we can ride the capacity scaling of FPGAs to support higher data rates.

The $O\left(S^3\right)$ LUT scaling means this technique will not support NFAs with hundreds or thousands of states anytime soon. However, when the large number of NFA states comes from combining regexps or NFAs, it is possible, and will be better, to decompose the NFA back into its constituent components and run these techniques on the independent NFAs in parallel, as can be easily implemented on an FPGA. This cost model suggest a different optimization criteria for regexp combining or decomposition which should be explored in future work.

Other prefix variants have better asymptotic area characteristics and will likely become interesting for larger designs.

VI. RELATED WORK

As multicore processors and SoCs become the norm and network bandwidth continues to increase, high throughput FA evaluation has gained increasing attention. As applications are parallelized to exploit multicore processors and SIMD datapaths, sequential parsing of input streams threatens to become the sequential bottleneck, limiting achievable performance. As we push to higher network bandwidth, there is a demand for higher throughput packet processing and deep-packet inspection; the push to software-defined networks

TABLE IV
SINGLE-STREAM PEAK THROUGHPUT COMPARISON

Design	Any NFA?	Real-Time?	Part	Thput (Gb/s)
HARE [19]	no	yes	Arria V	3
DP FSM [20]	yes	no	Xeon X5650	24
Automata Processor [12]	yes	yes	Micron AP	1
Parallel AP [21]	yes	no	Micron AP 32× D480 devices	25
Multistrided [22]	yes	yes	Virtex 5 LX-220	11
SFA [23]	yes	no	Xeon E5646	112
this work	yes	yes	ZCU9EG	1500

and in-network functions creates even greater demand for programmable parsing and flexible datastream processing.

Note that the key challenge is providing high throughput on a single, high-speed data stream. If the data stream is composed of multiple, independent data streams to process, each independent stream can be assigned to a separate core or a separate, single-character-per-cycle FA hardware datapath. As long as no single stream exceedes the throughput of a processor core or single-character-per-cycle FA, this simple data parallelism is sufficient. There are many examples in the literature that achieve high throughput regular expression processing only for this multiple, low throughput stream case. For example, [18] achieves 400 Gb/s with a pipelined automaton using a set of parallel FA evaluation pipelines. The rest of the designs in this section address the more challenging single-stream processing problem which our design addresses.

HARE provides a customized architecture for regular expression evaluation [19]. On an Altera Arria V (28 nm technology FPGA), they achieve 3.2 Gb/s of throughput. However, while HARE handles many common regular expression patterns, it does not handle all regular expressions or finite automata. It lacks the generality provided by our architecture.

Recent work explores parallel FA evaluation on multicore processors with a focus on statistical properties that predict the state the FA may be in at key points in the input sequence [20], [24]. Successful prediction, allows the input to be decomposed into subsequences that can be processed in parallel. [20] reports up to 24 Gb/s throughput. These statistical techniques cannot guarantee real-time throughput as our design can.

The Parallel Automata Processor work shows how to exploit parallelism in NFA evaluation on the Micron Automata Processor, but still requires sequential evaluation of state sequences and also relies heavily on statistics of state sequences to reduce the number of state sequences that must be evaluated [21]. Specifically, it does not exploit associative reduction nor show how to pipeline evaluation. They report a factor of 25.5 speedup over sequential processing on the Micron Automata Processor that operates at 1 Gb/s on 8-bit input characters [12].

Note that our work is fundamentally different than the work that deals with multiple characters per cycle under the name multistride [22], [25], [26] or multicharacter [27]. The multistrided techniques unroll the FA evaluation loop so that multiple characters can be processed at once [22]. [22] report speeds of 11 Gb/s, and the key difference is that the next state logic in these multistrided designs still depends on sequentially evaluating the logic of multiple states. It does expose multiple input transitions to logic optimization that can try to reduce the combinational path of the logic, but it still must complete evaluation of the composite path in a single cycle, whereas our design shows how to decompose and pipeline that composite evaluation over many cycles. Furthermore, the multistride implementation can have an exponential increase in area as the alphabet and hence edges increase exponentially with the number of characters (or unroll factor) as measured in [18].

The SFA formulation [23] also works on state transformers (Sec. III-B) by computing a new DFA representing the potential state transforms resulting form a sequence of inputs. This allows the parallel computation of state transformers using DFA state transitions, but, unlike our formulation, the number of states in the SFA are potentially exponential in the number of DFA states. To combine the results of parallel computations in parallel, they still require an associative reduce on state transforms, for which the solution in this paper would directly apply. Running on dual 2.4 GHz Intel Xeon E5646 6-core processors, they report peak throughputs of 14 GB/s, with many cases achieving lower throughputs. Their performance is DFA and input-sequence dependent, whereas ours can provide real-time guarantees based only on the number of FA states.

Previous work shows how to build FSM overlay designs that can be programmed by filling in memories without the need to invoke the slow FPGA CAD flow [28]. Our work can be seen as an extension that shows how to build parallel overlays that can consume N inputs per cycle for any FSM within a bounded number of states, S.

VII. CONCLUSION

We show that it is viable to use parallel-prefix computation to perform NFA evaluation consuming multiple input characters per cycle using today's FPGAs. Furthermore, we show how to pipeline evaluation down to a single S-input OR reduce for an S state NFA. For Ultrascale+ Virtex technology, it is possible to run at the full clock rate of the BRAMs up to S=48 states before next state (NS) evaluation starts to limit clock frequency. This makes it possible to achieve guaranteed real-time throughput on single streams of data into the hundreds of gigabits per second on today's FPGA technology. For NFAs with S states consuming N characters per cycles, LUT area scales as $O\left(N \cdot S^3\right)$. Both the resource scaling with S and the frequency challenges for large S mean the technique is limited to design with small numbers of states. Nonetheless, the throughput can scale roughly linearly with area allocated.

ACKNOWLEDGMENTS

Support for Jared Winograd from the Rachleff Scholar's Program was instrumental in initiating this work. This work is funded in part by the Office of Naval Research under grant N000141512006. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research. Vivado tools were donated by Xilinx.

REFERENCES

[1] B. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *FCCM*, 2002, pp. 111–120.

[2] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980. [Online]. Available: http://doi.acm.org/10.1145/322217.322232

[3] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. 31, no. 3, pp. 260–264, March 1982.

[4] W. D. Hillis and G. L. Steele, "Data parallel algorithms," *CACM*, vol. 29, no. 12, pp. 1170–1183, December 1986.

[5] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proc. IEEE*, vol. 103, no. 3, pp. 318–331, March 2015.

[6] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFETs with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, October 1974.

[7] M. Bohr, "A 30 year retrospective on Dennard's MOSFET scaling paper," *Solid-State Circuits Society Newsletter, IEEE*, vol. 12, no. 1, pp. 11–13, Winter 2007.

[8] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein, "Scaling, power, and the future of CMOS," in *IEDM*, December 2005, pp. 7–15.

[9] S. H. Fuller and L. I. Millett, Eds., *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011. [Online]. Available: http://www.nap.edu/catalog.php?record_id=12980

[10] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *FCCM*, 2001.

[11] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 114–125, April 1959.

[12] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3088–3098, Dec 2014.

[13] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, Jun. 1968. [Online]. Available: http://doi.acm.org/10.1145/363347.363387

[14] O. Stephens, "Determinism costs! a nfa with exponentially bigger dfa," https://www.owenstephens.co.uk/blog/2014/09/28/NFA_DFA.html, September 2014.

[15] *Zynq-7000 SoC: DC and AC Switching Characteristics*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, July 2018, DS187 https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf.

[16] *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, November 2018, DS925 https://www.xilinx.com/support/documentation/data_sheets/ds925-zynq-ultrascale-plus.pdf.

[17] *UG946: Vivado Design Suite Tutorial: Hierarchical Design*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, Aprial 2015. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug946-vivado-hierarchical-design-tutorial.pdf

[18] D. Matousek, J. Korenek, and V. Pus, "High-speed regular expression matching with pipelined automata," in *ICFPT*, Dec 2016, pp. 93–100.

[19] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "HARE: Hardware accelerator for regular expressions," in *MICRO*, 2016, pp. 1–12.

[20] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *Proc. ASPLOS*, 2014, pp. 529–542. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541988

[21] A. Subramaniyan and R. Das, "Parallel automata processor," in *ISCA*, ser. ISCA, 2017, pp. 600–612. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080207

[22] Y. H. Yang and V. Prasanna, "High-performance and compact architecture for regular expression matching on FPGA," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 1013–1025, July 2012.

[23] R. Sinya, K. Matsuzaki, and M. Sassa, "Simultaneous finite automata: An efficient data-parallel model for regular expression matching," in *Proceedings of the International Conference on Parallel Processing*, 2013, pp. 220–229. [Online]. Available: https://ieeexplore.ieee.org/document/6687355

[24] Z. Zhao and X. Shen, "On-the-fly principled speculation for FSM parallelization," in *Proc. ASPLOS*, 2015, pp. 619–630. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694369

[25] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ISCA*, 2006, pp. 191–202.

[26] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proc. ACM/IEEE Symp. ANCS*, 2008, pp. 50–59. [Online]. Available: http://doi.acm.org/10.1145/1477942.1477950

[27] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," in *FPL*, Sept 2008, pp. 131–136.

[28] P. Cooke, L. Hao, and G. Stitt, "Finite-state-machine overlay architectures for fast FPGA compilation and application portability," *ACM TECS*, vol. 14, no. 3, pp. 54:1–54:25, 2015. [Online]. Available: http://doi.acm.org/10.1145/2700082

Web link for this document: <http://ic.ese.upenn.edu/abstracts/parallel_fa_fpt2019.html>