# Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs

Reza Rahimi[1], Elaheh Sadredini[2], Mircea Stan[1], Kevin Skadron[2]
[1]Department of Electrical & Computer Engineering, [2]Department of Computer Science
University of Virginia
Charlottesville, Virginia, USA
Email: {rahimi, elaheh, mircea, skadron}@virginia.edu

***Abstract*—Regular expressions have been widely used in various application domains such as network security, machine learning, and natural language processing. Increasing demand for accelerated regular expressions, or equivalently finite automata, has motivated many efforts in designing FPGA accelerators. However, there is no framework that is publicly available, comprehensive, parameterizable, general, full-stack, and easy-to-use, all in one, for design space exploration for a wide range of growing pattern matching applications on FPGAs. In this paper, we present *Grapefruit*, the first open-source, full-stack, efficient, scalable, and extendable automata processing framework on FPGAs. Grapefruit is equipped with an integrated compiler with many parameters for automata simulation, verification, minimization, transformation, and optimizations. Our modular and standard design allows researchers to add capabilities and explore various features for a target application. Our experimental results show that the hardware generated by Grapefruit performs 9%-80% better than prior work that is not fully end-to-end and has 3.4× higher throughput in a multi-stride solution than a single-stride solution.**

## I. Introduction

Finite automata are an efficient computational model for widely used pattern recognition languages such as regular expressions, with applications in network security [1], [2], log analysis [3], and newly-demonstrated other applications in domains such as data-mining [4], [5], [6], [7], bioinformatics [8], [9], machine learning [10], [11], natural language processing [12], [13], and big data analytics [14] that have been shown to greatly benefit from accelerated automata processing.

Researchers are increasingly exploiting hardware accelerators to meet demanding real-time requirements as performance growth in conventional processors is slowing. In particular, several FPGA-based regex implementations for single-stride [15], [16], [17], [18], [19] and multi-stride [20], [21], [22], [23] automata processing have been proposed to improve the performance of regex matching. These solutions provide a reconfigurable substrate to lay out the rules in hardware by placing-and-routing automata states and connections onto a pool of hardware units in logic- or memory-based fabrics. This allows a large number of automata to be executed in parallel, up to the hardware capacity, in contrast to von Neumann architectures such as CPUs that must handle one rule at a time in each core. Most of the current FPGA solutions are inspired by network applications such as Network Intrusion Detection Systems (NIDS). However, patterns in other applications can have different structure and behavior, e.g., higher fan-outs, and this makes it difficult for NIDS-based FPGA solutions to map other automata to FPGA resources efficiently [18], [24], [25].

To enable architectural research, trade-off analysis, and performance comparison with other architectures on the growing range of applications, an open-source, full-stack, parameterized, optimized, scalable, easy-to-use, and easy-to-verify framework for automata processing is required. REAPR [15] is a reconfigurable engine for automata processing, and generates FPGA configurations that operate very similarly to the Micron Automata Processor (AP) style [26] processing model. The RTL generated from the automata graph is a flat design, which causes a very long compilation time. Due to this flat-design approach, this solution is not scalable and the synthesizer fails to generate RTL for larger designs. Moreover, REAPR only generates the matching kernel and does not provide a full-stack solution or even the automata reporting architecture.

Bo et al. [27] extend REAPR and provide an end-to-end solution on FPGAs using SDAccel for the I/O. However, their I/O design has two issues. First, the input stream should be segmented into limited-size chunks. Second, the reporting structure is very simple; whenever a state generates a report, a long vector (the size of the vector is equal to the number of total reporting states), mostly filled with zeroes, is read and sent to the host. This reporting architecture may become a bottleneck for applications with frequent but sparse reporting, which is a common reporting behavior [28]. Casias et al. [29] also extended REAPR and proposed a tree-shaped hierarchical pipeline architecture. However, their solution generates the HDL code for only the kernel and does not provide a full-stack solution (i.e., broadcasting input symbols to the logic elements and getting the reporting data out the FPGA chip). Furthermore, their source code is not publicly available.

Researchers are interested in using a tool that gives them the flexibility to explore design space parameters comprehensively. For example, in automata processing, symbol size impacts the throughput and hardware cost [30], and none of the prior tools provide support for that. Similarly, reporting architecture can be a performance bottleneck that can reduce the throughput significantly [28] (up to 46X stall overhead in the Micron AP). To improve the performance, Liu et el. [31]

2576-2621/20/$31.00 ©2020 IEEE
DOI 10.1109/FCCM48280.2020.00027

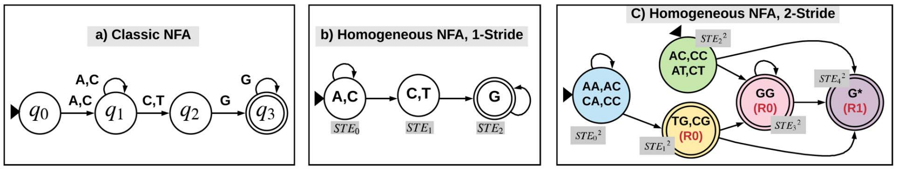

Fig. 1. (a) Classic NFA, (b) Homogeneous NFA, (c) Equivalent 2-stride automata.

propose a hybrid automata processing approach by splitting states between CPU and an automata processing accelerator, and this incurs higher reporting rate on the accelerator. Therefore, an efficient reporting architecture is more critical for the end-to-end performance. In this paper, we present *Grapefruit* (**G**eneral and **R**econfigurable **A**utomata **P**roc**E**ssing **FR**amework **U**sing **I**ntegrated Reporting and In**T**erconnect). We prioritize flexibility, extensibility, and scalability while developing this tool to provide an easy-to-understand interface and easy-to-modify code for other researchers to explore new features and design parameters.

In summary, this paper makes the following contributions:

- We present Grapefruit, **the first open-source, full-stack, comprehensive, and scalable framework for automata processing on FPGAs**[1]. Grapefruit provides an extensive set of compiler optimizations, hardware optimizations, and design parameters for design-space exploration on a wide range of emerging applications.
- We present an optimized pipeline architecture, an adaptive priority-based reporting architecture, and an interconnect model to address the issues in prior tools and support scalability to large numbers of rules/automata. We also investigate LUT-based, BRAM-based, and the combination of both in the design space.
- We present an integrated back-end compiler, with a rich and descriptive interface, to define an automaton and to perform cycle-accurate automata simulation, automata transformation, and automata minimization. An important feature provided in our framework is the support for multi-symbol (multi-stride) processing with variable symbol size, and this directly impacts throughput and hardware cost.
- We perform thorough performance analysis with different optimizations and parameters on a wide range of automata applications on a Xilinx Virtex UltraScale+. Our results confirm that we are achieving 9%-80% higher frequency in a single-stride solution than prior works that are not fully end-to-end (including reporting and I/O) and 3.4× higher throughput in a multi-stride solution than a single-stride solution.

## II. Background

Finite Automata: A regular expression can be represented by either deterministic finite automata (DFA) or non-deterministic finite automata (NFA). A DFA allows only one transition per input symbol. An NFA has the ability to be in several states at once, meaning that transitions from a state on an input symbol can be to any set of states. DFAs, NFAs, and regular expressions are equivalent in computational power (and can be converted to each other), but some applications are easier to express directly as finite automata. However, a DFA can have exponentially more states than an equivalent NFA (this is a side effect of the rule that a DFA can only have one active-state at a time), which greatly increases the memory footprint. On the other hand, an NFA can have many parallel transitions, which is bounded by the limited memory bandwidth in von-neumann architectures. Hardware accelerators for automata processing are based on NFAs, both to exploit parallel state-matching and transitions, and the benefit of the NFA's more compact representation.

Non-Deterministic Finite Automata Primer: An NFA is represented by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite set of symbols, $\Delta$ is a transition function, $q_0$ are initial states, and $F$ is a set of accepting states. The transition function determines the next states using the currently active states, and the input symbol just read. If an input symbol causes the automata to enter into an accept state, the current position of the input is reported.

We use the homogeneous automaton representation in our execution models. In a homogeneous automaton, all transitions entering a state must happen on the same input symbol [32]. This provides a nice property that aligns well with a hardware implementation that finds matching states in one clock cycle and allows a label-independent interconnect. Following [26] and [33], we call this element that represents both a state and performs input-symbol matching in homogeneous automata a *State Transition Element* (STE).

Figure 1 (a) shows an example of a classic NFA and its equivalent homogeneous representation (b). Both automata in this example accept the language $(A|C)^+(C|T)(G)^+$. The alphabets are $\{A, T, C, G\}$. In the classic representation, the start state is $q_0$, and accepting state is $q_3$. In the homogeneous one, we label each STE from $STE_0$ to $STE_3$, so starting state is $STE_0$, and the accepting state is $STE_2$. Figure 1 (c) strides the NFA in (b) and processes two symbols per cycle, and this provides higher throughput. Details of striding is explained in Section IV-B1.

## III. Related Work

A number of multi-stride automata processing engines have been proposed on CPUs and GPUs [22], [34], [35], [36], [37], [38]. Generally, automata processing on von Neumann

[1] https://github.com/gr-rahimi/APSim (check temp_scripts/FCCM folder)

architectures exhibits highly irregular memory access patterns with poor temporal and spatial locality, which often leads to poor cache and memory behavior [24] (which disables prediction and data forwarding techniques [39]). Moreover, multi-symbol processing causes more pressure on memory bandwidth, because more states and transitions need to be processed in each clock cycle.

FPGA implementations of regular expression matching are often inspired by networking applications [15], [16], [17], [18], [20], [40], which typically consist of many independent matching rules. These are often implemented using automata-based (NFA or DFA) computation. Hieu [17] proposes an accelerated automata processing on FPGA using the Aho-Corasick DFA model. DFAs can be easily mapped to BRAM, and reconfiguration in BRAM is cheap. However, a DFA model does not utilize the inherent bit-level parallelism in FPGAs and is better suited to memory-bound von Neumann architectures. Furthermore, DFAs are primarily beneficial for von Neumann architectures, because DFAs only have one active state at any point in time. But combining multiple independent automata or regex rules into a single DFA leads to a rapid blowup in the number of states, while keeping multiple independent automata obviates the main benefit of DFAs. NFAs are thus a better fit for a substrate with high parallelism such as FPGAs or the Micron AP. Karakchi et al. [18] present an overlay architecture for automata processing, which is inspired by the Micron AP architecture. Karakchi's architecture forces fan-out limitation and thus fails to map complex-to-route automata to the routing resources due to its logical interconnect complexity. *Our framework is optimized for general NFA processing, and it provides the user with a variety of parameter and flexibility to optimize the design not only for network regexes but for general and emerging automata applications.*

To improve the throughput of automata processing, some works have investigated multi-striding (multiple symbols per cycle) on FPGAs [20], [21], [22], [23]. Yang et al. [20] proposed a multi-symbol processing solution for regular expressions on FPGA which utilizes both LUTs and BRAMs. Their solution is based on a spatial stacking technique, which duplicates the resources in each stride. This increases the critical path when increasing the stride value. Yamagaki et al. [21] proposed a multi-symbol state transitions solution using a temporal transformation of NFAs to construct a new NFA with multi-symbol characters. This approach only utilizes LUTs and does not scale well due to the limited number of lookup tables in FPGAs. *None of these works provide an open-source tool. Our proposed framework provides a temporal multi-striding solution, and the user can choose to use the LUTs, BRAMs, or the combination of LUT/BRAM for design exploration.*

Furthermore, alphabet compression techniques [36], [41], [42], [43], [44] may be employed to reduce memory requirements in CPU/GPU and FPGA-based solutions. Becchi et al. [41] propose to merge symbols with the same transition rules. This reduces the number of unique alphabets in an automaton. However, 8-bit hardware accelerators cannot benefit from the compression techniques if the reduced alphabet size requires fewer than 8-bit symbols. *Our compiler can be utilized to reshape the compressed automaton to 8-bit symbol processing while keeping the benefit of compression. This provides full hardware utilization on the target FPGA and at the same time, increases the processing rate and throughput.*

REAPR [15] is an FPGA implementation of a single-stride NFA processing engine, and takes advantage of the one-to-one mapping between the spatial distribution of automata states and hardware resources such as lookup tables and block RAMs. REAPR uses VASim [45] as its backend compiler. VASim is hard-coded to 8-bits (256 symbols) and does not provide any freedom to modify the bitwidth. It also does not have any built-in function procedure to handle bitwidth transformations. In addition, the design space exploration for the interconnect does not exist natively in this tool. Further, HDL generation in VASim is not in release mode. Automata generated with VASim needs to be ported by automata descriptor formats such as ANML to other tools such as REAPR [46] to generate HDL code targeting FPGAs, which again works with fixed to 8-bit symbols and does not support other variable bitwidth processing or symbol striding.

## IV. Grapefruit Framework

### A. Architecture

This section describes an overview of the system design for an efficient automata processing on FPGAs using Figure 2 and explains some of the features and optimizations.

*1) Pipeline design:* NFAs for real-world applications are typically composed of many independent patterns, which manifest as separate *connected components (CCs)* with no transitions between them. Each CC usually has a few hundred states. All the CCs can thus be executed in parallel, independently of each other. Grapefruit uses a pipelining methodology to cluster the automata into smaller groups, each group with several CCs (Figure 2 - *Automata Plane*). The original automata are first partitioned to multiple sets of automata (the number of automata in each set can be defined by the user), and each set is implemented in the same pipeline stage. Each incoming input symbol first is matched against all the CCs in the first pipeline stage and then travels to the next stage, being processed by the next stage while the first stage processes the second symbol.

Our pipelining structure reduces synthesis time compared to REAPR's flat design [15] because the synthesizer only considers the automata in the same stage, as they share the same input signal. On the other hand, using pipelining reduces the report vector size to the total number of reporting states in CCs of the same stage.

*2) Reporting architecture:* The reporting states fire a signal when a match happens. Collecting the reporting data for automata processing is another dimension that affects the area, power, delay, and throughput in a system. In Grapefruit, all the report states in the same pipeline stage create a bit-vector (each vector index is assigned to one report state). If any of the report states gets activated (meaning there is at least one report in that cycle), a snapshot of the report vector is sent

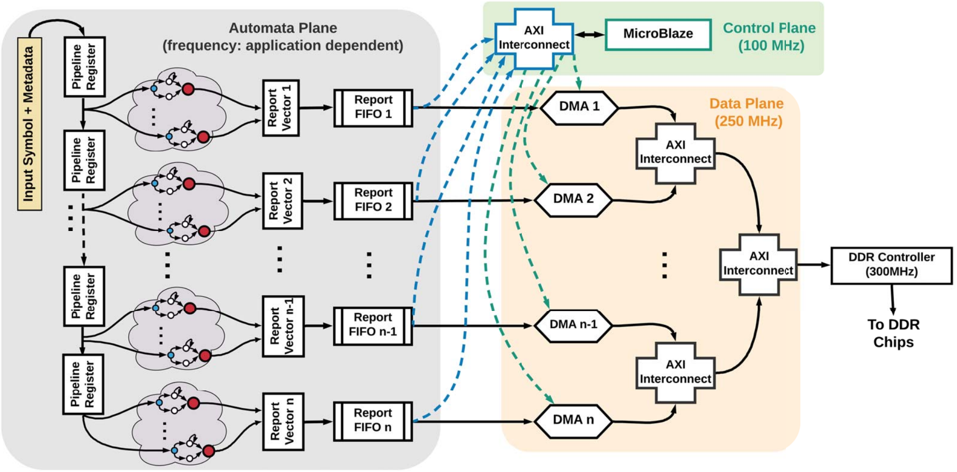

Fig. 2. Automata processing architecture on FPGAs. The cloud-shaped entities show several connected components.

to the report buffer allocated for each stage separately. Report buffers are flushed by the *Data Plane* interconnect toward the DDR memory. The *Data Plane* consists of a hierarchy of AXI-4 interconnects running at 250 MHz, which provides a path between every reporting buffer and the memory controller. Each report buffer is equipped with a DMA (Direct Memory Access) module to generate the memory address for storing the report data.

Current results are generated for a Xilinx VCU118 evaluation board, which has two DDR4 memory channels running in 1200 MHz. Grapefruit uses one memory channel for the report data and the other channel for the input symbols. Each memory channel has 5 DRAM chips, each with 16-bit data-width (80 bits in total). However, the Xilinx's IP core for memory controller only uses a 64-bit interface when ECC functionality is disabled. Therefore, in total, each memory channel can provide up to 18.75 GBps bandwidth. Using the hierarchical topology for the interconnect helps to avoid the signal congestion by distributing the complexity across multiple interconnect modules and providing a high-frequency yet area-efficient interconnect design.

In addition, our hierarchical design helps make the system scalable for larger applications, especially when the number of report buffers increases significantly. Grapefruit gives the user the flexibility to specify the desired Data Plane interconnect as an abstract tree data structure and generates the necessary scripts in the backend to implement it in an FPGA. As each of the automata pipeline stages has different report firing rate (which depends on the automata itself and the input characteristics), it is necessary to assign the Data Plane bandwidth wisely to the stages with a higher report rate (and possibly more filled reporting buffer). In addition, a static policy is not suitable as the input symbol pattern may change at run time, and this can lead to dynamic change in the reporting rate of different stages. To solve this issue, the *Control Plane* (shown in Figure 2) is used, where a MicroBlaze processor monitors the capacity of buffers to detect the highly reporting stages. The code running in the MicroBlaze reads the buffer size of each pipeline stage in a loop and configures the DMAs of buffers that running low in capacity to flush them into the off-chip DDR memory. The Control Plane uses AXI-Lite standard (no burst transmission) as transactions are simple register-reads/writes (buffer capacity and DMA control registers). In addition, Grapefruit utilizes the on-chip memory as the main memory for the MicroBlaze to make sure its instruction/data traffic does not interfere with the Data Plane memory operations.

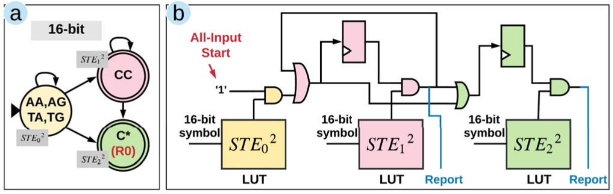

Fig. 3. Automata mapping in LUT-based design.

*3) Mapping Automata to LUT Resources:* This section discusses the mapping of an automaton (represented in *Automata Plane* in Figure2) to the LUT resource of the FPGA using an example. In Figure 3 (a), a homogeneous automaton is shown that processes two 8-bit symbols (16-bit) per cycle. $STE_0^2$ is the start state, and $STE_1^2$ and $STE_2^2$ are the reporting states. The states are color-coded to represent their equivalent units in the circuit shown in Figure 3 (b). Symbol matching is done entirely in LUTs based on the 16-bit symbols. Theoretically, FFs are equivalent to the states that may be activated in the following cycle. Once a state is activated, all of its children are considered as potential active states.

The input signals of the FFs come from an OR gate, which is the OR signal of all the states that have incoming transitions to that specific state. The states that have common parents can share their FFs and save resources. However, in theory, their corresponding states cannot be merged since they are not equivalent states. The report signals of the final states are generated from the AND gate of matching signals and potential active states.

*4) Mapping Automata to BRAM Resources:* In Addition to LUTs, FPGAs have fast and area-efficient on-chip memory resources such as BlockRAMs (BRAMs) and UltraRAMs (available in newer Xilinx FPGAs). An alternative to LUT-based design is to utilize the on-chip memories to calculate the symbol matching part efficiently. Technically, the symbol matching is responsible for detecting all the states that match against the current symbol, and BRAMs can implement this logic by simply putting each unique matching condition in one column and store '1' if the matching condition matches the cell location, assuming the input symbol is used as a read address at runtime; otherwise '0' is stored. In other words, each column is one STE. Because each pipeline stage is processing different symbols, it is not possible to combine matching rules from different stages into one BRAM. To minimize the BRAM waste in cases where the number of unique matching rules in a stage is not enough to fill all the available memory columns, Grapefruit configures the BRAMs in their narrowest configurations with smallest column-count and most-minor-row-size bigger than 256 (36 columns and 512 rows in Xilinx BRAMs).

Matching for higher stride values (processing multiple symbols per cycle) needs more bits to evaluate compared to the one symbol per cycle case. Xilinx BRAMs in their tallest configuration have 14 bits as the address input and one bit data output. This configuration is not even enough to implement the two symbols per cycle case (16-bit symbols). To solve this issue, Grapefruit combines multiple BRAMs (equivalent to the stride value) to implement multi-symbol conditions. Each BRAM is responsible for decoding 8 bits of the input symbol, and the final matching signal is calculated by applying a binary AND operation on the BRAMs output signals. For example, in Figure 3, the automaton on the left processes two symbols per cycle. Considering $STE_1$ as an example, it needs to compare the first and second 8-bit against 'C'. Two BRAMs (shown as two arrays in part b) are used for matching; the left array handles the first 8-bit, and the second array handles the second 8-bit. The character 'C' has been decoded in both these arrays in two separate columns (one column in each array), their output is ANDed, and its result is routed as the final match signal. In this design, columns with the same matching conditions can be merged to save the BRAM resources. For example, in Figure 3, both STE1 and STE2 compares the first 8 bits against 'C', so they can share the same column in the left array. However, the second 8-bit needs two separate columns, as their matching conditions are different ('C' versus *). Grapefruit looks for these common matching conditions in each array and tries to save BRAMs as much as possible by assigning the same column to equivalent matching considerations.

Addressing false positive issues for combined BRAMs:

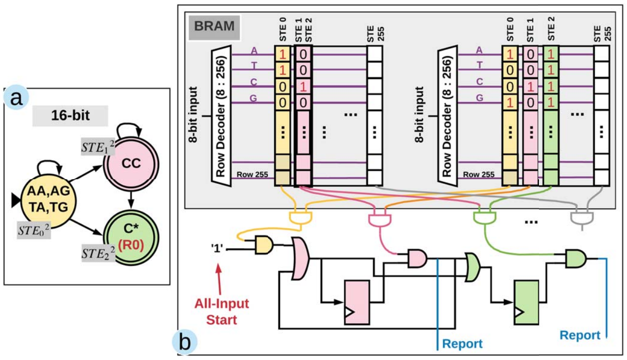

Fig. 4. Automaton mapping in BRAM-based design.

combining multiple BRAMs with an AND operation enables utilizing these on-chip memory resources for multi symbol matching, but it needs to be applied with careful consideration. Simply putting every 8-bit matching condition to one column of a BRAM array can potentially lead to a false-positive matching situation. For example, let us assume that one STE has a matching condition $AB, CD$ for an automaton that processes two symbols per cycle. Putting matching symbols $A, C$ of the first input symbol in one BRAM column and $B, D$ in another column of a separate BRAM can mistakenly match against AD *or* CB. Generally, every possible combination of matching symbols in each dimension can also be matched, as we generate the final matching symbol by an AND gate. To solve this issue, we need to detect states with possible false-positive matching conditions and refine the matching with simpler conditions that each can be implemented without regenerating this issue. In our previous example, if we split the matching conditions into two simpler cases, $AB$ and $CD$, we do not encounter this problem anymore. Grapefruit detects states with bogus matching conditions (if implemented directly in BRAM) automatically and refines them to minimum possible matching conditions and assigns each newly generated rule to a new state and removes the original state. Grapefruit repurposes the Espresso [47] logic minimizer to find the minimum number of refined rules set to implement every matching condition with BRAMs without generating false-positives. More details on this can be found in [48].

*5) Signal Sharing:* while LUTs are very flexible in implementing any matching condition (and generally in implementing every boolean function) without generating false positives, it is not the most efficient way to use them for pure LUT-based design. Our approach with BRAMs, to implement the matching condition partially and to combine them with an AND gate, can be emulated with LUTs as well. The benefit of this approach is that the middle 8-bit partial matching can share this signal with other states as well. For example, in Figure 5, part (a) shows an NFA that processes two 8-bit symbols per cycle. A pure LUT design without emulating BRAM design leads to the design in part (b). In this case, we rely only on synthesizer to reuse signals without any guide

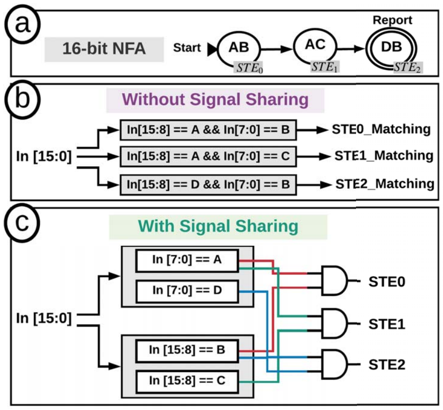

Fig. 5. An example on signal sharing.

via the generated HDL code. However, following the partial matching approach leads to the design in part (c). In this design, every partial matching condition is implemented in LUT, and each STE picks its required matching condition in different dimensions and combines them with an AND operation. As shown in part (c), STEs can share the partial matching conditions and save the total number of required LUTs. Design (c) needs to go through the false-positive checking procedure, as it is vulnerable to this issue, but it can be resolved using our matching refinement procedure explained for the pure BRAM design.

In summary, Grapefruit supports three matching architectures: pure BRAM, pure LUT, and LUT with signal-sharing. The user can select its preferred architecture at compilation time, and Grapefruit generates all the HDL codes necessary to implement it. In addition, Grapefruit can also implement a hybrid version of pure BRAM and LUT with signal-sharing by letting the user to freely put partial matching-conditions in BRAMs or LUTs.

## B. *Compiler Optimizations*

In the application layer, researchers want to have access to a tool that gives them a rich and descriptive interface to define an automaton and examine/debug them with an input stream. Our compiler, called APSim, uses a well-known and actively maintained python graph processing package, NetworkX, as its main building block to benefit from its reliability, speed, and extensive documentation with examples for further development by other collaborators. In APSim, an automaton is considered as a directed multigraph where the nodes or edges can carry symbol data and NFA metadata. All the automata related algorithms have been implemented on top of this concept. For example, a strided automaton that consumes two symbols per cycle can be achieved by finding every path of length two in the input automaton and replace it with a single edge in the strided automaton.

Stream-based transformation verification is also supported in APSim to check the correctness of transformations. Users can develop their own transformations and easily compare report status of the new automaton with the original automaton by streaming the same input to both of them and check the equivalence of report states status per cycle. This functionality also works with striding, where a multi-strided automaton can be compared against its original representation. APSim provides a comprehensive, parameterizable, and easy-to-modify tool. Figure 6 shows different components of APSim.

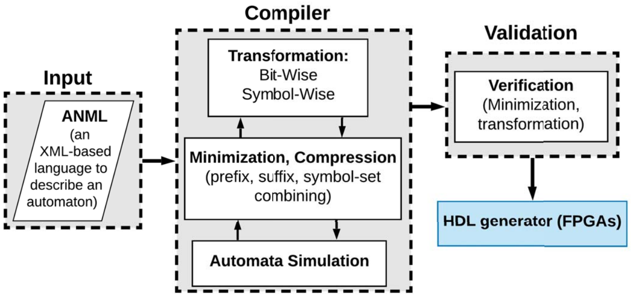

Fig. 6. Different components in our back-end compiler.

*1) Temporal Striding:* researchers are interested in using a tool that gives them enough flexibility to explore design space parameters comprehensively to discover high-throughput and efficient designs. One crucial design parameter in the architectural domain of automata processing is the symbol's bit-width (or alphabet size), which is mainly determined by the application. APSim provides temporal striding functionality, which is a set of transformations to change the bitwidth processing while keeping the automaton functionality intact.

Temporal Striding [41], [43] is a transformation that repeatedly squares the input alphabet of an input automaton and adjusts its matching symbols and transition graph accordingly. The transformed automaton is functionally equivalent to the original automaton, but it processes multiple symbols per cycle, thus increasing throughput. On the other side, the strided automata can cause state and transition overhead. The transformation overhead depends on the properties of automata, such as the in/out degree of states, the number of symbols, and the number of transitions. This creates an interesting performance vs. resource overhead tradeoff, and our framework supports and facilitates exploration of it and gives the hardware designer the chance to pick the sweet spot for a specific hardware platform. This sweet spot may vary significantly across different hardware domains such as in-memory processing, FPGA, or Von Neumann machines.

# V. Evaluation Methodology

NFA workloads: We evaluate our proposed claims using ANMLZoo [24] and Regex [49] benchmark suites. They represent a set of diverse applications, including machine learning, data mining, and network security. AutomataZoo [50] is mostly an extension of ANMLZoo (9 out of 13 applications are the same), and the difference is that ANMLZoo is normalized to fill one AP chip (with up to 48K states). To provide a fair comparison with prior work, we use ANMLZoo benchmarks because the prior work has used ANMLZoo in their evaluation, and their source code is not publicly available for evaluation with the AutomataZoo benchmark.

We present a summary of the applications in Table I, including the number of states and transitions in each benchmark as well as the average degree (the number of incoming and outgoing transitions) for each state. The higher the degree, the more challenging the benchmark is to map efficiently to the FPGA's underlying routing network.

TABLE I
BENCHMARK OVERVIEW

| **Benchmark** | **#Family** | **#States** | **#Transitions** | **Ave. Node Degree** |
|---|---|---|---|---|
| Brill [24] | Regex | 42658 | 62054 | 2.90 |
| Bro217 [49] | Regex | 2312 | 2130 | 1.84 |
| ClamAV [49] | Regex | 49538 | 49736 | 2.0 |
| Dotstar [49] | Regex | 96438 | 94254 | 1.95 |
| ExactMath [49] | Regex | 12439 | 12144 | 1.95 |
| PowerEN [24] | Regex | 40513 | 40271 | 1.98 |
| Protomata [24] | Regex | 42009 | 41635 | 1.98 |
| Ranges05 [49] | Regex | 12621 | 12472 | 1.97 |
| Snort [24] | Regex | 100500 | 81380 | 1.61 |
| TCP [49] | Regex | 19704 | 21164 | 2.14 |
| EntityResolution [24] | Widget | 95136 | 219264 | 4.60 |
| Fermi [24] | Widget | 40783 | 57576 | 2.82 |
| RandomForest [24] | Widget | 33220 | 33220 | 2 |
| SPM [24] | Widget | 69029 | 211050 | 6.11 |
| Hamming [24] | Mesh | 11346 | 19251 | 3.39 |
| Levenshtein [24] | Mesh | 2784 | 9096 | 6.53 |

Experimental setup: all the FPGA results are obtained on a Xilinx Virtex UltraScale+ XCVU9P with a PCIe Gen3 x16 interface, 75.9 Mb BRAM, and 1182k CLB LUTs in 16nm technology. The FPGA's host computer has an eight-core Intel i7-7820X CPU running at 3.6 GHz and 128 GB memory. Designs are synthesized with the Xilinx Vivado v2019.2.

For the interconnect structure, we follow an area-efficient tree structure for Data Plane and Control Plane. In the Data Plane, we construct the tree by starting from the report buffers and assign every five buffers to one AXI interconnect. Every five AXI interconnects are connected to the next level of AXIs. This iterative approach continues until we reach a single AXI interconnect and connect its master port to the DDR memory controller. The same procedure is applied for the Control Plane by letting ten nodes connected to the interconnects (instead of five), as the Data Plane is running at a slower frequency and a high-bandwidth design is not necessary. Register slicing and AXI datapath FIFOs are enabled for the Data Plane with the burst-size set to its maximum possible value (256), following Xilinx recommendations for high-throughput design. Bus-width in both Data Plane and Control Plane is set to 32 to reduce the routing congestion. All these parameters can be set to different values in compile time thanks to our rich compiler APIs. Moreover, we set the clock constraint to 250MHz to be satisfied by the synthesizer.

## VI. EXPERIMENTAL RESULTS

### A. LUT-based vs. DRAM-based design

Table II shows different statistics for LUT-based design and BRAM-based design. As expected, benchmarks with a higher number of states and transitions (shown in Table I have higher LUT, FF, and BRAM usage, and also higher power consumption. However, the frequency is a function of both automata size (number of state/transitions) and automata structure (such as fan in/out or average node degree). For example, EntityResolution has 95,136 states and SPM has 69,029 state. However, SPM frequency is 175.6 MHz while EntityResolution frequency is 228.8 MHz (30% higher frequency). This is because the average node degree in EntityResolution is 4.6, while the average node degree for SPM is 6.11.

BRAM-based design reaches a higher operational frequency for larger benchmarks, such as PowerEN, Protamata, and Snort, possibly due to the limitations of LUTs. LUTs in Xilinx FPGA's can implement a single 6-bit boolean function. To implement 8-bit functions, multiple LUTs need to be serially connected, which leads to a higher latency. However, BRAMs can implement 8-bit boolean functions with a single memory read operation. For smaller benchmarks, the sparsity of BRAMS across the chip and routing latency associated with it become a disadvantage, and the popularity of LUTs becomes a winning factor. The number of utilized BRAM18s in BRAM-based design is always higher than the number of utilized BRAM18s in LUT-based design because matching logic is implemented in BRAM18s blocks. However, BRAM36 remains the same for both designs as Grapefruit only uses the BRAM18 for matching, and BRAM36 is mainly used by the report interconnect. The number of LUTs in LUT-based design is always higher than BRAM-based design because these resources are utilized mostly for matching. However, a significant fraction of the LUTs are used for the interconnect logic. The number of FFs remains mostly the same as both designs have the same number of STEs.

### B. Striding and signal sharing effects

Table III represents various statistics for the TCP benchmark when applying the temporal striding technique to both BRAM-based and LUT-based designs, and when applying signal-sharing optimization to the LUT-based design.

As discussed, temporal striding increases the number of state and transitions, which translates to higher LUT, FF, and BRAM utilization and slightly higher power consumption. However, it significantly increases throughput as it processes multiple symbols in one cycle. In BRAM-based design, increasing processing rate from one symbol per cycle (8-bit processing) to four symbols per cycle (32-bit processing) results in 3× higher throughput at the expense of only 2.37× higher LUTs, 1.7× higher FFs, 2.5× BRAMs, and 1.26× higher power consumption. Similarly, In LUT-based design, increasing the processing rate from one symbol per cycle (8-bit processing) to four symbols per cycle (32-bit processing) results in 3.4× higher throughput at the expense of only 1.15× higher LUTs, 1.01× higher FFs, 1.58× BRAMs, and 1.01× higher power consumption. In summary, striding capability gives the user the ability to explore the area, power, delay, and throughput trade-offs and makes the best design choices based on the target objectives.

TABLE II
COMPARING LUT-BASED DESIGN VS BRAM-BASED DESIGN (IN SINGLE-STRIDE AUTOMATA OR 8-BIT PROCESSING).

| **Benchmark** | **LUT-based** | | | | | | **BRAM-based** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LUTs | FFs | BRAM18 | BRAM36 | Power (W) | Clock (MHz) | LUTs | FFs | BRAM18 | BRAM36 | Power (W) | Clock (MHz) |
| Brill | 144,273 | 205,162 | 82 | 503 | 6.9 | 231.3 | 130,876 | 203,385 | 160 | 503 | 7.1 | 258.1 |
| Bro217 | 31,442 | 45,552 | 9 | 72 | 4.2 | 291.0 | 30,413 | 45,086 | 20 | 72 | 4.2 | 264.8 |
| ClamAV | 90,747 | 113,685 | 23 | 149 | 5.7 | 259.9 | 89,919 | 115,729 | 96 | 149 | 5.8 | 245.9 |
| Dotstar | 247,999 | 324,571 | 116 | 710 | 9.7 | 237.6 | 222,694 | 321,527 | 292 | 710 | 9.5 | 240.6 |
| ExactMatch | 43,442 | 58,568 | 13 | 99 | 4.6 | 278.6 | 41,684 | 58,262 | 31 | 99 | 4.5 | 260.8 |
| PowerEN | 192,204 | 278,126 | 117 | 779 | 8.4 | 166.7 | 172,119 | 275,330 | 233 | 779 | 8.5 | 247.2 |
| Protomata | 171,941 | 230,871 | 95 | 591 | 7.6 | 170.7 | 149,861 | 228,257 | 367 | 591 | 8.0 | 249.0 |
| Ranges05 | 44,008 | 58,527 | 13 | 100 | 4.6 | 253.2 | 41,702 | 58,198 | 31 | 100 | 4.5 | 260.9 |
| Snort | 202,488 | 279,736 | 116 | 763 | 9.0 | 163.5 | 179,862 | 273,870 | 243 | 763 | 8.6 | 232.6 |
| TCP | 69,685 | 95,531 | 31 | 208 | 5.3 | 254.6 | 64,278 | 59,822 | 67 | 208 | 5.2 | 264.1 |
| EntityResolution | 122,452 | 148,907 | 41 | 268 | 6.1 | 228.8 | 108,406 | 148,611 | 81 | 268 | 7.2 | 248.9 |
| Fermi | 145,439 | 221,904 | 97 | 604 | 6.9 | 203.4 | 133,357 | 221,011 | 255 | 604 | 8.1 | 208.3 |
| RandomForest | 126,562 | 170,431 | 69 | 435 | 6.4 | 244.1 | 80,364 | 111,977 | 156 | 316 | 6.0 | 253.1 |
| SPM | 179,597 | 244,101 | 153 | 893 | 9.3 | 175.6 | 153,790 | 242,659 | 244 | 893 | 9.6 | 201.7 |
| Hamming | 29,066 | 40,703 | 7 | 60 | 4.2 | 293.3 | 27,848 | 40,611 | 15 | 60 | 4.3 | 271.7 |
| Levenshtein | 20,356 | 27,563 | 3 | 41 | 4.0 | 303.4 | 20,719 | 28,564 | 4 | 41 | 4.0 | 262.7 |

TABLE III
STRIDING AND SIGNAL-SHARING EFFECT FOR TCP BENCHMARK.

| **Design Parameter** | **Symbol Size** | LUTs | FFs | BRAM18 | BRAM36 | Power (W) | Clock (MHz) | Throughput (Gbps) |
|---|---|---|---|---|---|---|---|---|
| **BRAM-based** | 8-bit | 64,278 | 59,822 | 67 | 208 | 5.2 | 264.1 | 2.06 |
| | 16-bit | 64,083 | 96,862 | 115 | 296 | 5.7 | 244.5 | 3.82 |
| | 32-bit | 73,773 | 106,138 | 184 | 489 | 6.5 | 198.6 | 6.21 |
| **LUT-based** | 8-bit | 69,685 | 95,531 | 31 | 208 | 5.3 | 254.6 | 1.99 |
| | 16-bit | 105,150 | 96,604 | 33 | 362 | 5.3 | 225.5 | 3.52 |
| | 32-bit | 165,228 | 97,016 | 33 | 362 | 5.4 | 217.8 | 6.81 |
| **Signal-Sharing on LUT-based** | 8-bit | 65469 | 94900 | 31 | 208 | 4.8 | 245.6 | 1.92 |
| | 16-bit | 69,559 | 97,110 | 43 | 296 | 5.0 | 243.7 | 3.81 |
| | 32-bit | 78,412 | 106,265 | 37 | 489 | 5.3 | 223.1 | 6.97 |

The signal-sharing technique is applied to the LUT-based design. This optimization reduces the number of LUTs 1.5× in 16-bit design and 2.1× in 32-bit design. Moreover, signal sharing improves throughput and reduces power consumption.

### C. Comparison with prior work

We compare Grapefruit (in 8-bit processing) with REAPR+ [29]. This tool only generates the kernel and does not consider the I/O. The authors have evaluated their solution on ANMLZoo benchmarks using Xilinx Kintex-Ultrascale xcku060-ffva1156-2-e FPGA. Figure 7 shows that the Grapefruit full-stack solution performs 9%-80% better than kernel results in REAPR+ for the majority of the benchmarks. On average, Grapefruit has 3.8× more LUTs and 5.9× more FFs, and this is mainly because Grapefruit supports I/O.

REAPR+ performs better for Hamming and Levenshtein, and this is because these benchmarks are relatively small, and the designer can force a higher clock constraint. Our clock constraint is set to 250 MHz and increasing the clock constraint can significantly increase the frequency for smaller benchmarks, such as Hamming and Levenshtein.

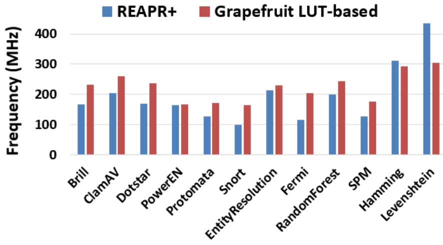

Fig. 7. Comparing Grapefruit (8-bit) with REAPR+ [29].

## VII. CONCLUSIONS AND FUTURE WORK

This paper presents *Grapefruit*, a publicly available framework for automata processing on FPGAs. Grapefruit consists of two main components: (1) an integrated compiler for automata simulation, minimization, transformation, and optimization, and (2) an HDL generator that produces a full-stack design for a set of automata to be processed on FPGAs. We develop an efficient reporting architecture and pipeline design, along with a set of hardware optimizations and parameters. Our framework allows researchers to investigate frequency and resource-usage trade-offs and provides an easy-to-understand and easy-to-modify code for them to explore new ideas. Grapefruit provides up to 80% higher frequency than prior works that are not fully end-to-end and 3.4× higher throughput in a multi-stride solution than a single-stride solution. An interesting direction for future work would be automatic learning-based parameter tuning using static and dynamic behavior of automata in an application.

## ACKNOWLEDGMENT

This work is funded, in part, by the NSF (CCF-1629450) and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA.

# REFERENCES

[1] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[2] C. Liu and J. Wu, "Fast deep packet inspection with a dual finite automata," *IEEE Transactions on Computers*, vol. 62, no. 2, 2013.

[3] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[4] E. Sadredini, R. Rahimi, K. Wang, and K. Skadron, "Frequent subtree mining on the automata processor: challenges and opportunities," in *International Conference on Supercomputing (ICS)*. ACM, 2017.

[5] K. Wang, E. Sadredini, and K. Skadron, "Sequential pattern mining with the micron automata processor," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016, pp. 135–144.

[6] K. Wang et al., "Association rule mining with the micron automata processor," in *IPDPS*. IEEE, 2015.

[7] K. Wang, E. Sadredini, and K. Skadron, "Hierarchical pattern mining with the micron automata processor," in *International Journal of Parallel Programming (IJPP)*. 2017.

[8] C. Bo, V. Dang, E. Sadredini, and K. Skadron, "Searching for potential gRNA off-target sites for CRISPR/Cas9 using automata processing across different platforms," in *24th International Symposium on High-Performance Computer Architecture*. IEEE, 2018.

[9] I. Roy and S. Aluru, "Discovering motifs in biological sequences using the micron automata processor," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 13, no. 1, pp. 99–111, 2016.

[10] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the automata processor," in *International Conference on High Performance Computing*. Springer, 2016, pp. 200–218.

[11] M. Putic, A. Varshneya, and M. R. Stan, "Hierarchical temporal memory on the automata processor," *IEEE Micro*, vol. 37, no. 1, pp. 52–59, 2017.

[12] E. Sadredini, D. Guo, C. Bo, R. Rahimi, K. Skadron, and H. Wang, "A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 665–674.

[13] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the micron automata processor," in *International Conference on Semantic Computing (ICSC)*. IEEE, 2015.

[14] C. Bo, K. Wang, J. J. Fox, and K. Skadron, "Entity resolution acceleration using the automata processor," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 311–318.

[15] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "REAPR: Reconfigurable engine for automata processing," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017, pp. 1–8.

[16] F. Yu, Z. Chen, Y. Diao, T. 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*. ACM, 2006, pp. 93–102.

[17] T. T. Hieu and N. T. Tran, "A memory efficient FPGA-based pattern matching engine for stateful nids," in *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*. IEEE, 2013, pp. 252–257.

[18] R. Karakchi, L. O. Richards, and J. D. Bakos, "A dynamically reconfigurable automata processor overlay," in *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE, 2017, pp. 1–8.

[19] X. Wang, *Techniques for efficient regular expression matching across hardware architectures*. University of Missouri-Columbia, 2014.

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

[21] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 2008, pp. 131–136.

[22] L. Vespa, N. Weng, and R. Ramaswamy, "MS-DFA: Multiple-stride pattern matching for scalable deep packet inspection," *The Computer Journal*, vol. 54, no. 2, pp. 285–303, 2010.

[23] V. Košar and J. Korenek, "Multi-stride NFA-split architecture for regular expression matching using FPGA," in *Proceedings of the 9th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, 2014.

[24] J. Wadden et al., "ANMLZoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *IISWC*. IEEE, 2016.

[25] K. Atasu, F. Doerfler, J. van Lunteren, and C. Hagleitner, "Hardware-accelerated regular expression matching with overlap handling on ibm poweren processor," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 1254–1265.

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

[27] C. Bo, V. Dang, T. Xie, J. Wadden, M. Stan, and K. Skadron, "Automata processing in reconfigurable architectures: In-the-cloud deployment, cross-platform evaluation, and fast symbol-only reconfiguration," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 2, p. 9, 2019.

[28] J. Wadden, K. Angstadt, and K. Skadron, "Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 749–761.

[29] M. Casias, K. Angstadt, T. Tracy II, K. Skadron, and W. Weimer, "Debugging support for pattern-matching languages and accelerators," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[30] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "FlexAmata: A universal and efficient adaption of applications to spatial automata processing accelerators," in *Submitted to 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[31] H. Liu, M. Ibrahim, O. Kayiran, S. Pai, and A. Jog, "Architectural support for efficient large-scale automata processing," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 908–920.

[32] V. M. Glushkov, "The abstract theory of automata," *Russian Mathematical Surveys*, 1961.

[33] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "eAP: A scalable and efficient in-memory accelerator for automata processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 87–99.

[34] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "infant: NFA pattern matching on gpgpu devices," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 20–26, 2010.

[35] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: a fast multi-pattern regex matcher for modern cpus," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 631–648.

[36] M. Becchi and P. Crowley, "A-DFA: A time-and space-efficient DFA compression algorithm for fast regular expression evaluation," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[37] Intel, https://github.com/01org/hyperscan.

[38] H. Liu, S. Pai, and A. Jog, "Why gpus are slow at executing NFAs and how to make them faster," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 251–265.

[39] M. Lenjani and M. R. Hashemi, "Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities," *IET Computers & Digital Techniques*, vol. 8, no. 1, pp. 30–48, 2014.

[40] M. Avalle, F. Risso, and R. Sisto, "Scalable algorithms for NFA multi-striding and NFA-based deep packet inspection on GPUs," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1704–1717, 2016.

[41] M. Becchi and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008, pp. 50–59.

[42] ——, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM, 2007, pp. 145–154.

[43] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *ACM SIGARCH computer architecture news*, vol. 34, no. 2, pp. 191–202, 2006.

[44] T. Liu, Y. Yang, Y. Liu, Y. Sun, and L. Guo, "An efficient regular expressions compression algorithm from a new perspective," in *2011 Proceedings IEEE INFOCOM*. IEEE, 2011, pp. 2129–2137.

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

[46] T. Xie, V. Dang, C. Bo, J. Wadden, M. Stan, and K. Skadron, "An end-to-end reconfigurable engine for automata processing," in *50th Conference on Government Microcircuit Applications and Critical Technology (GOMACTech)*, 2018.

[47] Wikipedia contributors, "Espresso heuristic logic minimizer — Wikipedia, the free encyclopedia," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer

[48] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching," in *Submitted to 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[49] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 79–89.

[50] J. Wadden et al., "AutomataZoo: A modern automata processing benchmark suite," in *IISWC*. IEEE, 2018.