Evaluating High Performance Pattern Matching on the Automata Processor

Indranil Roy, Ankit Srivastava, Matt Grimm, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru *Fellow, IEEE*

Abstract—In this paper, we study the acceleration of applications that identify all the occurrences of thousands of string-patterns in an input data-stream using the Automata Processor (AP). For this evaluation, we use two applications from two fields, namely, cybersecurity and bioinformatics. The first application, called Fast-SNAP, scans network data for 4312 *signatures* of intrusion derived from the popular open-source Snort database. Using the resources of a single AP-board, Fast-SNAP can scan for all these signatures at 1 Gbps. The second application, called PROTOMATA, looks for all the occurrences of 1309 *motifs* from the PROSITE database in protein sequences. PROTOMATA is up to 68 times faster than the state-of-the-art CPU implementation. As a comparison, we emulate the execution of the same NFAs by programming FPGAs using state-of-the-art techniques. We find that the performance derived by using the resources of a single AP-board, which houses 32 AP-chips, is comparable to that of the resources of five to six large FPGAs. The design techniques used in this paper are generic and may be applicable to the development of similar applications on the AP.

Index Terms—Finite automata, regular expressions, automata processor, FPGAs, intrusion detection, protein motifs.

1 Introduction

Acceleration of applications that find all the occurrences of thousands of patterns in an input data-stream presents significant challenges. Some of the primary challenges include (1) broadcasting the data-stream to all the processing units for concurrent execution of pattern matching operations, (2) executing nondeterministic finite automata (NFAs) without state-space explosion, and (3) scaling the solution to support the search for thousands of patterns at ever increasing streaming bandwidth.

Among the reported solutions, the ones which are GPU-based are the most generic and enjoy the highest clock-speeds over other accelerators. However, they struggle to handle execution divergence among the executing threads, i.e. if any of the threads makes a conditional jump which is different from the others, then all the executing threads are preempted, and a new set of threads is loaded. This is a serious impediment when the threads are programmed to search for different patterns [1]. On the other hand, solutions based on custom-made application-specific integrated circuits (ASICs) are either too specific or limited by the available memory bandwidth [2]. Solutions using ternary content-addressable memory (TCAM) have also been developed [3]; however, they lack in scalability.

Some of the best results thus far have been reported by the solutions exploiting the reconfigurability and parallelism of field-programmable gate array (FPGA). Through the concurrent execution of multiple NFAs in hardware, significant speedup is obtained without any state-space explosion. However, even the largest FPGAs cannot fit beyond a few hundred NFAs at a time. Therefore, large *rulesets* have to be partitioned and handled by multiple devices.

In this paper, we investigate the use of the Automata Processor (AP) [4], [5] which was specifically designed to accelerate such applications. The AP is a reconfigurable accelerator co-processor based on the multiple instruction single data (MISD) architecture. It can be programmed to execute numerous NFAs in parallel on a single data-stream. Owing to its specific programmability, it provides significant advantage over the FPGA-based solutions in terms of the number of NFAs that can be executed concurrently on a single board. The input patterns can be specified as regular expressions (regexes) using the Perl Compatible Regular Expression (PCRE) syntax [6], or as NFA using a proprietary language called Automata Network Markup Language (ANML, pronounced as "animal").

We have developed two applications as demonstrators. The first application is called Fast-SNAP (for Fast-SNort using AP), and it scans network data-streams for occurrences of *signatures* of intrusion derived from the Snort database [7]. The second application is called PROTOMATA (for PROTein autOMATA), and it inspects protein sequences for existing occurrences of protein *motifs* listed in the PROSITE database [8]. The automata developed for these applications illustrate simple design techniques to extract maximum performance benefits from the AP.

PROTOMATA is estimated to run up to 68 times faster than the best-known CPU-based counterpart [9]. Similarly, in contrast to the existing methods, Fast-SNAP is able to handle close to the whole Snort active ruleset. It is estimated that the Fast-SNAP application will support *Deep Packet Inspection* (DPI) of 4312 signatures of malicious traffic at 1 Gbps using the resources available on a single AP-board. These estimates are based on accurately known run-time features which are described in detail in this paper. For thoroughness and a meaningful comparison, we have developed a tool-chain for executing these NFAs in FPGAs and have included the corresponding results as well.

The rest of the paper is organized as follows. First, in Section 2, we describe the programming model and the run-time environment briefly. This is required to understand the design of the two applications, Fast-SNAP and PROTOMATA, as detailed in Section 3 and Section 4,

respectively. We discuss the methodology for comparing these applications with the state-of-the-art on FPGA and CPU in Section 5. Finally, the estimated speedup of the applications on production hardware vis-à-vis the state-of-the-art implementations are presented in Section 6.

2 Automata Processor

2.1 Overview

The *rules* to be executed on the AP are defined as regexes, using the PCRE syntax, or as NFAs, using ANML. These can be compiled into machine-loadable finite state machines (AP-FSMs) using an *AP-compiler*. Once compiled, a large number of AP-FSMs can be loaded into the processor and executed in parallel against a single *dataflow* streamed to the processor. If one or more rules are matched in any given clock-cycle, henceforth called a *symbol-cycle*, then the host CPU program is notified with a report for every match identifying the rule and the offset in the dataflow where the matches occurred.

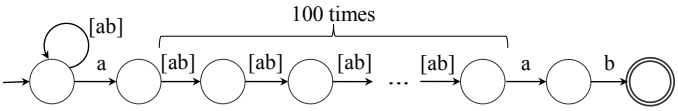
2.2 Automata Design

The PCRE syntax is well known to the programming community. However, ANML is proprietary to the AP. Therefore, we provide a brief description of ANML necessary to understand the automata designs presented in this paper.

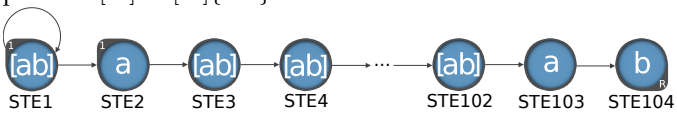
2.2.1 ANML Representation

The programmable elements in an AP-chip consist of processing elements called *State Transition Elements* (STEs), *Counter Elements*, and *Boolean Elements*; and a reconfigurable *routing network* to interconnect the processing elements. These elements and the routing network are configured to emulate NFAs. Defining an NFA using these native programmable elements is accomplished using ANML, and this representation is henceforth referred to as ANML-NFA. Fig. 1a shows the *state diagram* of a classical NFA which accepts any string from the alphabet $\{a, b\}$ that contains a substring of length 103 symbols whose first symbol is a and the last two symbols are a and b, respectively. The equivalent ANML-NFA is shown in Fig. 1b. *STE* 1 and *STE* 2 in the ANML-NFA are *start-STE*s, depicted by an indicator to the top-left of the circles. They correspond to the start-state in the classical NFA. Similarly, *reporting-STE*s are analogous to the accept-states in the state diagram. *STE* 104 represents a reporting-STE in Fig. 1b, distinguished by an indicator to the bottom-right of the circle with the letter R placed inside it. A detailed algorithm for converting any classical NFA to a corresponding ANML-NFA can be found in [10].

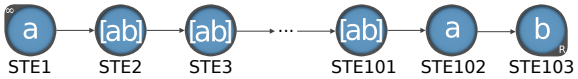
The matching of a dataflow by an ANML-NFA on the AP can be described as follows. In every symbol-cycle, an 8-bit symbol is broadcast from the dataflow. The processing begins from the start-STEs, all of which are *active* to process the first symbol. Each active STE processes the symbol relayed in that cycle, and activates the STEs connected to its outgoing routing lines on a match. If an element is programmed as *reporting*, then an *output-event* is generated on its match which identifies the corresponding element (and hence the rule it belongs to). The execution continues from the next symbol-cycle if there are more symbols in the dataflow and at least one STE is active.



(a) Classical state diagram of an NFA for accepting all the strings from the alphabet $\{a, b\}$ which satisfy the regular expression $[ab] * a[ab]\{100\}ab$.



(b) Equivalent ANML-NFA using *start-of-data* STEs.



(c) Equivalent ANML-NFA using an *all-input-start* STE.

Fig. 1: Representation of automata in ANML.

2.2.2 Special Features

All-input-start STEs: The start-STEs can be configured in two different modes, as a *start-of-data* STE or as an *all-input-start* STE. A start-of-data STE is active only during the first symbol-cycle of the dataflow, whereas an all-input-start STE is active during every symbol-cycle. The start-of-data STE is used when the occurrence of the pattern must be anchored to the beginning of the dataflow, whereas an all-input-start STE is used when the occurrence of the pattern can start at any offset in the dataflow. Therefore, the ANML-NFA in Fig. 1b can also be represented using an all-input-start STE as shown in Fig. 1c. Note that the start-of-data STE has the number 1 placed in the indicator to the top-left while the all-input-start STE has the symbol ∞ in the indicator instead.

Latched STEs: The STEs in the AP can be programmed to emulate a *latched-STE* which is represented by a ⌟ symbol inside the STE as shown in Fig. 2. A latched-STE activates the outgoing line on every symbol-cycle subsequent to the one in which the STE is matched for the first time. A use case of latched-STEs for generating compact ANML-NFA is discussed in Section 3.3.1.

Fig. 2: A *latched-STE* to identify the symbol '*a*'.

Counter Elements: The counter elements may be used to count the number of occurrences of a sub-pattern. They do not process symbols themselves and do not consume any symbol-cycles. For example, the ANML-NFA shown in Fig. 1c can be defined as the automaton shown in Fig. 3 by using a counter element to count the number of occurrences of $[ab]$. The counter element has two input lines shown as pentagons along the left boundary, and a single output line on the right boundary. The first input line, called the *count-*

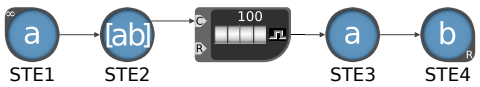


Fig. 3: ANML-NFA using a counter element with a *pulsed-output* equivalent to the regular expression $. * a[ab]\{100\}ab$.

line, is denoted by the letter *C*, whereas the other input line is called the *reset-line* and is denoted by the letter *R*. The element also has a programmable 12-bit *target-value* shown inside the element.

At the beginning of the first symbol-cycle, the *counter-value* is set to 0. It can be incremented by 1 by activating the count-line, or reset to 0 by activating the reset-line. If the counter-value reaches the target-value, the element activates its output-line. If the counter element is programmed to generate a *pulsed-output*, denoted by the ⎍ symbol, then the outgoing-line is activated only for the next symbol-cycle. However, if it is programmed to generate a *latched-output*, denoted by the ⌜ symbol, then the output line is activated until either the element is reset or the end of the dataflow is encountered.

Boolean Elements: The AP-chip contains several 16-input boolean elements which can perform the boolean operations *OR*, *AND*, *NOR*, *NAND*, *sum-of-product*, and *product-of-sum*. Similar to the counter elements, the boolean elements do not process any symbols or consume any symbol-cycles. If the input lines are simultaneously activated within a symbol-cycle in a manner such that the boolean operation is satisfied, then the output line of the boolean element is activated. An example of utilizing these boolean elements to develop compact ANML-NFA is described in Section 3.3.1.

2.2.3 Programming Environment

The AP-board is accompanied with an AP-SDK which enables users to define, compile, debug, load, and execute rules.

Design Environment: The ANML-NFA may be defined programmatically, or graphically using a *workbench*. The AP-compiler can then be used to compile these designs into loadable AP-FSMs. If the compiler is presented with PCRE patterns, it converts them into equivalent ANML-NFAs internally before creating the AP-FSMs. The generation of the AP-FSMs consists of mapping user-defined automata elements to the physical resources on the AP-chip. This requires the completion of complex *place-and-route* algorithms which may consume a significant amount of time. Therefore, whenever possible, the automata should be compiled beforehand. For the applications discussed in this paper, the rules are known a priori and hence the automata can be defined and compiled in advance.

Debug Environment: Although the AP-SDK does not provide a cycle-level simulator, the execution of AP-FSMs can be simulated using an *AP-emulator* against test dataflows. The AP-emulator may also be used to emulate the execution of native ANML-NFAs. Alternatively, the workbench can be used for visualizing the execution of ANML-NFAs against small dataflows.

Run-time Environment: The AP-SDK provides API calls to load the AP-FSMs, stream dataflows, and handle the output at run-time. After the compilation is complete, the host application uses the provided API calls to load the AP-FSMs into the AP-chip. This process is significantly faster as compared to compilation; an entire AP-board can be loaded with new AP-FSMs in about 50 milliseconds. Once the AP is programmed, the host application uses the API to stream the input dataflow and receive the processed output. In the AP-chip, one 8-bit symbol from the dataflow is broadcast to all the STEs in every symbol-cycle. A single AP-chip can process 133 million symbols per second, thus achieving a processing rate of 1 Gbps. However, if the loaded automata contains cascaded boolean and/or counter elements, the processing rate gets reduced by an integer factor called the *clock-divisor*, which is same as the maximum number of these elements connected in succession.

Output Handling: The reporting-STEs in an AP-chip are organized into 6 output regions. If an output-event is generated in an output region during a cycle, then an *output-vector* for that region is stored into an *output-buffer*. The length of this vector is independent of the number of elements which reported in that cycle. If the buffer has space for the incoming vectors, then they can be stored within the same symbol-cycle and the input processing continues unabated from the next cycle. Asynchronously, the run-time environment reads the output-vector from the output-buffer to the main memory, processes the same, and presents the host application with the ids of the reporting elements and the corresponding offsets in the dataflow. The elements can also be programmed to return a custom *report code* on match.

On the other hand, if the output-buffer is full, then the entire processing pipeline is stalled until it can be sufficiently emptied. Reading out vectors from the buffer takes $16 + 40p$ cycles, where 16 is the initial set-up latency for the transfer, 40 is the number of cycles required to transfer each output-vector, and p is the number of output-vectors generated in that cycle. Depending on the value of p, this can take between 56 and 256 cycles. Notice that, if no output-vectors are generated in a cycle ($p = 0$), then there is no output handling overhead.

2.2.4 ANML Macro

If multiple ANML-NFAs use a common sub-automaton where the instances differ only in the labels of the STEs, then such a sub-automaton can be defined as a *macro*. Not only do these macros help in creating building blocks for larger ANML-NFAs (or macros), but also reduce the compile time of larger automata. Once compiled, the macros obtain a partial mapping of the programmable elements and the routing lines to the logical entities in the AP-chip. Therefore, compiling automata with these constituent macros does not incur this overhead.

The labels of the STEs which vary across the instances are declared as the parameters of the macro and can be defined at run-time, just before loading, with negligible overhead. The STEs in the macros can be marked as start and/or reporting-STEs, and the processing elements inside the macro can be activated by the processing elements outside the macro through *input ports* and can also activate the outer elements using *output ports*. Additionally, macros may be nested inside other macros.

2.2.5 Programming Resources

The AP-board interfaces with the CPU using a high-speed PCIe interconnect and houses 32 AP-chips. Each AP-chip contains 49 152 STEs, 768 counter elements, and 2 304 boolean elements. Thus, cumulatively, a single AP-board contains 1 572 864 STEs, 24 576 counter elements, and 76 896 boolean elements. This is sufficient to accommodate NFAs

corresponding to a very large ruleset. For example, all the rules in the Snort and PROSITE databases can be programmed into 16 and 1 AP-chips, respectively.

The chips on an AP-board are arranged into four ranks of eight chips each. They can organized into *logical-cores* containing 1, 2, 4, or 8 chips in each rank, where all the chips within a logical-core operate on a common dataflow. By using a high-speed intra-rank bus, all the logical-cores in a rank can be presented with separate dataflows in parallel. Therefore, the chips on the AP-board can be configured to provide an overall processing rate between 1 Gbps (if a single logical-core includes all the chips on the board) to 32 Gbps (eight logical-cores per rank for a total throughput of 8 Gbps per rank $\times$ 4 ranks on the board). The size of the logical-cores is chosen to derive maximum data parallelization and hence performance from the AP-board.

3 FAST-SNAP

Fast-SNAP is a network intrusion detection (NID) tool which scans for *signatures* of intrusion detection in network data using the *rules* described in the widely used Snort database [7]. In this section, we discuss the state-of-the-art in Snort-based NID systems, followed by a brief introduction to the Snort rules, and finally a detailed description of the Fast-SNAP application.

3.1 Background

Snort [11] describes signatures of anomalous activity in network data as string patterns. The signatures are maintained in a database in the form of *rules*, one for every known signature. This ruleset is updated as and when new signatures are discovered. Currently, the ruleset contains 5 310 active pattern matching rules, which presents a significant computational challenge for contemporary bandwidth requirements and frequency of cyber-attacks. Therefore, accelerated solutions using GPUs and FPGAs have received considerable attention in literature.

Cascarano et al. [12] proposed the first NFA-based solution using GPUs, which involved maintaining a global NFA transition table along with vectors for active and future states. They reported a maximum throughput of about 1.5 Gbps for *Snort534* ruleset from [13], which contains 534 regexes. Zu et al. [1] noted that this approach suffered from the problem of serialization of threads because of execution divergence. They tried to address the issue by identifying *compatible states*, i.e. states that cannot be active simultaneously, and then using these compatible states to create *virtual-NFAs* from the original NFAs. Using the virtual-NFAs, they reported a maximum throughput of nearly 13 Gbps on small datasets consisting of 16 to 36 patterns.

The FPGA-based solutions rely on configuring the processor to concurrently execute multiple NFAs in hardware [14], [15], [16]. Yang et al. [14] used a modified version of McNaughton-Yamada algorithm to convert PCRE-based regexes to modular NFAs with multi-character transition labels. This allowed them to reach a maximum throughput of 10.3 Gbps. Mitra et al. [15] reported an interface throughput of 12.9 Gbps on the *SGI RASC RC 100* blade connected to *SGI ALTIX* supercomputing system, by transforming PCRE op-codes generated using the Snort rules compiler to *VHDL* code. However, the capacity of even the largest FPGAs is not enough to accommodate large rulesets [14], [16], requiring them to be partitioned across multiple FPGA devices.

RegX [17] is a regular expression matching engine which uses compressed DFAs and a variant of XFAs [18] as underlying automata. They reported a throughput of 45 Gbps for up to 600 synthetic patterns. RegX throughput, however, is sensitive to the complexity of the patterns and it drops below 10 Gbps for datasets including more than 5 000 complex patterns, i.e. patterns including wildcard repetitions and counting constraints. Fang et al. [19] proposed a special purpose automata processing architecture, called Unified Automata Processor, that can achieve throughputs up to 295 Gbps on datasets consisting of thousands of regexes.

3.2 Snort Rules

The Snort rules are written in a lightweight description language. Each rule contains a *header* section and an *options* section. The header section of a rule is written first and specifies the protocol, the source, and the destination of a network packet for which the rule is active and the type of action to be taken if the rule is matched. The options section of a rule is written next, enclosed in parentheses, and consists of one or more *keywords*, some of which may accept a value. A comprehensive description of the rules format can be found in the Snort user manual.[1]

```
alert tcp any any -> any 80 ( sid:42;
content:"foo"; content:"bar"; distance:10;
pcre:"/foo[0-9]{10}bar"; content:"kludge";
http_header; content:"cluft"; http_header;
content:"baz"; http_header; content:"qux";
http_header; content:"abc"; http_uri; )
```

Consider the dummy sample rule defined above. The first keyword in the options section of the rule specifies that the *sid*, a unique integer identifier of the rule, is 42. The following keywords are of the *payload detection* type, which are used for defining most of the patterns to be matched in the data section of a network packet. These keywords either specify strings to be matched exactly (using the keyword *content*) or PCRE (using the keyword *pcre*). The sample rule contains multiple exact strings to be matched (e.g., `foo`, `bar`, etc.) and one PCRE (namely `/foo[0-9]{10}bar`). Matching of the patterns can be constrained by the *modifier* keywords. A modifier keyword is associated with the previous pattern in the rule. There are two types of modifier keywords: *distance-modifiers*, which place distance constraints between the occurrence of different patterns, and *location-modifiers*, which restrict the search for the pattern to a specific part of the payload. These are described in further detail in Section 3.3.1. A rule is said to be *triggered* if all the patterns defined in the rule are matched, along with the constraints specified by the modifiers.

3.3 Methodology

3.3.1 Automata Design

81% of the active pattern matching rules in Snort can be efficiently implemented using the AP. 17% of the rules

1. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node27.html

contain the keywords *byte_test*, *byte_jump*, or *byte_extract*, which require the use of *capture groups*. These rules cannot be represented as *regular languages*, and therefore by NFA. The remaining 2% of the rules can not be implemented using the AP because of multiple known issues with the AP-compiler.[2] Checking for the rules which can not be implemented efficiently using the AP is carried out using the Snort software [7] running on the CPU.

For each rule defined in the Snort ruleset, we create the corresponding ANML-NFA in six steps. These steps are described below, using the sample rule defined in Section 3.2.

Step 1. Handling Location-Modifiers: Searching for the occurrences of a pattern can be restricted to a particular section of the payload through the use of the location-modifiers. Additionally, the location-modifiers can specify whether the pattern should be matched in raw data or normalized data. For example, using the keyword *http_uri*, the sample rule restricts the search for the pattern `abc` to the normalized request URI section of the data.

In the first step, separate *buckets* are created corresponding to each location-modifier defined in Snort. Then, for each rule, the patterns qualified by the different location-modifiers are placed in the respective buckets, along with the *sid* of the rule. This allows us to program patterns from different buckets into separate logical-cores and stream only the data relevant to the location-modifier to the corresponding logical-core. For our sample rule, the patterns `foo`, `bar`, and `/foo[0-9]{10}bar` are placed in the *general* bucket; `kludge`, `cluft`, `baz`, and `qux` in the *http_header* bucket; and `abc` in the *http_uri* bucket.

Step 2. Handling Distance-Modifiers: The distance-modifiers specify constraints on the location of the occurrence of a pattern in the dataflow relative to an anchor. The anchor could either be the beginning of the dataflow or the end of the occurrence of the previous pattern. For example, our sample rule contains the modifier *distance* with an argument of 10. This modifier mandates that the occurrences of the two preceding patterns (namely `foo` and `bar`) are separated by at least 10 characters.

In the second step, the patterns within each bucket are considered separately. The patterns related through distance-modifiers, belonging to the same rule, are combined into a single PCRE using repetition quantifiers. In our sample rule, the patterns `foo` and `bar` in the *general* bucket are combined to get the PCRE: `foo.{10,}bar`.

Step 3. Handling Long Repetition Quantifiers: PCRE repetition quantifiers are used to specify the number of times a sub-pattern should match in the string. The size of such repetitions can be as high as 13 280 in some of the patterns in the Snort ruleset. The counter elements can be used for handling these repetitions only if it can be guaranteed that a second count will not be started while a count is in progress, a condition that is not satisfied by most of the repetitions. The repetitions are therefore handled using the STEs, thereby consuming a lot of resources.

In the third step, repetition quantifiers of size more than 512 in the patterns are substituted by unbounded repetitions. These constitute less than 1% of the supported rules. The language of the resulting pattern, after substitutions, is a superset of the language of the original pattern. Therefore, the *sid* of the corresponding rule and the original pattern is recorded for eliminating false positive matches during the execution stage.

Step 4. Handling PCRE Back References: In a PCRE, back references are used for matching the same string as the one matched by a previous sub-pattern [6]. 6% of the supported rules contain patterns with back references. However, since the match depends on the input dataflow, the AP cannot handle back references efficiently.

In the fourth step, all the back references in the patterns are substituted by the corresponding sub-patterns being referred to. As in the third step, the substituted pattern can report false positive matches. Therefore, the *sid*s of the rules and the original patterns are noted for verifying the matches reported by the AP.

Step 5. Generating Final ANML-NFA: After combining patterns into PCRE in the second step, multiple PCREs for a rule may be left in a bucket. All such PCREs should match in the dataflow, in any order, for a rule to be triggered.

In the fifth step, the PCRE(s) for every rule in all the buckets are converted into an ANML-NFA, with its report code set to the *sid* of the rule. This is trivial if there is a single PCRE corresponding to the rule in the bucket. However, if there are multiple PCREs for a rule, then a boolean *AND* element and latched-STEs are used. For example, there are four patterns in the *http_header* bucket: `kludge`, `cluft`, `baz`, and `qux`. The ANML-NFA for the same is shown in Fig. 4. Notice that, all the PCREs pertaining to a single rule from the same bucket can also be combined to form a final unified PCRE using *lookahead assertions* [6]. For example, the patterns in the *http_header* bucket can be expressed as a PCRE with three lookaheads: `(?=.*kludge)(?=.*cluft)(?=.*baz).*(?:qux).` However, the ANML-NFA corresponding to PCRE with lookahead assertions is very large. Instead, the ANML-NFA described above is considerably smaller.

Step 6. Handling Clock-Divisor: The streaming rate for a bucket is determined by the ANML-NFA with the highest clock-divisor (discussed in Section 2.2.3) in the bucket.

In the sixth and final step, the ANML-NFAs within the same bucket are separated according to their clock-divisors. The clock-divisor for the ANML-NFA generated in the fifth step is determined. If the divisor is found to be greater than 1, then a separate bucket for the location-modifier is created corresponding to the divisor, if one doesn't already exist. The ANML-NFA is then added to the bucket corresponding

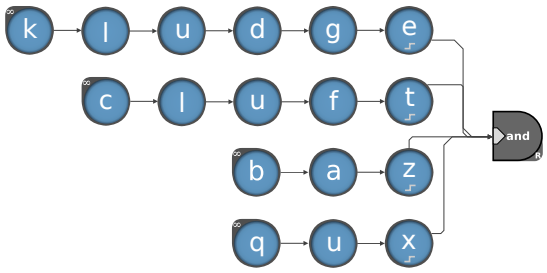


Fig. 4: Final ANML-NFA corresponding to the *http_header* bucket of the sample rule.

2. http://www.micronautomata.com/apsdk_documentation/latest/h1_known_issues.html#h2_pcre_known_issues

to the divisor.

At the end of these steps, there is at most one ANML-NFA per rule in every bucket. The automata from each bucket is compiled into a single AP-FSM. Compiling ANML-NFAs for all the patterns in a bucket together results in more efficient resource utilization and placement. However, the time required for compiling complete buckets might be prohibitive in scenarios when a rule corresponding to a newly detected threat is to be added quickly. In such cases, the new patterns can be compiled separately and loaded with the corresponding buckets, while the compilation of the modified buckets runs in the background.

3.3.2 Execution Stage

The compiled AP-FSMs are directly loaded into the AP-board and the processing of the network packets begins instantaneously. The host application breaks the incoming network packets corresponding to the different buckets and generates the dataflow for each logical-core on the AP-board. The *sid* of a rule is reported whenever the corresponding automata detects a match in a network packet. If the reported *sid* corresponds to one or more patterns with long repetition quantifiers or back references then the packet is matched against the original patterns, recorded in *Step 3* or *Step 4* of the configuration stage, using the Snort software [7]. In case a rule is programmed as multiple automata, pertaining to patterns with different location-modifiers, the host application triggers the necessary actions specified in the original Snort rule only if all the constituent automata generate a report within the same network packet. In this way, a very high throughput DPI engine is realized.

4 PROTOMATA

PROTOMATA scans protein sequences for occurrences of the protein *motifs* from the PROSITE database [8]. In this section, we briefly describe the PROSITE pattern-motifs, the state-of-the-art in finding these motifs in protein sequences, and the PROTOMATA application in detail.

4.1 Background

PROSITE is a large annotated database of known protein motifs. A motif is described as a small conserved region in a protein sequence which plays a biologically meaningful role in the behavior of the protein. The motifs are described as either *pattern-motifs* or *profile-motifs*. A pattern-motif is expressed using *PROSITE pattern notation* described in Section 4.2. On the other hand, the profile-motifs use a weight-matrix-based method to calculate similarity. PROTOMATA only scans for occurrences of the pattern-motifs, and henceforth in this paper, 'pattern-motifs' and 'motifs' are used interchangeably. Currently, PROSITE has 1 309 motifs of which 12 are classified as *frequently occurring*.

Multiple tools to scan protein sequences for occurrences of the motifs were developed by academic groups [20], [21], [22], [23] and commercial companies [24] in the early 1990s. ScanProsite [25], an online tool provided by PROSITE, is the current de facto standard for the purpose. The tool supports the following three modes of operations: (1) protein sequences can be submitted to be scanned against all the motifs in the PROSITE database, (2) motifs can be submitted to be scanned against a protein database (UniProtKB [26], PDB, or user-defined), and (3) motifs and protein sequences can be submitted to be scanned against each other. A Perl-based version of the tool, called *ps_scan*, can be downloaded for execution on a local machine. However, in our experiments, we found that *ps_scan* is orders of magnitude slower than the state-of-the-art PCRE engines. Therefore, for the comparative studies, we used Hyperscan (discussed in Section 5.2).

4.2 PROSITE Motifs

The PROSITE motifs are expressed in *PROSITE pattern notation*. Each motif consists of a sequence of *pattern elements* separated by a concatenation symbol '$-$'. The complete syntax of these pattern elements is available online.[3] However, we briefly describe the pattern elements used in the PROSITE motif *PS00430* (*TonB-dependent receptor proteins signature 1*):
$< x(10,115) - [DENF] - [ST] - [LIVMF] - [LIVSTEQ] - V - \{AGPN\} - [AGP] - [STANEQPK]$

In the motif, '$<$' denotes the beginning of the sequence, 'x' denotes any amino acid, '$(10,115)$' denotes a repetition between 10 and 115 times, '$[\ldots]$' denotes a character class for matching any one of the amino acids in the list, and '$\{\ldots\}$' denotes a *complementary class*, i.e. any amino acid but the ones listed within the curly braces. Fig. 5 shows the occurrence of the PROSITE motif *PS00430* in the UniProtKB protein sequence *D3BUN1 (Lissencephaly-1 homolog)*. Next, we use this motif to illustrate our automata designs.

4.3 Methodology

4.3.1 Automata Design

In order to generate the required automata, the database of motifs is downloaded periodically (or on user request)

3. https://prosite.expasy.org/scanprosite/scanprosite_doc.html

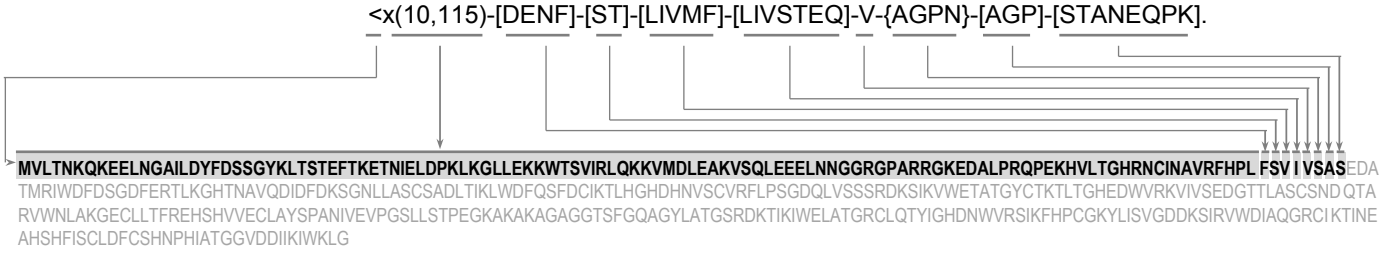


Fig. 5: Occurrence of the PROSITE motif *PS00430* in the *Lissencephaly-1 homolog* protein sequence.

and all the pattern-motifs are extracted from the same. For each motif in the database, two pre-compilable automata are required. One for reporting the location of each occurrence of the motif and the other for checking if the motif occurs in all of the multiple protein sequences provided by the user. We use the *Selective-enable* automaton and the *All-repeat-check* automaton for this purpose as described next. The reader is referred to [10] for details on the design and working of these automata.

Two pattern macros called the *Report-on-match* macro and the *Continue-on-match* macro are created for every motif. Further, each motif is assigned a unique 2-byte *PROTOMATA-id*. This id is different from the alphanumeric *PROSITE-id*, which can contain between 2 and 21 characters, and a simple one-to-one mapping is maintained on the host application to provide the necessary interface between the user input, using the PROSITE-ids, and the working of PROTOMATA, using the PROTOMATA-ids. The Report-on-match macro and the PROSITE-id is then used to instantiate the Selective-enable automaton shown in Fig. 6. The Continue-on-match macro is used in the All-repeat-check automaton shown in Fig. 7. Both these automata correspond to the motif *PS00430*, which is assigned a PROTOMATA-id of $018a_{16}$. For simplicity of expression, the STE labels in the figures are represented as follows:

- An upper-case letter represents an amino acid. An STE-label with one or more upper-case letters represents the character class containing the ASCII equivalent of the letters in the upper and lower cases. For example, the label A represents the character class $[41_{16}, 61_{16}]$ and the label AGV represents the character class $[41_{16}, 47_{16}, 56_{16}, 61_{16}, 67_{16}, 76_{16}]$.
- 'Σ' represents the character class containing the ASCII equivalent of all the letters representing the amino acids in upper and lower cases.
- '•' represents the entire 8-bit symbol-set.
- All the other labels are 2-digit hexadecimal numbers.

A *preamble sequence* is streamed at the beginning of the dataflow which contains both the bytes of the PROTOMATA-ids of only those motifs whose search is to be enabled. For example, if the search for frequently occurring motifs is to be disabled, then the PROTOMATA-ids of only the rest of the motifs are included in the preamble sequence. The preamble sequences for standard choices are pre-computed and stored. The end of the preamble sequence is earmarked by the character ff_{16}. In order to avoid any confusion between this end-marker and the first byte of a PROTOMATA-id in the preamble sequence, the range of the latter is restricted from 00_{16} to fe_{16}, i.e. the PROTOMATA-ids lie between 0000_{16} and $feff_{16}$.

Enable macro: The *Enable* macro is part of both the Selective-enable and the All-repeat-check automata. It processes the preamble sequence and activates the subsequent processing elements only if the PROTOMATA-id of the associated motif is present in the preamble sequence. This obviates the need to reprogram the AP-chip every time the user enables the search for a different set of motifs. If the occurrence of the motif is not anchored to the beginning of the protein sequence, then the output port *o2* is also connected to the input port *i2* of the pattern macro. The output port *o2* is driven by *STE* 6 which creates an activation signal for every symbol in the first protein sequence.

Selective-enable automaton: The Selective-enable automaton creates an output report as soon as the occurrence of a motif is detected in the dataflow, thereby marking the location of the occurrence. If the search for *PS00430* is enabled, then *STE* 9 of the Report-on-match macro is activated to process the first symbol of the first protein sequence. The connections between *STE* 9, *CTR* 1, and *CTR* 2 ensure that the output of *CTR* 1 is activated only for the 11^{th} to the 116^{th} symbol in the protein sequence. The matching of the sequence using *STE*s $11 - 18$ is straight-forward. *STE* 18 creates an output as soon as the occurrence of the motif is found in the protein sequence. The protein sequences to be scanned are also separated by the character ff_{16}. *STE* 10 is used to restart the processing for the next protein sequence by reactivating *STE* 9 and resetting the counter elements on encountering ff_{16}.

All-repeat-check automaton: The All-repeat-check automaton is used when the motifs present in all the input protein sequences need to be identified. It minimizes the rate of output generation and generates a single report at the end of the dataflow identifying only the common motifs as follows. The Report-on-match macro in Fig. 6 is replaced by the Continue-on-match macro in Fig. 7 which does not have any reporting-STEs. Instead, the last STE, namely *STE* 18, is connected to the input ports of the *Serial-repeat* macro. If the occurrence of the motif needs to be anchored to the end of the sequence, the connection to the port *i3* is excluded. Only if the motif occurs in the first protein sequence, *STE* 19 and *STE* 20 are activated. *STE* 19 keeps *STE* 20 activated till the end-delimiter of the sequence is encountered. On encountering ff_{16}, *STE* 20 reactivates

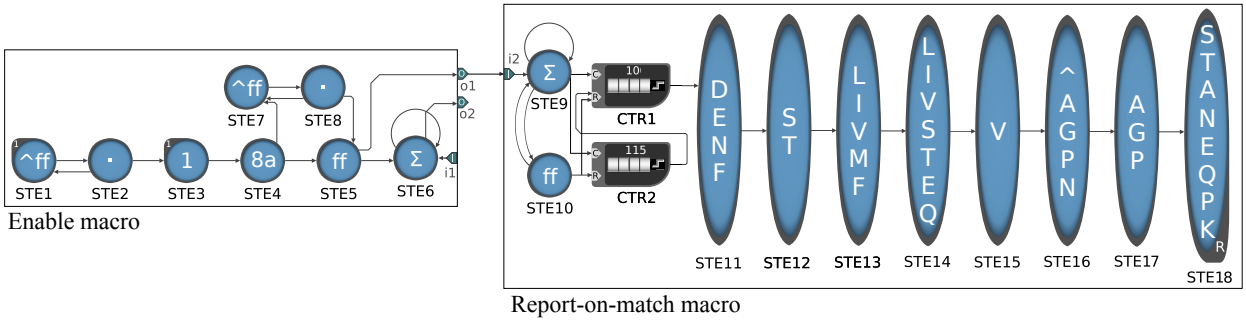


Fig. 6: The *Selective-enable* automaton for the PROSITE motif *PS00430*.

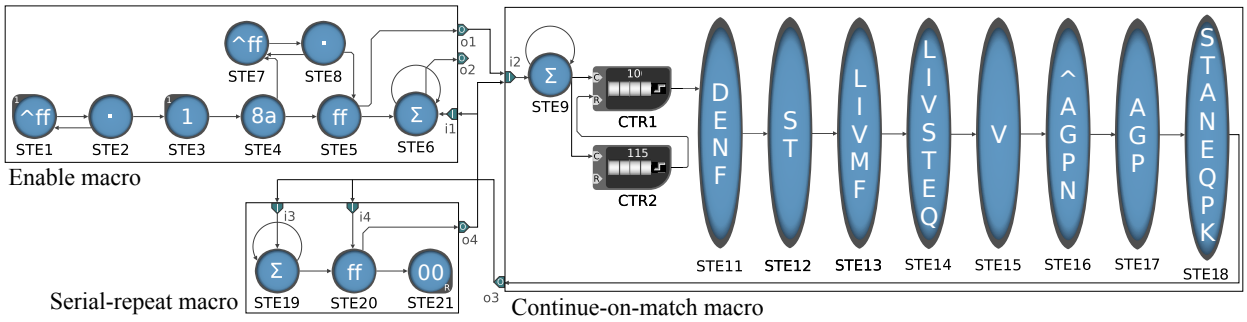


Fig. 7: The *All-repeat-check* automaton for the PROSITE motif *PS00430*.

the first STE of the Continue-on-match macro to enable the search for this motif in the next sequence. Simultaneously, *STE* 6 of the Enable macro is also activated to handle motifs whose occurrences need not be attached to the beginning of the protein sequence. If *STE* 20 is active after the last sequence has streamed, then it activates *STE* 21 to generate a report on encountering 00_{16} at the end of the dataflow.

4.3.2 Execution Stage

The Selective-enable and the All-repeat-check automata for all the motifs are compiled together into two separate AP-FSMs and loaded into the board. The dataflow is constructed based on the user input, from the preamble sequence and the proteins to be scanned. If the location of each occurrence of every enabled motif is to be reported, then the assembled dataflow is streamed to the AP-FSM corresponding to all the Selective-enable automata. On the other hand, if the motifs that occur in all the sequences need to be reported, then the All-repeat-check AP-FSM is used instead. In this case, if the location of the common motif in the individual protein sequences is also required, then a second pass is made using the Selective-enable AP-FSM with only the motifs that reported in the first pass.

5 Comparative Evaluation

5.1 Comparison with FPGA

Nourian et al. [27] compared the AP with FPGA for automata processing in multiple representative applications and found that processing throughputs are generally higher for FPGA at the cost of high preprocessing times. In this paper, we compare the performance of the AP with FPGA for FastSNAP and PROTOMATA. For a fair comparison, we implemented a tool-chain that takes an ANML-NFA as input and automatically generates *Verilog* code for implementing it. Then, we used *Xilinx ISE Design Suite 10.1* for performing synthesis, mapping, and place-and-route of the generated HDL design onto the FPGA hardware. In our FPGA design, we used the well-known one-hot encoding scheme proposed by Sidhu and Prasanna [28] for implementing the NFA processing. In this encoding, the NFA states are represented by flip-flops and the state transitions are implemented by *AND*-ing and *OR*-ing the outputs of the flip-flops with the decoded input character. This representation allows processing one input character per clock-cycle, thus ensuring worst-case performance guarantees independent of the number of states that are concurrently active during the traversal. Further, this representation could be trivially extended to support the counter and boolean elements.

We optimized our FPGA designs using the single input and multiple outputs optimizations described by Becchi and Crowley [29]. Additionally, in order to handle large ANML-NFA networks with resource requirements exceeding the capacity of a single FPGA device, we implemented partitioning schemes in our tool-chain. These schemes use estimates of the FPGA logic utilization to break the input ANML-NFA network into multiple sub-networks that could then be deployed on multiple FPGA devices.

5.2 Comparison with Hyperscan

Hyperscan is a software library by Intel specifically designed for the purpose of matching large rulesets on CPUs [9]. Given a set of regexes, the library compiles them and creates a database which can be used for matching against streaming or blocked data with multiple threads. Since its open-source release in 2015, the library has emerged as the best pattern matching engine on CPU for advertisement filtration [30], database queries [31], etc. Therefore, we used Hyperscan for comparing the performance of the AP with the state-of-the-art on CPUs. However, extended PCRE features such as lookahead assertions are not supported by Hyperscan.[4] Since we use lookaheads for generating final PCRE from Snort rules, in place of the boolean *AND* element for generating ANML-NFA (as discussed in Section 3.3.1), we had to limit the comparison to PROTOMATA.

For benchmarking the performance of Hyperscan, we used the *hsbench* tool which is part of the Hyperscan library.[5] The tool takes the patterns to be searched for, in the PCRE format, and the data to be scanned as inputs. It then reports the time taken by a single-thread implementation of Hyperscan in scanning the given data for the set of patterns. Therefore, we converted the PROSITE motifs to the corresponding PCRE and used *hsbench* for comparing the performance of Hyperscan with that of PROTOMATA.

6 Results

For both the applications described in this paper, the ANML-NFA can be defined and compiled beforehand.

4. https://intel.github.io/hyperscan/dev-reference/compilation.html#pattern-support

5. http://intel.github.io/hyperscan/dev-reference/tools.html#benchmarker-hsbench

Clock-divisor (d)	location-modifier	#Boolean elements	#Counter elements	#Blocks	#Chips
1	*file_data*	0	0	1	1
	general	735	209	711	4
	general_raw	1	1	1	1
	http_client_body	104	62	41	1
	http_cookie	6	1	2	1
	http_header	155	5	53	1
	http_header_raw	1	0	1	1
	http_method	0	0	1	1
	http_stat_code	0	0	1	1
	http_uri	394	79	128	1
	http_uri_raw	3	3	1	1
2	*general*	3 191	273	1 820	11
	general_raw	8	2	2	1
	http_client_body	117	9	40	1
	http_header	226	6	33	1
	http_uri	425	4	47	1
	http_uri_raw	4	2	23	1
3	*general*	22	0	3	1
	http_uri	21	2	2	1

TABLE 1: AP resource requirements of Fast-SNAP buckets.

Therefore, although the compilation times of both these applications have been reported here, they do not figure in the run-time calculations. All the CPU-based operations (compilation and execution of the host application) are executed on a quad-core *Intel(R) Core(TM) i5-3570* CPU, running at 3.4 GHz with 8 GB of main memory.

The computation on the CPU comprises of fetching the input data, organizing the same into a dataflow, streaming the dataflow to the AP-board, and taking actions based on the occurrence of the patterns as reported by the AP. This is not expected to be the bottleneck for the applications discussed in this paper and can be easily hidden by an asynchronous multi-threaded pipeline.

The FPGA experiments were performed on two *Virtex-5* devices with different capacities: *XC5VLX30* and *XC5VFX200T*, the former comprising of 4 800 slices and the latter comprising of 30 720 slices. The Virtex-5 FPGAs have 4 flip-flops and 4 lookup tables (LUTs) per slice, resulting in 19 200 flip-flops and LUTs each on a XC5VLX30 device and 122 880 flip-flops and LUTs each on a XC5VFX200T. In all the cases, we report the processing throughput of a single input stream and the FPGA utilization. Supporting multiple input streams with the considered implementation requires duplicating the ANML-NFA network which requires multiple FPGA devices in some cases.

6.1 Fast-SNAP

6.1.1 Configuration Overhead

The Snort ruleset can be downloaded from the Snort website. The conversion of the active rules to the equivalent ANML-NFAs, as described in Section 3.3.1, and the compilation of the ANML-NFAs for all the buckets takes 659 minutes. The patterns for the rules are divided into 19 buckets for different combinations of location-modifiers and clock-divisors. If the AP-FSMs for all the buckets are loaded together in a single logical-core, they take up 16 AP-chips.

6.1.2 AP Run-time Estimation

The on-board resources required by different buckets in Fast-SNAP, grouped by their clock-divisors (d), are tabulated in TABLE 1. All the buckets require one chip each, except for the two *general* buckets with $d = 1$ and $d = 2$ which require 4 and 11 chips, respectively.

There are two possible methods of loading the buckets on the AP-board, one of which is chosen depending on the availability of the boards and the expected load corresponding to different location-modifiers. The first method is used when working with a single board. In this method, all the buckets with the same clock-divisor are combined and loaded in one logical-core. The buckets with $d = 1$ combined require 5 chips, those with $d = 2$ require 11 chips, and the $d = 3$ buckets require just 1 chip. As discussed in Section 2.2.3, the matching throughput for each bucket can be calculated as $1/d$ Gbps. Therefore, in order to extract a total matching throughput of 1 Gbps from the logical-cores with clock-divisor greater than 1, they are replicated d times. The total number of chips required in this configuration is $(5 \times 1) + (11 \times 2) + (1 \times 3) = 30$, available on one AP-board to provide a matching throughput of 1 Gbps. Alternatively, if the matching load is expected to be balanced across all the location-modifiers, then a higher throughput can be extracted using two AP-boards. The AP-FSM corresponding to every bucket is loaded in a separate logical-core and the buckets with $d > 1$ replicated. This requires a total of $(14 \times 1) + (16 \times 2) + (2 \times 3) = 52$ chips and is estimated to support a matching throughput of 1 Gbps for every location-modifier, leading to a total throughput of 11 Gbps.

Notice that, both the methods discussed above can be extended using more boards for higher throughputs. Further, since the AP-FSMs for a given ruleset will have to be loaded only once, the time required for loading is not considered while calculating the throughput. The output handling time is also not considered because the application is expected to generate sparse output.

6.1.3 Evaluation on FPGA

The results of evaluation of the Fast-SNAP network on the larger XC5VFX200T FPGA are shown in TABLE 2. Since the clock-divisor does not affect the processing rate on FPGA, the ANML-NFAs for the buckets with the same location-modifier but different clock-divisor were combined for the evaluation on FPGA. However, whenever a bucket could not fit on one FPGA device, it was partitioned to fit on multiple FPGA devices. The first ten rows report the results for the ten smaller buckets that do not require partitioning. These buckets are compiled together in a single partition to form an aggregate bucket, the result for which is shown in the next row. The number of LUTs, flip-flops, and slice utilization for every partition are reported in the columns $2 - 4$ and the throughput reported after running synthesis, mapping, and place-and-route on the FPGA device is shown in the column 5. The *general* bucket is the only one requiring partitioning. When the maximum expected logic utilization is set to 100%, five partitions are created, as shown in the table. Consequently, the whole Fast-SNAP dataset can be encoded on six XC5VFX200T devices (with a 71–96% occupancy), leading to per-stream processing throughput between 1.1 and 1.9 Gbps.

Bucket	#LUTs	#flip-flops	Slice utilization	Single-Stream throughput (Gbps)
file_data	66	112	1%	9.383
general_raw	589	422	1%	3.115
http_client_body	11 431	13 081	27%	1.813
http_cookie	308	347	1%	3.964
http_header	3 966	10 433	20%	1.751
http_header_raw	23	16	1%	9.195
http_method	62	42	1%	9.090
http_stat_code	6	6	1%	9.389
http_uri	9 998	26 740	61%	1.841
http_uri_raw	3 303	3 219	6%	1.956
Aggregate	24 735	45 750	94%	1.531
general - partition 1	63 835	84 172	96%	1.306
general - partition 2	61 448	77 567	85%	1.168
general - partition 3	83 598	98 287	89%	1.133
general - partition 4	45 883	56 580	71%	1.921
general - partition 5	60 635	61 193	75%	1.339

TABLE 2: Results of synthesis, mapping, and place-and-route on XC5VFX200T FPGA for the Fast-SNAP network.

6.2 PROTOMATA

6.2.1 Configuration Overhead

A file named *prosite.dat* can be downloaded from the PROSITE website which contains all the pattern-motifs in the database. The conversion of these motifs from the PROSITE pattern notation to the ANML-NFAs, described in Section 4.3.1, takes 1.6 seconds and the compilation of these ANML-NFA takes about 20 minutes. All the Selective-enable automata for the motifs can be programmed using half the resources on a single AP-chip. Similarly, all the All-repeat-check automata can fit into one AP-chip.

6.2.2 AP Run-time Estimation

The streaming time for PROTOMATA is the estimated time required to stream the dataflow, including all the input proteins and the preamble sequence, in the absence of any stalling by the output-handling bottleneck, i.e. at a streaming rate of 1 Gbpbs. The output handling time is the time required to read out all the output-vectors from the output-buffer. Assuming the worst case, the number of symbol-cycles required to read out an output-vector can be calculated as $40 \times \min(p, 6) + 16$, where p is the number of reporting-STEs matched in the output-event. In our tests none of the motif occurrences happened on the same symbol-cycle. Therefore, the time taken to read out the vector is taken to be 56 symbol-cycles for our calculations. The overall run-time of PROTOMATA can then be estimated as a maximum of the streaming time and the output handling time. The time required for loading the AP-FSMs into the board can be easily amortized over multiple queries and is therefore not considered in the run-time calculations.

6.2.3 Comparison with Hyperscan

The performance of Hyperscan and PROTOMATA is compared in TABLE 3 using all the motifs from PROSITE and various proteomes (all proteins from a single organism) from the UniProtKB database. The Swiss-Prot section lists the manually annotated sequences from the database, whereas TrEMBL contains the computationally analyzed sequences from the database. The number of protein sequences from the proteomes and their combined lengths are expressed in the columns 3 and 4 respectively. The next column indicates if the search for frequently occurring motifs is enabled or not. This has a huge impact on the number of motif occurrences found in the protein sequences as reported in the column 6.

The run-times of Hyperscan and PROTOMATA are listed next. The average time taken by Hyperscan in searching the proteomes for the motifs over 20 repetitions, as reported by *hsbench*, is noted in the column 7. These times represent the *scanning time* only and do not include the *compilation times*. The estimated streaming time, output handling time, and the overall run-time for PROTOMATA, discussed in the previous section, are listed in the columns 8 through 10. The estimated speedup of PROTOMATA over Hyperscan, reported in the next column, greatly depends on the output generation rate. If the rate of output generation is very high, then the overall processing rate on the AP slows down. This is evident in the case of all the proteomes in the UniProtKB/TrEMBL. In this case, an output is generated every 4 to 5 cycles, the highest among all the cases. Correspondingly, the speedup is the lowest. The rate of output generation is also greatly enhanced if the search for frequently occurring motifs is enabled. Using this setting in all the other cases, an

						Hyperscan	PROTOMATA			
Database	Organism(s)	#Protein sequences	#Amino acids	Frequently occurring motifs	#Motif occurrences	Average run-time (in ms)	Streaming time (in ms)	Output handling time (in ms)	Overall run-time (in ms)	Speedup
UniProtKB/ Swiss-Prot	E. coli	4 305	1 360 331	enabled	66 263	696	10	28	28	24.86
				disabled	1 642	671	10	1	10	67.10
	human	20 183	11 336 473	enabled	633 487	5 656	85	267	267	21.18
				disabled	25 228	5 403	85	11	85	63.56
	all	479 406	174 899 570	enabled	9 827 918	87 627	1 307	4 138	4 138	21.18
				disabled	767 995	83 061	1 307	323	1 307	63.55
UniProtKB/ TrEMBL	E. coli	4 333	1 372 277	enabled	66 996	705	10	28	28	25.18
				disabled	1 653	668	10	1	10	66.80
	human	67 084	22 252 781	enabled	1 239 750	11 629	166	522	522	22.28
				disabled	40 141	11 241	166	17	166	67.72
	all	18 394 018	6 583 760 868	enabled	1 694 988 495	3 465 616	49 190	713 679	713 679	4.86
				disabled	1 338 370 343	3 287 743	49 190	563 524	563 524	5.83

TABLE 3: Comparison of run-times from Hyperscan and PROTOMATA.

output is generated every 18 to 21 cycles and the speedup varies between 21 and 25 times. If the search for the frequently occurring motifs is disabled (general practice in the field), then the rate of output generation is much lower and the expected speedup increases to between 63 and 68 times.

The results for PROTOMATA are calculated considering a single logical-core executing on the AP-board. Since the automata for PROTOMATA can fit inside a single AP-chip, 32 logical-cores can be executed in parallel on the AP-board. One of the features of the execution on the AP is that every logical-core can operate on a separate dataflow. Consequently, if the input size is large enough it can be uniformly broken up into 32 parts and streamed to all the logical-cores in parallel to extract a further speedup of 32 times. This is not true of Hyperscan. The authors ran experiments by distributing the data uniformly across multiple cores, and then distributing the patterns uniformly across multiple cores. In either case, uniform load balancing could not be achieved automatically. Therefore, multiplication of the single-core performance of Hyperscan by the number of cores on the CPU may be achieved only in the best case.

6.2.4 Evaluation on FPGA

TABLE 4 shows the results of FPGA evaluation of the PROTOMATA network. The whole PROTOMATA network fits inside the larger XC5VFX200T device, requiring nearly 30% of the LUTs and flip-flops and 64% of the slices. On the smaller XC5VLX30 FPGA, we performed multiple experiments by limiting the FPGA utilization to a given percentage, which is the reference occupancy parameter reported in the second column. Limiting the occupancy to 30%, 50%, and 70% of the FPGA capacity leads to 7, 4, and 3 partitions, respectively. Each partition, represented by a row in the table, requires a separate FPGA device. The reported throughput varies between 1.3 and 2 Gbps per partition on XC5VLX30 and is equal to 1.4 Gbps on XC5VFX200T. Since smaller partitions allow easier routing on the FPGA resources, the reported throughput is usually larger for them.

The FPGA streaming time can be computed by dividing the bit length of the input stream by the processing throughput (1.4 Gbps for PROTOMATA). As in the AP implementation, we store the motif occurrences in an output-vector. This output-vector is buffered and outputted in $\lceil \frac{r}{op} \rceil$ clock-cycles, where r is the number of reporting states in the ANML-NFA and op is the number of output ports on the FPGA device. The FPGA used has 960 output ports and the PROTOMATA network has 1 309 reporting states. For the considered datasets, this results in the output processing overhead varying from 0.018 ms, for the case of E. coli proteomes in the UniProtKB/Swiss-Prot database with the search for frequently occurring motifs disabled, to 19310.9 ms, when searching for frequently occurring motifs in all the proteomes from the UniProtKB/TrEMBL database.

Device	Reference Occupancy Parameter	#LUTs	#flip-flops	Slice utilization	Single-Stream throughput (Gbps)
XC5VFX200T	-	33 005	36 278	64%	1.400
XC5VLX30	30%	5 697	5 889	62%	1.867
		5 858	5 987	64%	1.560
		5 861	5 952	64%	1.874
		5 643	5 918	63%	1.576
		5 810	5 953	67%	1.798
		6 008	5 749	59%	1.949
		1 324	842	13%	2.052
	50%	8 953	9 709	92%	1.394
		9 330	9 775	93%	1.910
		9 062	9 758	90%	1.325
		7 447	7 045	69%	1.477
	70%	12 439	13 552	97%	1.573
		12 562	13 626	97%	1.746
		9 296	9 102	94%	1.883

TABLE 4: Results of synthesis, mapping, and place-and-route on Virtex-5 FPGA for PROTOMATA.

7 Conclusion

The AP is a soon to be released reconfigurable processor which is purpose-built to execute thousands of NFAs in parallel. Therefore, it lends itself well to the acceleration of software which check for the occurrences of thousands of patterns in an input data stream. Using this capability, we have developed two applications, Fast-SNAP and PROTOMATA, which check for patterns of intrusion detection in network packets and biologically meaningful patterns in protein sequences, respectively. Both of these provide a glimpse of how such applications should be programmed using this new accelerator hardware. In addition, the techniques described in this paper are applicable to the design and analysis of a wide variety of applications on the AP.

Acknowledgments

- Paul Dlugosch and his team for providing insights on various aspects of the AP. This research is funded in part by Micron Technology, Inc.
- Roy, Srivastava, and Aluru are partly funded for this work by the NSF Exploratory Grant CCF-1448333 and partly by Micron Technology, Inc.
- Nourian and Becchi are funded through NSF awards CCF-1740583 and CNS-1724934.

References

[1] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong, "GPU-based NFA implementation for memory efficient high speed regular expression matching," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 129–140, 2012.

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

[3] K. Peng, S. Tang, M. Chen, and Q. Dong, "Chain-based DFA deflation for fast and scalable regular expression matching using TCAM," in *Proc. of ANCS*. IEEE/ACM, 2011, pp. 24–35.

[4] "The Micron Automata Processor Documentation," http://www.micronautomata.com/documentation, accessed: Jan, 2018.

[5] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[6] "Perl Compatible Regular Expressions (PCRE) documentation," https://www.pcre.org/original/doc/html, accessed: Jan, 2018.

[7] "Snort," https://www.snort.org/, accessed: Jan, 2018.

[8] C. J. Sigrist, E. De Castro, L. Cerutti, B. A. Cuche, N. Hulo, A. Bridge, L. Bougueleret, and I. Xenarios, "New and continuing developments at PROSITE," *Nucleic acids research*, vol. 41, no. D1, pp. D344–D347, 2013.

[9] "Hyperscan," https://www.hyperscan.io, accessed: Feb, 2018.

[10] I. Roy, A. Srivastava, and S. Aluru, "Programming techniques for the Automata Processor," in *45th International Conference on Parallel Processing*, Aug 2016, pp. 205–210.

[11] M. Roesch *et al.*, "Snort: Lightweight Intrusion Detection for Networks." in *LISA*, vol. 99, 1999, pp. 229–238.

[12] 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.

[13] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '09. New York, NY, USA: ACM, 2009, pp. 30–39.

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

[15] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for Accelerating SNORT IDS," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS '07. New York, NY, USA: ACM, 2007, pp. 127–136.

[16] M. Becchi, "Data structures, algorithms and architectures for efficient regular expression evaluation," Ph.D. dissertation, Washington University, 2009. Department of Computer Science and Engineering., 2009.

[17] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, "Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator," in *Proc. of MICRO*. IEEE, 2012, pp. 461–472.

[18] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata," in *Proc. of SIGCOMM*, 2008, pp. 207–218.

[19] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: The unified automata processor," in *Proc. of MICRO*. New York, NY, USA: ACM, 2015, pp. 533–545.

[20] R. Staden, "Screening protein and nucleic acid sequences against libraries of patterns," *Mitochondrial DNA*, vol. 1, no. 6, pp. 369–374, 1991.

[21] L. Kolakowski Jr, J. Leunissen, and J. Smith, "ProSearch: fast searching of protein sequences with regular expression patterns related to protein structure and function." *Biotechniques*, vol. 13, no. 6, pp. 919–921, 1992.

[22] P. R. Sibbald and P. Argos, "Scrutineer: a computer program that flexibly seeks and describes motifs and profiles in protein sequence databases," *Computer applications in the biosciences: CABIOS*, vol. 6, no. 3, pp. 279–288, 1990.

[23] J. C. Wallace and S. Henikoff, "PATMAT: a searching and extraction program for sequence, pattern and block queries and databases," *Computer applications in the biosciences: CABIOS*, vol. 8, no. 3, pp. 249–254, 1992.

[24] M. J. Sternberg, "PROMOT: A FORTRAN program to scan protein sequences against a library of known motifs," *Computer applications in the biosciences: CABIOS*, vol. 7, no. 2, pp. 257–260, 1991.

[25] E. De Castro, C. J. Sigrist, A. Gattiker, V. Bulliard, P. S. Langendijk-Genevaux, E. Gasteiger, A. Bairoch, and N. Hulo, "ScanProsite: detection of PROSITE signature matches and ProRule-associated functional and structural residues in proteins," *Nucleic acids research*, vol. 34, no. suppl 2, pp. W362–W365, 2006.

[26] M. Magrane, U. Consortium *et al.*, "UniProt Knowledgebase: a hub of integrated protein data," *Database*, vol. 2011, p. bar009, 2011.

[27] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi, "Demystifying automata processing: Gpus, fpgas or micron's ap?" in *Proceedings of the International Conference on Supercomputing*. ACM, 2017, p. 1.

[28] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Proc. of the 9th Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 227–238.

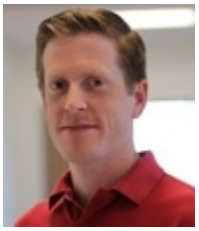
[29] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proc. of 4th Symposium on Architectures for Networking and Communications Systems (ANCS)*. New York, NY, USA: IEEE/ACM, 2008, pp. 50–59.

[30] J. Li, D. He, F. Liu, and H. Wang, "The application of regex in advertisements filtration and performance analysis," in *Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2016 8th International Conference on*, vol. 1. IEEE, 2016, pp. 28–32.

[31] Z. István, D. Sidler, and G. Alonso, "Runtime parameterizable regular expression operators for databases," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 204–211.

Indranil Roy is the Senior Vice President of Research and Development at Natural Intelligence Semiconductor. Before this, he was an architect of application development at Micron. He received his PhD from the School of Computational Science and Engineering within the College of Computing at Georgia Institute of Technology. His current research interests includes mapping a variety of graph applications for acceleration on the Automata Processor.

Ankit Srivastava is a PhD student in the School of Computational Science and Engineering within the College of Computing at Georgia Institute of Technology. His research interests include high performance computing, parallel graph algorithms, and automata processing.



Matt Grimm was a senior application developer with Micron Technology, Inc. at the time of writing this manuscript. His primary interest was the development of accelerated applications using the Automata Processor. Currently, Matt is employed with Clearwater Analytics.

Marziyeh Nourian is a PhD student at North Carolina State University. Her research interests are in automata processing, parallel computing, heterogeneous and reconfigurable architecture.

Michela Becchi is an associate professor at North Carolina State University. Earlier, she held a faculty position at the University of Missouri. Her research interests lie at the intersection between parallel computer architecture, systems software, and applications.

Srinivas Aluru is a professor in the School of Computational Science and Engineering within the College of Computing at Georgia Institute of Technology. He conducts research in high performance computing, bioinformatics, and systems biology, combinatorial scientific computing, and applied algorithms. He is a Fellow of the American Association for the Advancement of Science (AAAS) and the Institute for Electrical and Electronic Engineers (IEEE).