High Performance Pattern Matching using the Automata Processor

Indranil Roy*, Ankit Srivastava*, Marziyeh Nourian†, Michela Becchi† and Srinivas Aluru*

Email:{iroy, asrivast}@gatech.edu, mndk3@mail.missouri.edu, becchim@missouri.edu and aluru@cc.gatech.edu

**School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA*

†*Department of Electrical and Computer Engineering, University of Missouri, Columbia, MO 65211, USA*

***Abstract*—In this paper, we study the acceleration of applications that require searching for all occurrences of thousands of string-patterns in an input data-stream, using the Automata Processor (AP). For this purpose, we use two applications from two fields, namely, network security and bioinformatics. The first application, called Fast-SNAP (for Fast-SNort using AP), 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 10.3 Gbps. The second application, called PROTOMATA (for PROTein autOMATA), looks for all occurrences of 1308 protein motifs from the PROSITE database in protein sequences. PROTOMATA is up to half a million times faster than its single-CPU-based counterpart. The techniques developed to program these applications may be useful in the design and development of similar applications using this new hardware accelerator.**

***Keywords*-Finite Automata, Regular Expressions, Automata Processor, FPGAs, Intrusion detection, Protein motifs.**

I. Introduction

Acceleration of applications that find all occurrences of thousands of patterns in an input data-stream presents significant challenges. While GPU-based solutions struggle with handling execution divergence [1], custom-made 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 have been reported by solutions exploiting the reconfigurability and parallelism of FPGAs. Through the concurrent execution of multiple Nondeterministic Finite Automata (NFAs) in hardware, significant speed-up 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 FPGA based solutions in terms of the number of NFAs that can be executed concurrently on a single device.

We have developed two applications as demonstrators. The first application, called Fast-SNAP (for Fast-SNort using AP), scans network data-streams for occurrences of *signatures* of intrusion derived from the *Snort* database [6]. The second application, called PROTOMATA (for PROTein autOMATA), inspects protein sequences for existing occurrences of protein *motifs* listed in the PROSITE database [7]. The NFAs developed for these applications illustrate simple design techniques to extract maximum performance benefits from the AP.

Recently, the first engineering samples of AP chips were demonstrated at the *International Supercomputing Conference* (ISC-15), with our design and implementation of the PROTOMATA application running in hardware with $40X$-speedup over the *de facto* CPU based implementation. These engineering samples suffer from a variety of software and hardware limitations hindering overall performance. With these limitations eradicated in the production samples to be marketed by the end of 2016, we estimate that our PROTOMATA application would run hundreds of thousands of times faster than its single-CPU counterparts. Similarly, we estimate that our Fast-SNAP algorithm will support *Deep Packet Inspection* (DPI) of 4312 signatures of malicious traffic at 10.3 Gbps. In contrast to the existing methods, Fast-SNAP is able to handle close to the whole Snort active ruleset. 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 included results obtained by implementing these NFAs in FPGAs as well.

The rest of the paper is organized as follows. In Section II, we briefly describe the programming model and run-time environment of the AP. We then outline our optimization strategies in Section III. Subsequently, Fast-SNAP and PROTOMATA are detailed in Sections IV and V respectively. Finally, the estimated speedup of these applications vis-à-vis other state-of-the-art implementations and our FPGA implementations are presented in Section VI.

II. Automata Processor

A. Overview

The *rules* to be executed on the AP are defined as regular expressions using the Perl Compatible Regular Expression (PCRE) syntax or as NFAs, using a proprietary language called Automata Network Markup Language (ANML, pro-

nounced as "animal"). 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 compute-cycle (henceforth called a *symbol-cycle*), then the host CPU program is notified with a report identifying the rule(s) and the offset in the dataflow where the match(es) occurred.

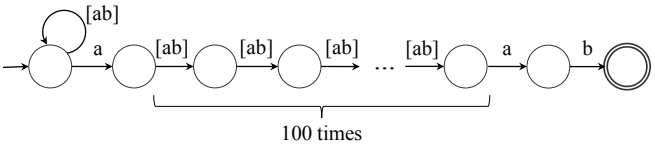
B. Automata Design

The PCRE syntax [8] is well known to the programming community. Features of PCRE supported by the AP-compiler are listed online [9]. However, ANML is proprietary to the AP and therefore we provide a brief description of the same.

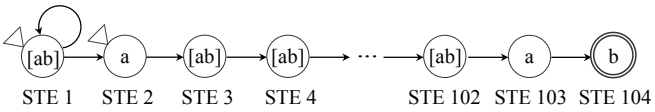
1) ANML Representation: The programmable elements in an AP chip consist of processing elements called *State Transition Elements* (STEs), *Counter Elements* and *Boolean Elements*; and a reconfigurable *routing network*. Representation of an NFA in ANML (henceforth called *ANML-NFA*) depicts the connections of these native programmable elements using the routing network. Although an ANML-NFA is different from a classical *state-diagram*, we describe an easy way to convert the latter to the former.

The NFA depicted by the state-diagram in Figure 1a accepts any string from the alphabet $\{a, b\}$ containing a substring of length 103 symbols whose first symbol is an a and whose last two symbols are a followed by b. Notice that the conversion of this NFA to its equivalent DFA leads to state-space explosion according to well-known theory [10]. The equivalent ANML-NFA is shown in Figure 1b. Edge transitions are modeled as STEs in ANML-NFA. An STE is depicted using a circle with its label placed inside it. The label of an STE can be any character-class from the 8-bit symbol space. STEs corresponding to outgoing edges of a start state are marked as *start-STE*s and denoted by triangles on top. Similarly, STEs representing incoming edges of the final states are marked as *reporting-STE*s, denoted by double-outlines. All internal states are captured by connecting STEs representing incoming edges of the state to all STEs representing outgoing edges of that state.

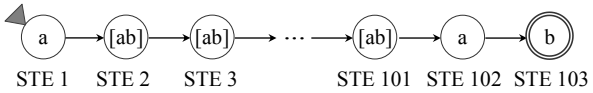
The processing of a dataflow by an ANML-NFA can be described as follows. In every symbol-cycle, a single 8-bit symbol from the dataflow is processed. In the first symbol-cycle, only the start-STEs are active. If the first symbol in the dataflow matches the label of a start-STE, then all STEs connected to its outgoing edges are activated for the next symbol-cycle. The processing continues in the next cycle by processing the second symbol in the dataflow and so on. If a reporting-STE is matched, then an *output event* is generated which identifies the reporting-STE (and hence the rule it belongs to), along with the offset in the dataflow where the match occurred.



(a) NFA in classical state diagram representation.



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



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

Figure 1: Representation of automata in ANML.

2) Special Features:

All-input-start STEs: A start-STE can be classified as a *start-of-data* STE or an *all-input-start* STE, depicted with a solid triangle. While a start-of-data STE is active only during the first symbol of the dataflow, an all-input-start STE is active in every symbol-cycle. Therefore, the occurrences of rules defined using the latter need not be anchored to the beginning of the dataflow. For example, the ANML-NFA in Figure 1b can also be represented using an all-input-start STE as shown in Figure 1c.

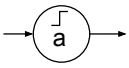


Figure 2: *Latched-STE* to identify the symbol 'a'.

Latched STEs: The output of an STE may be *latched* so that the STEs connected to its outgoing routing lines remain activated, until the end of the dataflow, after the STE is matched for the first time. Such an STE is denoted by a ⌟ sign as shown in Figure 2.

Counter Elements: If a rule contains a repeated pattern, then a counter element may be used to compress the ANML-NFA. For example, the ANML-NFA shown in Figure 1c can be defined as the compact automaton shown in Figure 3. The counter element is represented using a rectangle with two input lines shown as triangles along the left boundary and a single output line on the right boundary. The first input line, called the *count-line*, is denoted by the letter C, whereas the other input line, called the *reset-line*, is denoted by the letter R. The element also has a programmable 12-bit *target-value* shown within a solid rectangle.

Counter elements do not process symbols themselves and do not consume any symbol-cycles. At the beginning of the

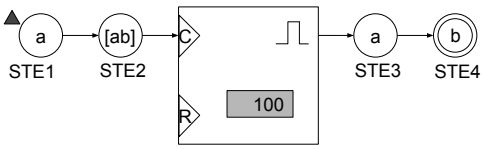


Figure 3: Compressed ANML-NFA using a counter-element with *pulsed-output*.

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 counter element activates its output-line. If the counter element is programmed to generate a *pulsed-output*, denoted by the ⎍ sign, 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 ⎾ sign, then the output line is activated until either the counter 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 different boolean operations namely *OR*, *AND*, *NOR*, *NAND*, *sum-of-product* and *product-of-sum*. Similar to counter elements, boolean elements do not process any symbols or consume any symbol-cycles. If, within a symbol-cycle, the input lines are simultaneously activated in a manner such that the boolean operation is satisfied, then the output line of the boolean element is activated. An example of utilizing a boolean element for combining multiple ANML-NFA is described in Section IV-C1.

C. ANML Macro

Parts of ANML-NFA can be defined modularly as an ANML macro. These macros not only help in creating building blocks for larger ANML-NFA (or macros), but also reduce the compile time of larger automata. A macro, once compiled, obtains a partial mapping of the programmable elements and routing lines to logical entities in the AP chip. Therefore, compiling automata with these constituent macros does not incur this overhead. Different instances of a macro are identical in structure, but may differ in the labels of the STEs. These labels are declared as parameters of the macro which may be defined at run-time, just before loading, with negligible overhead. The use of these features of ANML macros are illustrated in Section V-B1.

D. Software Development Kit (SDK)

The AP board is accompanied with an SDK which enables the user 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-NFA internally before creating the AP-FSMs.

Debug environment: The execution of the native ANML-NFA can be simulated using the workbench or an *AP-emulator* against test dataflow(s). The AP-emulator may also be used to emulate the execution of AP-FSMs. The feedback from these simulations and emulations can be used to correct the original ANML-NFA designs.

Run-time environment: The SDK provides API calls to load the AP-FSMs into the AP board, stream dataflows and handle output at run-time.

E. Programming Resources

Each AP chip contains $49,152$ STEs, 768 counter elements and $2,304$ boolean elements. An AP board consists of 48 such chips arranged into 6 physical *ranks*. Therefore, a single AP board cumulatively contains $2,359,296$ STEs, $36,864$ counter elements and $115,344$ boolean elements. This is sufficient to accommodate very large rulesets. For example, all the rules in Snort and PROSITE databases can be programmed into 18 and 1 AP chips, respectively.

The chips on a board can be organized into *logical cores* containing 2, 4 or 8 chips in each rank. Every logical core is presented with a separate dataflow in parallel and each dataflow is processed at 128 million symbols per second, or at 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 depends on the maximum number of these elements connected in succession. The size of logical cores is therefore chosen to derive maximum data parallelization and hence performance from the AP board which is connected to the CPU through a high-speed PCIe interconnect.

III. Application Design Guidelines

1) Using Precompiled Automata: Compilation of ANML-NFA involves complex place-and-route algorithms which may take considerable time. Therefore, whenever possible, the automata should be compiled beforehand. Fortunately, for the applications discussed in this paper, the rules are known *a priori* and hence the automata can be defined and compiled in advance.

2) Batching Output Generation: The output handling rate of the AP is much slower than the input processing rate and may end up as the rate-constraining step. The output handling on the AP is described as follows: The reporting STEs in the AP chip are organized into 6 output regions. If any STE(s) from an output region reports during a cycle, then an *output-vector* for that region is stored into an *output-buffer*. The length of this output-vector is independent of the number of STE(s) that reported in that cycle. If the buffer has space for the incoming vectors, then they can be stored within the same clock-cycle and the input processing continues unabated from the next cycle. On the other hand, if the buffer is full, then the entire processing pipeline is stalled, so that the buffer 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 be between 56 and 256 times slower. Although the user has no control over the placement of the reporting STEs in these regions, applications can gain significantly by batching multiple output generating events into a single symbol-cycle. This will be demonstrated in Section V-B1.

IV. FAST-SNAP

A. Background

Fast-SNAP is a Network Intrusion Detection (NID) tool which scans for *signatures* of intrusion in network data based on *rules* described in the widely used Snort database [6]. Pattern matching rules in the database define string patterns for identifying signatures of anomalous activity in the network data. This ruleset is updated as and when new signatures are discovered. Currently, the ruleset contains $5,310$ active pattern matching rules, which provides a significant computational challenge for current bandwidth requirements and frequency of cyber-attacks. Therefore, accelerated solutions using GPUs and FPGAs have received considerable attention in literature.

Cascarano et al. [11] proposed the first NFA-based pattern-matching engine 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 Becchi et al. [12], which contains 534 regular expressions. 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 problem by identifying states that cannot be simultaneously active in the original NFAs and creating *virtual-NFAs*. Using the virtual-NFAs, they reported a maximum throughput of nearly 13 Gbps on small datasets consisting of only 16 to 36 patterns.

FPGA based solutions rely on configuring the processor to concurrently execute multiple NFAs in hardware. Yang et al. [13] 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. [14] 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 by the Snort rules compiler to *VHDL* code. However, the capacity of even the largest FPGAs is not enough to accommodate large rulesets [13], [15], requiring them to be partitioned and handled by multiple FPGA devices.

RegX [16] is a regular expression matching engine which uses compressed DFAs and a variant of XFAs [17] 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 5000 complex patterns (that is, patterns including wildcard repetitions and counting constraints). Fang et al. [18] proposed a programmable Unified Automata Processor, a special purpose automata processing architecture that can achieve throughputs up to 295 Gbps on datasets consisting of hundreds to thousands regular expressions.

B. Snort Rules

Snort rules are written in a lightweight description language [6]. Each rule contains a *header* section and an *options* section. The header section specifies the protocol, source and destination of the network packet for which the rule is active and the type of action to be taken if the rule is matched. The options section consists of one or more *keywords* belonging to the following categories: *general*, *non-payload detection*, *payload detection* and *post-detection*.

General keywords provide generic information about the rule such as an *sid*, a unique integer identifier for the rule. Non-payload detection keywords describe anomalous values for various fields in the header section of a network packet. On the other hand, payload detection keywords define patterns to identify in the data section of a network packet. Post-detection keywords specify actions to be taken if an occurrence of the signature is detected.

For example, consider the following sample rule.

```
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; )
```

The header section is specified in the beginning and states that the rule is active for *tcp* packets going to port 80 and an *alert* should be raised if the rule is matched. The options section is specified within parenthesis and lists multiple keywords separated by semicolons. The first keyword specifies that the *sid* of the rule is 42 while the following keywords are of payload detection type.

Most of the patterns in the Snort rules are defined using payload detection keywords. These keywords either specify strings to be matched exactly (designated by the keyword `content`) or PCRE (designated by the keyword `pcre`). Our sample rule contains seven strings to be matched (`foo`, `bar`, `kludge`, etc.) and one PCRE (namely `/foo[0-9]{10}bar`). Matching of the patterns can be constrained by two types of *modifier* keywords: *location-modifier* and *distance-modifier*, both of which are described in Section IV-C1. 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.

C. Methodology

1) Automata Design: 81% of the active pattern matching rules in Snort can be efficiently implemented using the AP. 17% rules contain the keywords `byte_test`, `byte_jump` or `byte_extract`. These rules extract various parameters for the pattern matching operations from specific `bytes` in the input data-stream. The implementation of these rules on the AP would require reprogramming the device based on the input data, which is inefficient. The remaining 2% of the rules cannot be implemented using the AP because of

known issues with the AP-compiler, which are documented in [9]. Checking for the rules which cannot be implemented efficiently using the AP should be carried out using existing methods.

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

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, location-modifiers can specify whether the pattern should be matched in the raw data or in the normalized data. For example, using the keyword `http_uri`, our 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, patterns qualified by different location-modifiers are placed in their respective buckets, along with the *sid* of the rule. This allows us to program patterns from different buckets into different 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: Distance-modifiers specify constraints on the location of the occurrence of a pattern in the data-stream, relative to an anchor. This anchor could either be the beginning of the stream or the end of the occurrence of the previous pattern. Keywords *offset* and *depth* are used to specify minimum and maximum distances from the beginning of the stream; whereas, keywords *distance* and *within* define minimum and maximum distances relative to 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, patterns within each bucket are considered separately. Patterns belonging to the same rule which are related through distance-constraints are combined into a single PCRE using *repetition quantifiers*. In our sample rule, patterns `foo` and `bar` in the *general* bucket are combined to get the PCRE: `foo.{10,}bar`.

Step 3. 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 [8]. 6% of the supported rules contain patterns with back-references. However, since the match depends on the input stream, the AP cannot handle back-references efficiently.

In the third step, back-references in patterns are substituted with the corresponding sub-patterns being referred to.

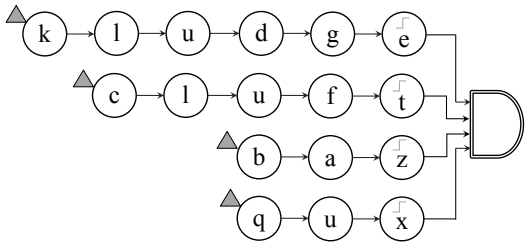


Figure 4: ANML-NFA corresponding to the *http_header* bucket of the sample rule.

The language of the resulting pattern, after substitutions, is a superset of the language of the original pattern. Therefore, *sid* of the corresponding rule and the original pattern is recorded for eliminating false positive matches during execution stage.

Step 4. 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 data-stream (in any order) for a rule to be triggered.

In the fourth and final step, PCRE(s) for a rule in a bucket are converted into an ANML-NFA. In the case of multiple PCREs for a rule, 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 combined ANML-NFA for the same is shown in Figure 4. Notice that the classical NFA for this automaton is fairly complex and large because it needs to capture all the combinatorial ways in which the substrings `kludge`, `cluft`, `baz` and `qux` may be ordered in the dataflow.

At the end of these steps, in every bucket, there is at most one ANML-NFA per rule. Automata from each bucket is compiled into a single AP-FSM which is then loaded into a dedicated logical core at run-time. However, some buckets may contain very few automata or correspond to segments in the network packet which are very short. Automata from such buckets are combined and a single composite AP-FSM is generated to be loaded into one logical core.

2) Execution Stage: Since the automata are already compiled, the AP-FSMs are directly loaded into the AP board and the processing of the network packets begins instantaneously. The host application breaks a network packet 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 back references then the packet is matched against the original pattern(s), recorded in *Step 3* of the design stage, using existing methods. In case a rule is programmed as multiple automata in different buckets, pertaining to keywords with different location-modifiers, the host application triggers the necessary action(s) specified in the original Snort rule only if all the constituent automata generate a report within the same network packet.

Even after loading all the automata for the Snort rules,

significant portions of the AP board remain unused. Therefore, the logical cores for some buckets are replicated so that data from different network packets can be handled in parallel. In this way, a very high throughput DPI engine is realized.

V. PROTOMATA

A. Background

PROSITE [7] is a large annotated ruleset of known protein *motifs*. A motif is defined as a small conserved region in a protein sequence which plays a biologically meaningful role. A motif can be described as a string-based *pattern-motif*, or as a *profile-motif* which uses a weight-matrix-based method to calculate similarity. PROTOMATA only scans for occurrences of pattern-motifs. Currently, PROSITE has 1308 pattern-motifs. The database can be searched in the following three modes.

- *Use case 1*: Select protein sequences(s) against all motifs in the PROSITE database.
- *Use case 2*: Select motif(s) against a protein database (UniProtKB, PDB, or user-defined).
- *Use case 3*: Select motif(s) against selected protein sequence(s).

A Perl-based version of the tool called *ps_scan* [19] can be downloaded for execution on a local machine. This serves as the baseline for our comparative studies. Almost all other solutions were developed in the early '90s and remained untraceable in spite of our best efforts.

B. Methodology

For each motif, two pre-compilable automata are generated. Both these automata allow the search for the motif to be selectively enabled at run-time. The first one is called *Locate-occurrence* automaton which reports the location of each occurrence of a motif in the protein sequences. The second one is called the *Global-match* automaton which generates a single report at the end of streaming of all the protein sequences if the motif occurs in all of them. This reduces the frequency of output-generation and increases overall performance. If the location of occurrences of these common motif(s) in the sequences is also desired, then a second pass is made using the Locate-occurrence automaton and turning on the search for the common motifs only.

For the purpose of demonstration, we chose the PROSITE motif *PS00430*:
$< \quad x(10,115) - [DENF] - [ST] - [LIVMF] - [LIVSTEQ] - V - \{AGPN\} - [AGP] - [STANEQPK]$
The motif is expressed in *PROSITE pattern notation*, description of which can be found online. In this notation, '$<$' denotes the beginning of the sequence, 'x' denotes any amino acid, '$(10,115)$' denotes a repetition between 10 and 115 times, '$-$' denotes concatenation, '$[\ldots]$' denotes a character class and '$\{\ldots\}$' denotes a *complementary class*, i.e. any amino acid but the ones listed within the curly braces. Figure 5 shows the occurrence of the PROSITE motif *PS00430* in the *Lissencephaly-1 homolog (D3BUN1)* protein sequence.

Every motif is assigned a unique 2-byte *PROTOMATA-id* between 0_{16} and $feff_{16}$. This id is different from the PROSITE-id, e.g. the PROTOMATA-id of the above motif is $018a_{16}$. A simple one-to-one mapping is maintained on the host application to provide the necessary interface between the user input (using PROSITE-ids) and the working of PROTOMATA (using PROTOMATA-ids).

The dataflow consists of a *preamble-sequence* and the protein sequences concatenated to each other. The preamble-sequence contains the concatenated list of PROTOMATA-ids of the motifs to be enabled. This list can be created based on user input or pre-generated based on standard choices such as disabling the search for "frequently occurring motifs". The end of the preamble and protein sequences is delimited with the hexadecimal ff_{16} symbol. Notice that the range of PROTOMATA-ids is chosen to ensure that this end-delimiter is not confused with the first symbol of any PROTOMATA-ids in the preamble-sequence. The end of the dataflow is earmarked by the hexadecimal 0_{16} symbol.

1) Automata Design: The notations used for the labels of STEs in this section are as follows. An upper case letter denotes an amino acid and the corresponding label consists of the ASCII equivalent of the letter in both upper and lower cases. The square brackets for the character class have been omitted. $\sum$ represents the character class containing the ASCII equivalent of all letters representing amino acids in both upper and lower cases. '.' represents the entire 8-bit symbol set. All other labels denote hexadecimal numbers used as delimiter or symbols in PROTOMATA-ids.

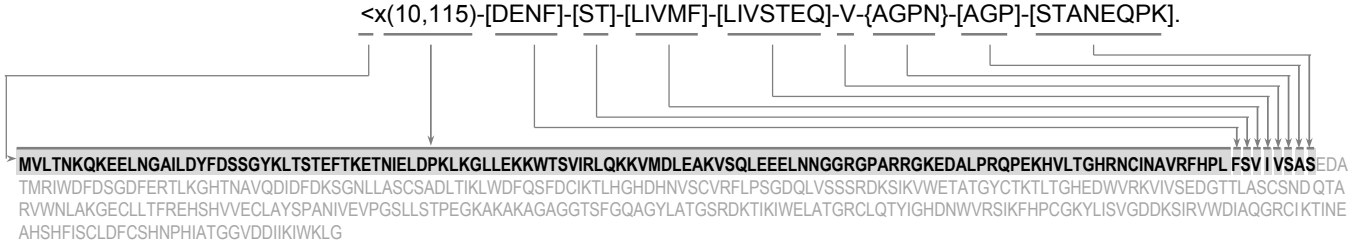


Figure 5: Occurrence of the PROSITE motif *PS00430*: *TonB-dependent receptor proteins signature 1* ($< x(10,115) - [DENF] - [ST] - [LIVMF] - [LIVSTEQ] - V - \{AGPN\} - [AGP] - [STANEQPK]$.) in the *Lissencephaly-1 homolog (D3BUN1)* protein sequence.

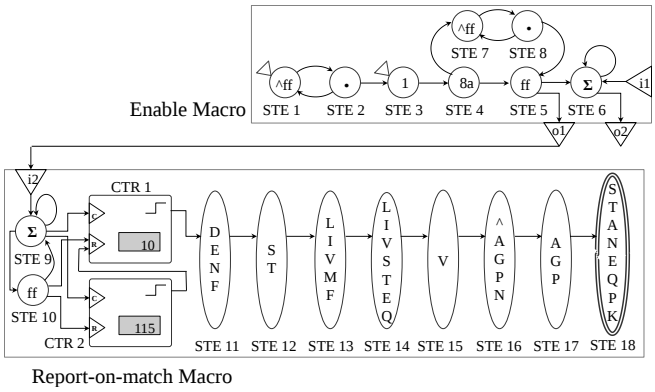


Figure 6: The *Locate-occurrence* automaton.

Locate-occurrence Automaton: The Locate-occurrence automaton for *PS00430* is shown in Figure 6. It has two constituent macros: *Enable* macro and *Report-on-match* macro. The Enable macro is used to allow the search for this motif if its PROTOMATA-id, namely $018a_{16}$ is found in the preamble-sequence.

STE 1 and *STE* 3 of the Enable macro are start-of-data STEs, i.e. they are active for the first symbol in the dataflow. They process the first symbol of the first PROTOMATA-id in the preamble-sequence. Thereafter, *STE* 1 and *STE* 2 ensure that *STE* 3 is active to process the first symbol of every subsequent PROTOMATA-id in the preamble-sequence. If *STE* 3 and *STE* 4 match the first and the second symbol of the id respectively, then *STE* 5 is activated signifying that the search for the motif is enabled. If there are more PROTOMATA-ids in the preamble-sequence, *STE* 7 and *STE* 8 are used to keep *STE* 5 activated to match the delimiter at the end of the preamble-sequence. On encountering the same, it activates the processing elements connected to the outgoing port $o1$. Notice that all instances of this macro for different motifs differ only in the labels of *STE* 3 and *STE* 4. Hence these labels are parameterized.

The occurrence of the *PS00430* is anchored to the beginning of the protein sequence and hence only port $o1$ of the Enable macro is used. For other motifs, where this is not the case, the output port $o2$ is also connected to the input port $i2$ of the Report-on-match macro. Output port $o2$ is driven by *STE* 6 which creates an activation signal for every symbol in the first protein sequence.

If the search for the *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. *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} end-delimiter.

Global-match Automaton: In some use-cases, only the motifs present in all the input protein sequences need to be identified. The Locate-occurrence automaton may be utilized to handle these use-cases by first finding occurrences of all motifs in all protein sequences and then employing a CPU-based algorithm to identify only those motif(s) which are common. However, the rate of output generation of this method may be too high. To overcome this shortcoming, the Global-match automaton was designed to generate a single output event at the end of the dataflow identifying the common motifs only.

The functionality of the Enable Macro in the Global-match automaton is nearly identical to that in the Locate-occurence automaton. The Report-on-match macro is replaced by the *Continue-on-match* macro which does not have any reporting STEs, or *STE* 10 to automatically restart the search for the motif in every protein sequence. Instead, the last STE, namely *STE* 18, is connected to the input ports of a *Repeat* macro through output ports $o3$ and $o4$. Notice that, if the occurrence of the motif needs to be anchored to the end of the sequence, the connection to port $o3$ 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. Only on matching the end-delimiter of the sequence, *STE* 20 reactivates 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 to check for the end-delimiter of the last sequence, then it must have been serially matched in all the sequences. On matching this delimiter, *STE* 20 activates *STE* 21 to generate an output on encountering the 00_{16} symbol at the end of the dataflow.

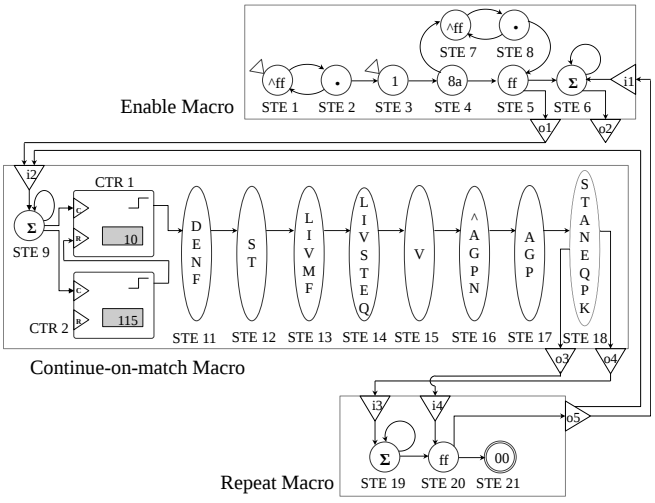


Figure 7: The *Global-match* automaton.

VI. RESULTS

For both of these applications, the ANML-NFA can be defined and compiled before-hand. 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 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 host application comprises 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 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.

A. Fast-SNAP

1) Configuration Overhead: Snort ruleset can be downloaded from the Snort website. Conversion of the active rules to the equivalent ANML-NFAs, as described in Section IV-C1, takes 218 seconds. Patterns for the rules are divided into 11 buckets. Compiling ANML-NFAs for all the buckets takes 716 minutes. If the AP-FSMs for all the buckets are loaded together in a single logical core, they take up 18 AP chips.

2) AP Run-time Estimation: The number of blocks and chips required by different buckets in Fast-SNAP are tabulated in Table I. The AP-FSM for every bucket is loaded in a separate logical core. 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.

The clock divisor (d), discussed in Section II-E, for each bucket is listed next in the table. Logical cores corresponding to the buckets with clock divisor greater than 1 are replicated d times for getting the total matching throughput for the bucket to be 1 Gbps. However, even though $d = 3$ for the *general* bucket, it isn't replicated because replicating it 3 times would require all the resources on the AP board. The number of times each bucket is replicated listed next in the table, followed by the total number of chips required by the bucket on the board and the corresponding matching throughput. If the matching load is balanced across all the buckets, then the total matching throughput is estimated to be $0.3 + (10 \times 1) = 10.3$ Gbps. However, if the load is unbalanced, then the throughput would be constrained by the slowest bucket and can be as low as 0.3 Gbps.

The arrangement of logical cores requires all the 48 chips on the AP board. However, because of the restriction that the minimum width of a logical core should be 2 chips, 16 of those chips are idle. The throughput would therefore improve if the restriction is removed, because of better utilization of the resources on board.

Bucket	#Blocks	#Chips	Clock divisor	Replication count	#Chips required on board	Matching throughput (in Gbps)
general	2,831	16	3	1	16	0.3
http_uri	146	1	3	3	6	1
http_client_body	83	1	2	2	4	1
http_header	75	1	2	2	4	1
http_uri_raw	24	1	2	2	4	1
general_raw	3	1	2	2	4	1
http_cookie	2	1	1	1	2	1
file_data	1	1	1	1	2	1
http_header_raw	1	1	1	1	2	1
http_method	1	1	1	1	2	1
http_stat_code	1	1	1	1	2	1

Table I: Matching throughput supported by different buckets in Fast-SNAP.

B. PROTOMATA

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 V-B1, takes about 1.6 seconds and compilation of these ANML-NFA takes about 20 minutes. All the Locate-occurrence automata for the motifs can be programmed using half the resources on a single AP chip. Similarly, all the Global-match automata can be fit into a single AP chip.

2) AP Run-time Estimation: The run-times of the *ps_scan* application and PROTOMATA are compared in Table II using all the motifs from PROSITE and various proteomes (all proteins from a single organism) from the UniProtKB database [20]. 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 columns 3 and 4 respectively. The two search settings are whether 'Greediness' (reporting the longest possible match starting at a location) is enabled or not; and if the search for 'frequently occurring motifs' is enabled or not.

The run-times of *ps_scan* and PROTOMATA are listed next. 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 should be 56 symbol-cycles. However, we have used 100 cycles for our calculations to account for unforeseen overheads in the pipeline handling. The 'Overall run-time' has been arrived at by taking a maximum of the input and output-handling times and adding 50 milliseconds to account for the load-time.

Database	Organism(s)	#Protein sequences	#Amino acids	Settings		#Motif occurrences	ps_scan	PROTOMATA			Speed up (approx.)
				Greediness	Frequently occurring motifs		run-time (in ms)	Streaming time (in ms)	Output handling time (in ms)	Overall run-time (in ms)	
UniProtKB/ Swiss-Prot	E.coli	4305	1,360,331	enabled	enabled	65,826	213,064	11	52	102	2,088
					disabled	1,644	209,168	11	2	61	3,428
				disabled	enabled	65,826	216,152	11	52	102	2,119
					disabled	1,644	211,922	11	2	61	3,474
	human	20,183	11,336,473	enabled	enabled	630,396	1,238,029	89	493	543	2,279
					disabled	24,332	1,217,581	89	20	139	8,759
				disabled	enabled	630,476	1,220,367	89	493	543	2,279
					disabled	24,385	1,196,148	89	20	139	8,605
	all	479,406	174,899,570	enabled	enabled	9,761,277	23,413,614	1,367	7,626	7,676	3,050
					disabled	755,365	23,632,751	1,367	591	1,417	16,678
				disabled	enabled	9,767,490	24,039,496	1,367	7,631	7,681	3,129
					disabled	761,442	23,623,676	1,367	595	1,417	16,671
UniProtKB/ TrEMBL	E.coli	4,333	1,372,277	enabled	enabled	66,552	216,273	11	52	102	2,120
					disabled	1,655	211,773	11	2	61	3,471
				disabled	enabled	66,552	219,984	11	52	102	2,156
					disabled	1,655	213,158	11	2	61	3,494
	human	67,084	22,252,781	enabled	enabled	1,226,823	3,290,032	174	959	1009	3,260
					disabled	38,812	3,225,670	174	31	224	14,400
				disabled	enabled	1,226,906	3,265,080	174	959	1009	3,235
					disabled	38,865	3,205,392	174	31	224	14,309
	all	18,394,018	6,583,760,868	enabled	enabled	1,681,423,194	980,991,184	51,436	1,313,612	1,313,662	746
					disabled	1,327,068,910	962,727,380	51,436	1,036,773	1,036,823	928
				disabled	enabled	1,690,088,977	962,368,859	51,436	1,320,383	1,320,433	728
					disabled	1,322,376,307	942,714,285	51,436	1,033,107	1,033,157	912

Table II: Comparison of run-times from *ps_scan* and PROTOMATA.

The results discussed above are for a single logical core. If the input size is large enough, it can be broken into parts and streamed to a maximum of 24 logical cores executing in parallel on the AP board. For example, consider the case of all proteomes in the *UniProtKB/Swiss-Prot* database being scanned in a greedy fashion with the search for occurrences of frequently-occurring motifs disabled. With 24 logical cores, the overall run-time for this case is expected to be 108 milliseconds, resulting in an almost 13 times further speedup. For larger input sizes, like all proteomes in the *UniProtKB/TrEMBL* database, a more linear speed-up (almost 23 times) can be achieved, which is approximately a 0.4 million times speedup over *ps_scan*.

C. Evaluation on FPGA

We compared the AP and FPGA for automata processing. In order to provide a fair comparison of the two, we implemented a tool-chain that takes an ANML-NFA as input and automatically generates Verilog code for implementing the given network. We then used Xilinx ISE Design suite v10.1 for performing synthesis, mapping and place-and-route of the generated HDL design onto FPGA hardware. In our FPGA design, we used the well-known one-hot encoding scheme proposed by Sidhu et al. [21] for implementing NFA processing. Further, we applied the single input and multiple outputs optimizations, described by Becchi et al. [22], to our FPGA design. Finally, in order to handle large ANML-NFA exceeding single FPGA capacity, we implemented partitioning schemes that use estimates of the FPGA logic utilization to break the input ANML-NFA into multiple sub-networks that could then be deployed on different FPGA devices.

We performed our experiments on XC5VFX200T Virtex5 device, comprising $30,720$ slices (resulting in $122,880$ flip-flops and LUTs) and have tabulated the results in Table III. 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 (in some cases requiring multiple FPGA devices).

Fast-SNAP network corresponding to the *general* bucket requires partitioning the patterns across 5 FPGA devices. The results for the partitions are tabulated in the first five rows of the table. The networks for the other 10 buckets are compiled together as a single network and the results are tabulated in the sixth row of 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 throughputs of 1.1–1.9 Gbps. Also, as shown in the last row of the table, the whole PROTOMATA network fits the XC5VFX200T device, requiring nearly 30% of the LUT and flip-flop resources and 64% of slices.

Dataset	#LUT	#flip-flip	Slice Utilization	Single-Stream Throughput (Gbps)
general_part1	63,835	84,172	96%	1.306
general_part2	61,448	77,567	85%	1.168
general_part3	83,598	98,287	89%	1.133
general_part4	45,883	56,580	71%	1.921
general_part5	60,635	61,193	75%	1.339
aggregated_buckets	24,735	45,750	94%	1.531
PROTOMATA	33,005	36,278	64%	1.400

Table III: Results of synthesis, mapping and place-and-route on XC5VFX200T for Fast-SNAP and PROTOMATA.

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). The streaming time for the datasets in Table II varies from 7.7 ms (UniPortKB/Swiss-Port, E. Coli) to $37,621$ ms (UniPortKB/TrEMBL, all). 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 $1,308$ reporting states. For the considered datasets, this results in an output processing overhead varying from 0.018 ms (UniPortKB/Swiss-Port, E. Coli, frequently occurring motifs disabled) to $19,310.9$ ms (UniPortKB/TrEMBL, all, frequently occurring motif enabled).

VII. 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 applications which check for 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 these applications show improvements over existing methods and provide a glimpse of how such applications should be programmed using this new accelerator hardware. In addition, the techniques described in this paper should be applicable in the design and analysis of a wide variety of applications on the AP.

Acknowledgments

- 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 CNS-1319748 and CCF-1421765.

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 Developer Portal," http://www.micronautomata.com/, accessed: Apr, 2015.

[5] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *Trans. on Parallel and Distributed Systems*, vol. 99, no. PrePrints, p. 1, 2014.

[6] "Snort," https://www.snort.org/, accessed: Nov, 2014.

[7] 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.

[8] "Perl Compatible Regular Expressions (PCRE) documentation," http://www.pcre.org/current/doc/html/pcre2pattern.html, accessed: Apr, 2015.

[9] "Automata Processor SDK Documentation," http://www.micronautomata.com/documentation/ap_sdk, accessed: Apr, 2015.

[10] J. E. Hopcroft, *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[11] 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, 2010.

[12] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proc. of ANCS*. New York, NY, USA: IEEE/ACM, 2009, pp. 30–39.

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

[14] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for Accelerating SNORT IDS," in *Proc. of ANCS*. IEEE/ACM, 2007, pp. 127–136.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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, 2001, pp. 227–238.

[22] 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.