Implementation of a Content-Scanning Module for an Internet Firewall

James Moscola, John Lockwood, Ronald P. Loui, Michael Pachos
jmm5@arl.wustl.edu, lockwood@arl.wustl.edu, loui@cse.wustl.edu, Michael.Pachos@hp.com
Department of Computer Science and Engineering
Washington University, St. Louis, MO 63130

Abstract

A module has been implemented in Field Programmable Gate Array (FPGA) hardware that scans the content of Internet packets at Gigabit/second rates. All of the packet processing operations are performed using reconfigurable hardware within a single Xilinx Virtex XCV2000E FPGA. A set of layered protocol wrappers is used to parse the headers and payloads of packets for Internet protocol data. A content matching server automatically generates the Finite State Machines (FSMs) to search for regular expressions. The complete system is operated on the Field-programmable Port Extender (FPX) platform.

1 Introduction

Internet firewalls and intrusion detection systems have become critical components of the Internet. They provide protection for Local Area Networks (LANs) by enforcing Access Control Policies (ACPs) for both incoming and outgoing traffic and by alerting a system administrator if any of these policies is broken. Currently, the scope of ACPs usually covers packet headers and exact string matches within the packet payload. These capabilities can be greatly expanded by adding regular expression matching within the packet payload. With regular expressions, a single ACP is capable of enforcing rules which previously took multiple ACPs to enforce (just as one IP address/netmask pair can be used to specify multiple IP addresses). More importantly, regular expressions can give ACPs the ability to enforce rules on mutable content such as that found in many Denial Of Service (DOS) attacks and viruses.

This paper discusses the design and performance of an FPGA-based content-scanning module for an Internet firewall. The module can be (and has been) combined with other modules such as a Content-Addressable Memory (CAM) module to provide the firewall with a rich set of features. The module has many compile-time configuration options, including which regular expressions to scan for, what to do with a packet that contains the specified regular expression, and other options that will be discussed later. An analysis of the required chip resources, frequency, and throughput is also presented.

2 Background

Below is a review of regular expressions along with a description of some previous related work. This section includes a short description of the implementation platform, the Field Programmable Port Extender (FPX) and the layered protocol wrappers, part of the infrastructure.

2.1 Regular Expressions

A regular expression is a pattern that describes a set of strings. The basic building blocks for these patterns consist of individual characters which match themselves such as "a", "b", and "c". Combining characters with meta-characters $(*, |, ?)$ allows more complex regular expressions to be created. If r_1 and r_2 are regular expressions then r_1* matches any string composed of zero or more occurrences of r_1; $r_1?$ matches any string composed of zero or one occurrences of r_1; $r_1|r_2$ matches any string composed of r_1 or r_2; and r_1r_2 matches any string composed of r_1 concatenated with r_2. For instance, a is a regular expression that denotes the singleton set {"a"}, while $a|b$ denotes the set {"a", "b"}. The expression $a*$ denotes the infinite set {"", "a", "aa", "aaa", ...}.

Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)
1082-3409/03 $17.00 © 2003 IEEE

2.2 Regular Expressions in FPGAs

There has been some previous work in the area of string matching on FPGAs. Recent work has been done by Sidhu and Prasanna [1] and by Franklin, Carver and Hutchings [2]. The work by Sidhu and Prasanna was primarily concerned with minimizing the time and space required to construct Nondeterministic Finite Automata (NFAs). They run their NFA construction algorithm in hardware as opposed to software. Their work yielded an exceptional approach to string matching in hardware. Franklin, Carver and Hutchings followed with an analysis of this approach for the large set of expressions found in a Snort database [3].

2.3 Creating Deterministic Finite Automata

In previous work, NFAs were chosen due to the shorter time and smaller space required for *constructing* the automata. In this work however, the time and space required for constructing the automata was not a concern. We were however, concerned with the size of the completed automata. Theoretically, Deterministic Finite Automata (DFAs) can contain up to $O(2^n)$ states, where n is the number of characters in the expression. However, in practice it was found that the number of states required is most often less than or equal to n. In addition to this, DFAs are preferable in stream-by-stream multi-context searching. With a DFA, the machine can only have one active state and thus can be represented compactly if the state of the search needs to be stored for a context switch.

Many existing tools are available for converting regular expressions into DFAs. Among these tools, we chose to use JLex [4], a lexical analyzer generator for Java. As input, JLex takes a regular expression. It outputs a Java file that contains state and transition tables for a minimized DFA that implements the given expression. For this work, the necessary tables are extracted from the Java file using GAWK and converted into a standard VHDL representation of a state machine.

In practice, DFAs are generally compact. To verify this with real data, we extracted SPAM-matching rules from the current version (2.60) of the *SpamAssassin* program. We generated state machines from 358 regular expressions found in this database. The JLex tool was run to generate an NFA and a minimal DFA from each of the regular expressions. It was found that most of the expressions optimized to contain fewer states than the NFA. Figure 1 shows the ratio of the number of states in the JLex-optimized DFA to the number of states in the NFA. Two-thirds of the DFAs were smaller than the NFAs. Only 2.5% of the expressions had more than a 10x expansion and only 0.5% went beyond 40x. A typical *SpamAssassin* expression is:

```
U\.?S\.?(D\.?)?[\    ]*(\$[\     ]*)?([0-9]+,
[0-9]+,[0-9]+|[0-9]+\.[0-9]+\.[0-9]+|[0-9]+
(\.[0-9]+)?[\    ]*milli?on)
```

The output of JLex was:

```
Processing first section -- user code.
Processing second section -- JLex
declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 78 states.
Working on character classes.:
::.:.::.::.............:::..:::
NFA has 15 distinct character classes.
Creating DFA transition table.
Working on DFA states.............................
Minimizing DFA transition table.
24 states after removal of redundant states.
Outputting lexical analyzer code.
```

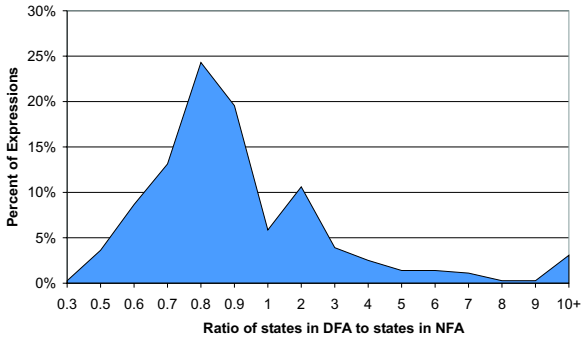

Figure 1. The ratio of the size of the DFA to the NFA for all regular expressions used by the SpamAssassin Program. Note that the majority of the DFAs optimize to be smaller than the NFAs

2.4 Field Programmable Port Extender

The FPX (Figure 2) is a general purpose, reprogrammable platform that performs data processing in FPGA hardware [5, 6, 7]. The FPX extends the operation of the Washington University Gigabit Switch (WUGS) by adding FPGA hardware at the ingress and

Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)
1082-3409/03 $17.00 © 2003 IEEE

egress ports [8]. Data packets can be actively processed in hardware by user-defined, reprogrammable modules as they pass through the device. The hardware-based processing allows the FPX to achieve multi-gigabit per second throughput, even when performing deep processing of the packet payload.

The current version of the FPX contains two FPGAs. One FPGA on the system is called the Reprogrammable Application Device (RAD). It is implemented with a Xilinx Virtex XCV2000E. The second FPGA is called the Network Interface Device (NID). It is implemented with a Xilinx Virtex XCV600E. The FPX also contains two banks of 36-bit wide Zero-Bus-Turnaround Static RAM (ZBT SRAM) and two banks of 64-bit PC-100 Synchronous Dynamic RAM (SDRAM).

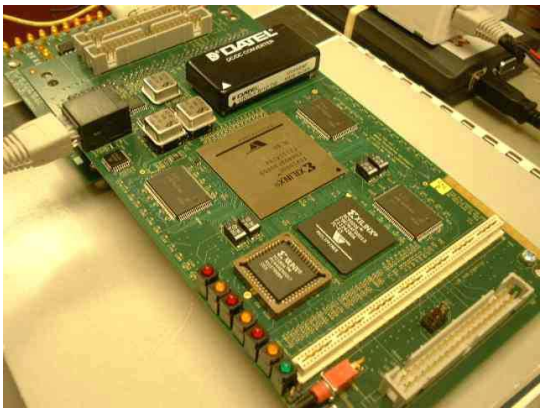

Figure 2. FPX Platform

2.5 Protocol Wrappers

A set of layered protocol wrappers was implemented to simplify the processing of Internet Protocol (IP) packets directly in hardware [9]. They use a layered design and consist of different processing circuits within each layer. At the lowest level, the Cell Processor processes raw cells between network interfaces. At the higher levels, the Frame Processor reassembles and processes variable length frames while the IP Processor processes IP packets. Figure 3 shows the typical layout of a hardware module using the protocol wrappers.

3 Content Scanning in Hardware

The content scanner was implemented as a module on the FPX platform. The scanner utilizes the protocol wrappers to reassemble cells into IP packets and delineate the header and payload fields. When designing

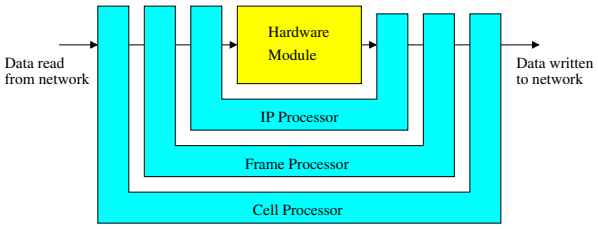

Figure 3. Typical Layout of a Hardware Module Using the Protocol Wrappers

the content scanner, five initial behaviors were desired. These were: (1) the ability to scan every character of every packet's payload for a given set of expressions, (2) the ability to actively drop packets that match a given expression, (3) the ability to generate an alert message identifying which expressions in the given set match, (4) the ability to send an alert message to a log server when a match is detected, and (5) the ability to easily reconfigure the scanner to search for a new set of expressions.

3.1 Generating the Hardware

As new DOS attacks or viruses arrive on the Internet, administrators may create new scanning circuits. To make the content scanner easily configurable, and therefore more useful, a design flow was implemented to automate the creation of the hardware.

The design flow begins with an input specification in common regular expression syntax. The specification contains a list of regular expressions, each with an identification number associated with it which is also programmed into the hardware. The syntax and an example of a single list entry can be seen below:

syntax:

$/expression/prop(id\ number)/$

example:

$/Vi(R|r)u(S|s)/prop(6)/$

Each regular expression in the specification is parsed and sent through JLex to get a representation of the DFA required to match the expression. The JLex representation is subsequently processed and converted into a VHDL representation. Next, a top-level entity is generated to connect all the DFAs with the static components of the circuit. Finally, the design flow proceeds to synthesize, place and route, and program the FPGA.

A web interface was created to simplify designing new circuits. The interface allows expressions to be added, edited, or deleted from a list of available ex-

Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)
1082-3409/03 $17.00 © 2003 IEEE

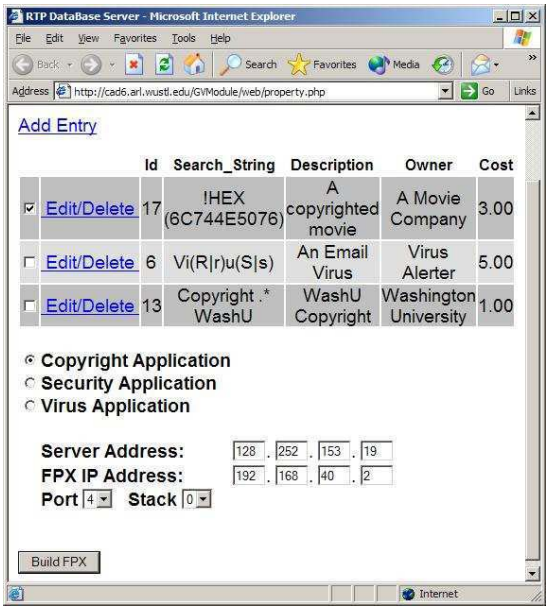

Figure 4. Web Interface for Creating Circuit

pressions. A screenshot of the interface is shown in Figure 4. From the web, an administrator can select any subset of the available expressions to be programmed into the hardware. The resulting circuit can be used for several applications, such as copyright protection, virus protection or security. For network security applications, the Internet address of a server is specified to determine where alarm messages should be delivered. For networks that contain multiple FPX devices, the FPX IP address specifies which FPX should be reprogrammed.

For example, if the first and second boxes of Figure 4 are checked, then the FPX will be programmed to scan for two regular expressions. First, it will scan packets for the hex string *6C744E5076* which appears in a video stream. Additionally, it will scan packets for the regular expression *Vi(R|r)u(S|s)*. If a packet containing the *Vi(R|r)u(S|s)* signature is found, then an alert message is sent to the server address *128.252.153.19*. Once the "Build FPX" button is pressed, the design scripts proceed to synthesize, place and route, and reprogram the FPX over the network.

3.2 Hardware Implementation

This section describes the design of the content scanning circuit. It is broken down into three parts: (1) *Receiving Packets*, (2) *Processing Packets*, and (3) *Outputting Packets*. Each of these three operations is controlled independently of the other two. All three operations run in parallel. A block diagram of the implementation can be seen in Figure 5.

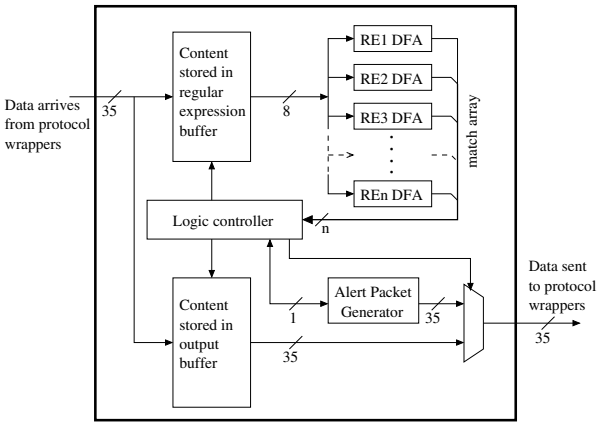

Figure 5. Block diagram of Content Scanner

3.2.1 Receiving Packets

Data enter the circuit in 32-bit chunks after passing through the protocol wrappers. The protocol wrappers assert control signals to indicate the beginning of an AAL5 frame, the beginning of an IP packet payload, and the end of an AAL5 frame. There is also a data enable signal to indicate the presence of a valid 32-bit data word on the incoming bus. Every valid data word, along with the three control signals, is written to two parallel 512x35 dual-port memory buffers. By using two identical buffers, it is possible to read newer packets for processing while older packets (that are not being dropped) are read for output. This could be achieved with a single tri-port memory buffer if available.

3.2.2 Processing Packets

Once data are available in the input buffer (a packet has started arriving), the circuit can begin processing the packet. To process a packet with the content scanner, a counter is used to address one of the 512x35 memory buffers. On each clock tick, one character (8-bits) is read from the memory buffer and sent to each of the regular expression DFAs. All of the DFAs search in parallel. Each DFA maintains a 1-bit *match* signal which is asserted high when a match is found within the packet that is being processed. When the counter reaches the end of the packet, one or more of the following can occur:

(1) if the *match* signals from *all* of the DFAs indicate no match was found, then a pointer to the packet is

Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)
1082-3409/03 $17.00 © 2003 IEEE

inserted into a queue for output.

(2) if any of the *match* signals indicate a match was found but do not require dropping the packet, then a pointer to the packet is inserted into a queue for output.

(3) if one or more of the *match* signals indicates a match was found that requires dropping the packet, then a pointer to the packet is *not* inserted into a queue for output, hence the packet is dropped.

(4) if one or more of the *match* signals indicates a match was found that requires an alert message to be sent, a special pointer is inserted into a queue which indicates an alert message should be output. The special pointer contains a bit array that indicates which DFAs found a match. It should be noted that if a match is found which requires an alert message but does not require dropping the packet, two pointers (one for the original packet and one for the alert message packet) are inserted into the queue for output.

3.2.3 Outputting Packets

A packet is output from the content scanner whenever there is an available pointer in the output queue. Each pointer that is dequeued can be either for a regular packet or for an alert packet. In the case of a regular packet, a counter is assigned the value of the pointer and used to address the 512x35 output memory. The packet is then output 32-bits per clock cycle until the end of the packet is detected. The most significant 3 bits of the output memory are used to recreate the necessary control signals for communicating with the protocol wrappers. When an alert packet pointer is dequeued, a UDP alert packet has to be generated since one does not already exist. The alert packet is addressed to a predetermined log server (specified at compile time but also runtime reconfigurable). The payload of the alert packet contains the source and destination IP address of the packet that caused the alert, along with the identification numbers of *all* of the regular expressions that matched in the original packet. Figure 6 shows the layout of an alert packet.

3.2.4 Increasing Throughput via Parallel Scan Engines

As mentioned in sections 3.2.1 and 3.2.2, data enter the content scanner at a rate of 32-bits per clock cycle, and are processed at 8-bits per clock cycle. As a result, the content scanner can only process data at one-quarter of the maximum input rate. In order to process data at the full input rate, four parallel content scanners are arranged as shown in Figure 7. Arriving packets are dispatched to an available content scanner in a round-robin fashion. With four parallel scanners, the circuit

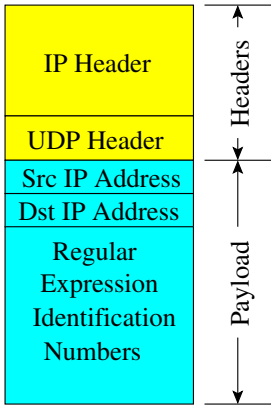

Figure 6. Layout of an Alert Packet

is now capable of scanning 32-bits per clock cycle.

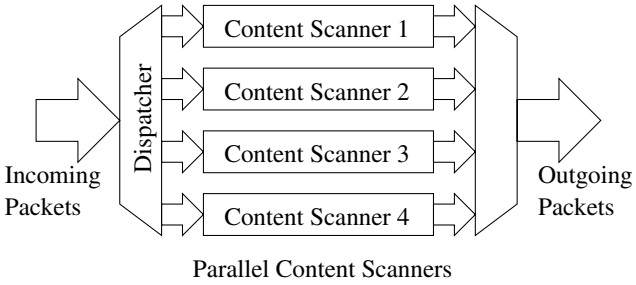

Figure 7. Arrangement of parallel scanners

4 Results

Several different versions of the content scanner were synthesized with the protocol wrappers into the RAD of the FPX. The regular expression set for each of the content scanners consisted of 21 regular expressions. The expressions chosen were aimed primarily at dropping SPAM. For example, *"Get Rich Quick"* and *"(L|l)imited (T|t)ime (O|o)ffer"* were among the expressions in the set. On average, each regular expression was 20 characters long. Note that these were simpler expressions than those found in *SpamAssassin*.

Each of the circuits was tested in the lab using NCHARGE [10] for initial testing. NCHARGE allowed single packets to be sent to the content scanner for easier debugging. Later stages of testing were conducted using real Internet traffic via web browsers, email clients, and FTP clients. The configuration of the lab setup is shown in Figure 8. This type of testing allowed placement of pseudo-viruses and other content

Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)
1082-3409/03 $17.00 © 2003 IEEE

on the Internet to verify detection and potential dropping by the scanning module.

The following sub-sections describe the device utilization and throughput of the content scanner circuits on a Xilinx Virtex XCV2000E part.

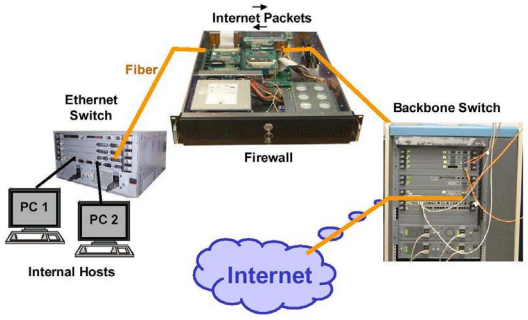

Figure 8. Laboratory Test Layout

4.1 Device Utilization

Device utilization for three different circuits is shown in Tables 1, 2, and 3. Table 1 shows the device utilization for a circuit containing only the infrastructure and the protocol wrappers. These values represent the overhead of the packet processing done by the protocol wrappers. Table 2 details the device utilization for a single content scanner with the protocol wrappers and all the necessary infrastructure. Table 3 shows the device utilization for the four parallel content scanners shown in Figure 7 with the protocol wrappers and necessary infrastructure. The chart in Figure 9 helps to illustrate the relative sizes of each of the circuits.

As mentioned earlier, the simplified DFAs used on average of n states for an n length regular expression. This translated into the hardware as using on average 1 flip-flop per character (minus the overhead associated with the infrastructure, the protocol wrappers, and the controller).

Table 1. Device Utilization for Infrastructure and Protocol Wrappers

Resources	Virtex XCV2000E Device Utilization	Utilization Percentage
Logic Slices	3263 out of 19200	16%
Flip Flops	3611 out of 38400	9%
Block RAMs	11 out of 160	6%
External IOBs	142 out of 512	27%

Table 2. Device Utilization for Content Scanning Module with Single Search Engine

Resources	Virtex XCV2000E Device Utilization	Utilization Percentage
Logic Slices	4422 out of 19200	23%
Flip Flops	4547 out of 38400	11%
Block RAMs	22 out of 160	13%
External IOBs	142 out of 512	27%

Table 3. Device Utilization for Content Scanning Module with Quad Search Engines

Resources	Virtex XCV2000E Device Utilization	Utilization Percentage
Logic Slices	7330 out of 19200	38%
Flip Flops	6628 out of 38400	17%
Block RAMs	55 out of 160	34%
External IOBs	142 out of 512	27%

4.2 Throughput

The single-scanner regular expression circuit with the protocol wrappers currently places and routes at 37 MHz. The critical path of the circuit was found to be the fanout of the 8-bit character lines to each of the DFAs. The quad-scanner circuit has similar results; it also places and routes at 37 MHz. Given that each scanner can process 8-bits of data per cycle, the calculated throughput of the single-scanner circuit is $8 \times 37MHz = 296$ Megabits/second. By running four content scanners in parallel, the circuit can reach $4 \times 8 \times 37MHz = 1.184$ Gigabits/second.

When limiting the scanner to only several DFAs (thus minimizing the fanout bottleneck), the circuit was capable of achieving frequencies in the range of 75-80 MHz. At these frequencies, the circuit is capable of exceeding 2.5 Gigabits/second.

5 Conclusion

This paper has described an implementation of a content scanning module for an Internet firewall. The module is capable of passively reviewing packet payloads for a match, actively dropping packets that contain a match, and generating alert packets to notify administrators of a match. The content scanner was implemented on the FPX and tested using real Internet traffic on the WUGS. The scanner is capable of operating at speeds of 1.184 Gigabits/second for twenty-

Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)
1082-3409/03 $17.00 © 2003 IEEE

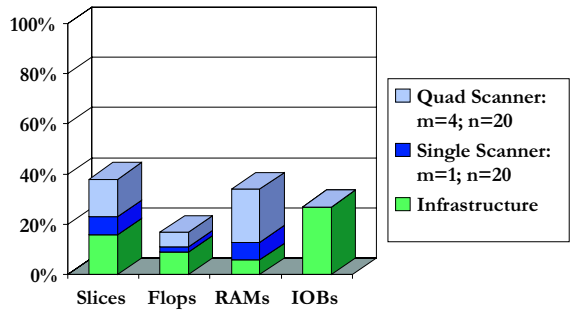

Figure 9. Device Utilization

one ∼20-character regular expressions, and exceeding speeds of 2.5 Gigabits/second for smaller numbers of similar regular expressions.

6 Future Work

Currently, the bottleneck in the system is the fanout associated with sending 8-bits of data to all of the parallel DFAs. To improve the timing, a tree structure (as described and implemented in [2]) can be used to distribute the data to all of the DFAs to minimize the propagation delay.

Currently, the content scanner is designed to support several different behaviors as described in section 3. However, the behaviors such as sending an alert message are not on an expression-by-expression basis. For example, if the content scanner is compiled to send an alert message, it will send that alert message for all the regular expressions in the circuit. It is more desirable to have the action rules based on which expression matched. Doing this would allow alert packets to be sent for some regular expressions in the circuit and not for others. An enhanced version of awk's pattern-rule syntax (but with reduced instruction set) has been defined which is suitable for this task. In a related publication, we report on sed-like processing on the FPX.

Finally, the content scanner currently looks for matches on a packet-by-packet basis. This means that if a string that should cause a match spans multiple packets, it will be missed by the content scanner. We look to improve this behavior by utilizing the TCP-Splitter [11] to process data on a stream-by-stream basis. This entails augmenting the content scanner to a multi-context design that maintains a match context for each available flow and switches contexts based on the stream that is currently being presented by the TCP-Splitter.

7 Acknowledgements

The authors would like to thank Todd Sproull for his work on the NCHARGE control software and his never-ending help in the lab, David Lim for his work on the Network Interface Device, and Michael Pachos for his initial work on the regular expression search engine. This work on the FPX platform has been supported by grants from the National Science Foundation (ANI-0096052) and work on the string matching circuit has been supported by a grant from Global Velocity.

References

[1] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Rohnert Park, CA, USA), Apr. 2001.

[2] R. Franklin, D. Carver, and B. L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Napa, CA, USA), Apr. 2002.

[3] M. Roesch, "Snort - lightweight intrusion detection for networks, booktitle = 13th Administration Conference, LISA'99, address = Seattle, WA, month = nov, year = 1999,"

[4] E. Berk and C. Ananian, "Jlex: A lexical analyzer generator for java." Online: `http://www.cs.-princeton.edu/~appel/modern/java/JLex/`.

[5] J. W. Lockwood, "An open platform for development of network processing modules in reprogrammable hardware," in *IEC DesignCon'01*, (Santa Clara, CA), pp. WB–19, Jan. 2001.

[6] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, (Monterey, CA, USA), pp. 87–93, Feb. 2001.

[7] "Field Programmable Port Extender Homepage." Online: `http://www.arl.wustl.edu-/arl/projects/fpx/`, Aug. 2000.

[8] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke, "Design of a Gigabit ATM Switch," in *Infocom 97*, Mar. 1997.

[9] F. Braun, J. W. Lockwood, and M. Waldvogel, "Layered Protocol Wrappers for Internet Packet Processing in Reconfigurable Hardware," *IEEE Micro*, vol. Volume 22, pp. 66–74, Feb. 2002.

[10] T. Sproull, J. W. Lockwood, and D. E. Taylor, "Control and Configuration Software for a Reconfigurable Networking Hardware Platform," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Napa, CA, USA), Apr. 2002.

[11] D. V. Schuehler and J. Lockwood, "TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware," in *Proceeding of Hot Interconnects 10 (HotI-10)*, (Stanford, CA, USA), Aug. 2002.

Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)
1082-3409/03 $17.00 © 2003 IEEE