FPsed: A Streaming Content Search-and-Replace Module for an Internet Firewall

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

Abstract

A module has been implemented in Field Programmable Gate Array (FPGA) hardware that is able to perform regular expression search-and-replace operations on 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, compiles, synthesizes, and programs the module into the Field-programmable Port Extender (FPX) platform.

1 Introduction

As the speed of networks continues to increase, it becomes increasingly difficult to monitor content sent over the Internet with software-based processing techniques. Hardware-based processing is needed to keep pace with modern high-performance networks. To achieve high network performance, hardware devices known as Field Programmable Gate Arrays (FPGAs) have been used. FPGAs offer a method for implementing functions in hardware in a way that allows the circuit to be modified. Hardware-based search (FPgrep) [1] and hardware-based search-and-replace systems (FPsed) have been developed that can scan and modify packets as they stream through the network.

FPgrep and FPsed utilize regular expressions (REs) to specify a set of string patterns that may be searched for within the payload of a packet as it passes through a network. The RE patterns can range in complexity from a simple single character string to a string consisting of multiple wildcards.

By combining the power of REs and the flexibility of FPGAs on the Field Programmable Port Extender (FPX) [2, 3], the FPgrep and FPsed packet payload processors may be used to process packet contents on high-speed networks.

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 Field Programmable Port Extender (FPX) implementation platform and the layered protocol wrappers.

2.1 Regular Expressions

A regular expression (RE) is a pattern that describes a set of strings. The basic building blocks for these patterns consist of individual characters that match themselves such as "a", "b", and "c". Combining characters with meta-characters $(*, |, ?)$ allows more complex REs to be created. If r_1 and r_2 are REs 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 RE that denotes the singleton set {"a"}, while "$a|b$" denotes the set {"a","b"}. The expression "$a*$" denotes the infinite set {"","a","aa","aaa",...}.

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 [4] as well as by Franklin, Carver, and Hutchings [5]. The work by Sidhu and Prasanna was primarily concerned with minimizing the

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Kelvin Smith Library @ CASE. Downloaded on July 24,2021 at 16:43:52 UTC from IEEE Xplore. Restrictions apply.

time and space required to *construct* Nondeterministic Finite Automata (NFAs). This is because they run their NFA construction algorithm in hardware as opposed to software. Franklin, Carver, and Hutchings followed with an analysis of this approach for the large set of expressions found in a Snort database [6].

2.3 Field Programmable Port Extender

The FPX (Figure 1) is a general purpose, reprogrammable platform that performs data processing in FPGA hardware [2, 3, 7, 8]. The FPX extends the operation of the Washington University Gigabit Switch (WUGS) by adding FPGA hardware at the ingress and egress ports of a high-speed Internet router [9, 10]. 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 processing of packet payloads.

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).

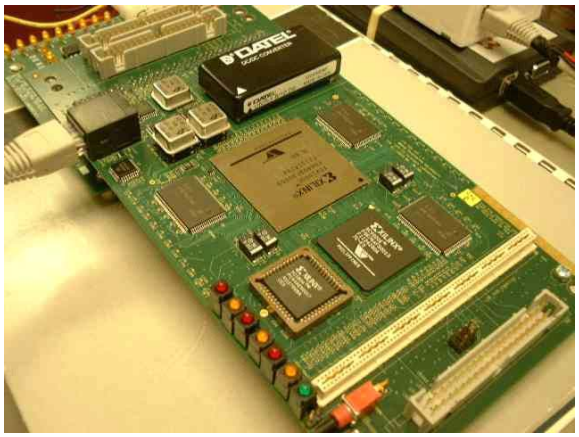

Figure 1. FPX platform

2.4 Protocol Wrappers

A set of layered protocol wrappers was implemented to simplify the processing of Internet Protocol (IP) packets directly in hardware [11]. They use a layered design and consist of different processing circuits within each layer. At the lowest level, the Cell Processor operates on small fixed length cells that flow between network interfaces. At the higher levels, the Frame Processor reassembles and processes variable length frames while the IP Processor processes IP packets. Figure 2 shows the typical layout of a hardware module using the protocol wrappers.

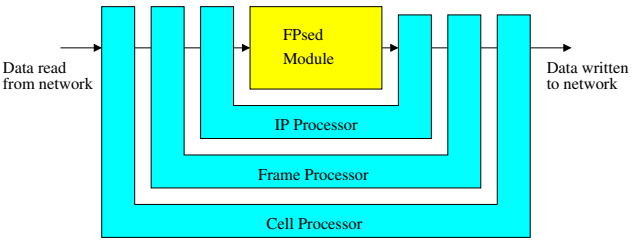

Figure 2. FPsed module in protocol wrappers

3 RE Search-and-Replace

The streaming content editor, FPsed, was implemented as a module on the FPX platform. The content editor has the ability to perform RE search-and-replace operations on network packets passing through the module; this is similar to the substitute command of UNIX's stream-editor utility (sed). The scanner utilizes the protocol wrappers to process IP packets and delineate the header and payload fields.

3.1 RE State Machine Terminology

Before beginning a more in-depth discussion we define some of the terminology used in this paper to describe the matching of REs using finite state machines.

Start: The transition of a machine from the *idle* (initial) state to a non-*idle* state.

Accept: The machine has *accepted* the substring if the machine has determined that the substring is a member of the language defined by the RE. Once a machine *accepts* a substring it must *match* on the substring some time in the future.

Match: The machine has determined the boundaries of the complete substring.

Running: The machine has *started* but not yet *failed*. The machine may or may not be in an *accepting* state.

Reset/Fail: The machine was *running* and a character read caused the substring to no longer be a member

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Kelvin Smith Library @ CASE. Downloaded on July 24,2021 at 16:43:52 UTC from IEEE Xplore. Restrictions apply.

of the language defined by the RE. If the substring was previously *accepted* then a *match* was created over a portion of the substring up to, but not including the character that caused the machine to *reset*.

Idle: The machine is not *running*.

3.2 Searching

The search function, FPgrep, was implemented to search packet payloads for substrings that belong to the language defined by the RE. When FPgrep *matches* a substring in a packet it transmits information about the packet to a monitoring host system. The information sent for network intrusion detection functions specifies the content that was found and the sender's and receiver's IP addresses.

The search runs in linear time (proportional to packet size) $O(n)$ (where n is the number of bytes in a packet) and in constant space. That is, there is never a need to examine a character more than once and the amount of hardware is proportional to the size of the RE. Approximately one flip-flop is required per character.

When a RE search is requested, a "$.*$" is prepended to the beginning of the original RE. It is natural to think about it this way since searching involves finding any number of characters followed by a *matching* substring. The prepended "$.*$" allows the machine to recognize a *matching* substring anywhere in the record [12].

If the "$.*$" is not prepended, then there are situations in which a substring that should be *matched* by the machine is missed. This situation arises if a machine M enters the *running* state when it encounters character c_i and then transitions to the *failed/reset* state when it encounters c_{i+n}. If the machine simply continues reading from the next character, c_{i+n+1}, it would not detect a substring whose first character is in the range c_{i+1} to c_{i+n}.

A small example that illustrates this problem. Assume we are searching for "ARL". If both "ARL" and "$.*ARL$" are converted into DFAs, two functionally different machines (DFA_1 and DFA_2 respectively) are produced. These DFAs can be seen in Figure 3 and Figure 4.

If the input to the machines is "$A_1R_2A_3R_4L_5$", then DFA_1 will recognize that "A_1" followed by "R_2" is part of the language. When "A_3" is input, the machine *fails* and thus transitions to the *idle* state. It is clear that machine will not find the substring "$A_3R_4L_5$", since when the next character "R_4" is input into DFA_1 it remains in the *idle* state. On the other hand DFA_2 does operate correctly and finds the substring.

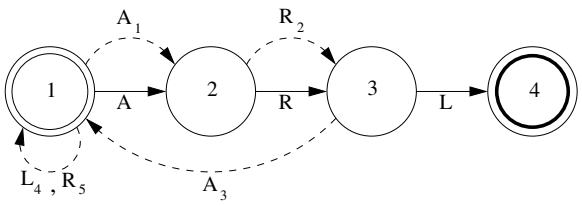

Figure 3. DFA_1 **for "**ARL**" does not** *accept*

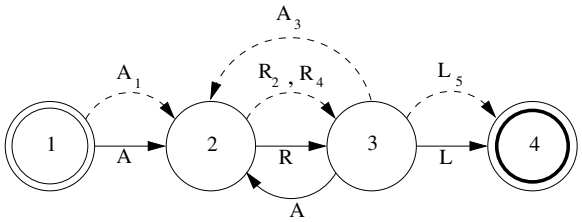

Figure 4. DFA_2 **for "**$.*ARL$**" does** *accept*

3.3 Search and Replace

The FPsed module performs both replacement and global replacement operations on packet payloads. The task of string replacement of a RE is not as straightforward or efficient as searching. String replacement requires that the machine do more than simply determine the presence of *matching* substrings in a record. The machine must also determine the position of the first and last character of all complete substrings that are *matched* by the machine. It is this requirement that causes the task of RE search-and-replace to be more complicated and less efficient than a simple search.

Searching for the complete substring is logical when the goal is to replace that substring. Consider the task of replacing every occurrence of a certain hexadecimal string associated with a computer virus "$37F43(B + |7*)$" with the text "*Virus Pattern Detected*". For the input string "$3_1 7_2 F_3 4_4 3_5 B_6 B_7 B_8$", the substring could be replaced from the point where the machine *starts running*, "3_1", to the point where the substring is *accepted* "B_6". But this would allow a portion of the virus to remain in the content stream. In most situations, it is preferable to replace only complete substrings.

To search for complete substrings a "$.*$" can no longer be prepended to a RE before it is converted to a FSM. This is because prepending "$.*$" would make it difficult to determine the first character of the *matched* substring. We do not believe that there is a general, easily automatable, method for determining the position of the first character in a *matching* substring when a "$.*$" is prepended. For instance, to replace all occurrences of "ARL", a state machine like Figure 5 would

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Kelvin Smith Library @ CASE. Downloaded on July 24,2021 at 16:43:52 UTC from IEEE Xplore. Restrictions apply.

be generated after prepending a ".*".

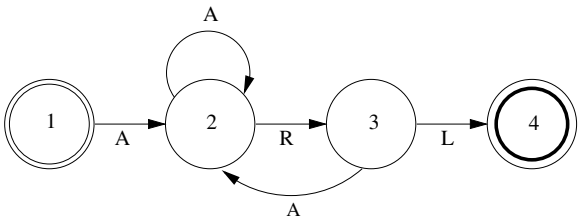

Figure 5. DFA for ".*ARL"

If the string "$A_1A_2A_3A_4R_5L_6X_7$" is input to the machine it would begin *running* when it encounters "A_1" and *matches* when "X_7" is read. This would have the effect of replacing "$A_1A_2A_3A_4R_5L_6$" when the intention is to replace "$A_4R_5L_6$". One could counter that this would not be a problem if the machine simply kept a count of how many characters were read after entering state 2, resetting the count every time it entered state 2. For this particular example that would be true, but there are examples that do not have such a simple and formulaic solution.

A slightly more complicated example is ".*AR*L", seen in Figure 6. Once again there is a solution to the problem, but this time it is more complicated. In this situation, the machine would again have to keep track of the number of characters read after entering state 2. However, this time the machine should only reset its count if the input causing the transition to state 2 is not an "R".

An even more complicated example is the machine for ".*A(AR)*L" in Figure 7. It is left to the reader to see that the machine becomes quite complex.

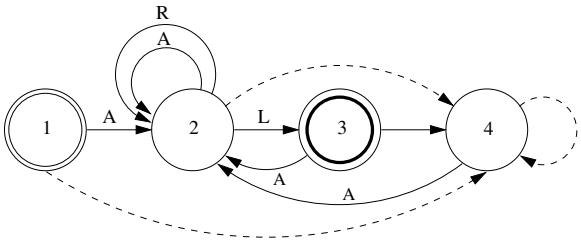

Figure 6. DFA for ".*AR*L"

It is therefore difficult to devise a general method for determining the start of the string. One can always find a more complicated RE that would require the addition of more rules to the method.

It has been shown that prepending a ".*" to REs that are to be replaced is clearly not a viable solution as it was with searching. In the previous section it was shown that if a ".*" is not prepended, the searching

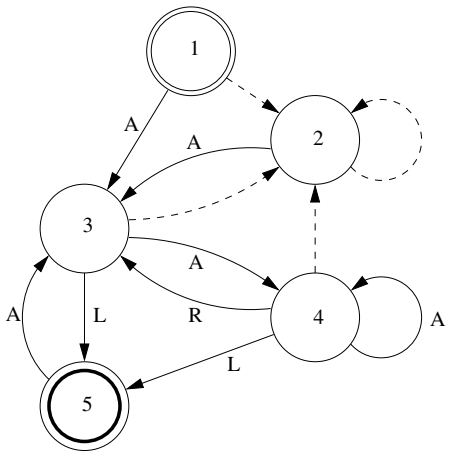

Figure 7. DFA for ".*A(AR)*L"

functionality will not have the correct semantics if each character is read only once. Because of this, FPsed must employ a different technique for dealing with the problem of finding all complete substrings that *match* a particular RE. A solution to this problem is to use a *brute force* method of searching.

3.3.1 Brute Force Method

A brute force technique checks all characters of the input to determine if they could be the beginning of a substring that *matches* the RE.

The worst case running time occurs when every input character is a possible *starting* position, and all but the last character of the input *matches* the RE. For example, if the machine is searching for "$A*B$" and the input is "$A_0A_1 \ldots A_{n-1}$". The machine must make $O(nm)$ comparisons, where n is the string length and m is the pattern length, to determine that the pattern does not occur in the record.

The worst case condition is unlikely to appear when searching for English language expressions, but it is less rare when searching binary text. Davies and Bowsher [13] examined the efficiency of the backtracking technique when searching strings from the English language and binary strings. Their experiments involved keeping track of the number of references to an input string divided by the number of characters occurring before the *matched* substring (the index position of the pattern minus one) thus obtaining the number of inspected characters in the text string per character passed. The results of their experiments showed that when searching English text, the backtracking method referenced the text string 1 to 1.1 times per character passed. When searching binary text for an expression of length six or greater, approximately 2 characters

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Kelvin Smith Library @ CASE. Downloaded on July 24,2021 at 16:43:52 UTC from IEEE Xplore. Restrictions apply.

were inspected for every character passed.

4 Hardware Design and Operation

The FPsed hardware module has been implemented to perform a brute force method for search-and-replace. The hardware consists of several components, all of which are controlled by the logic controller. A block diagram for the module can be seen in Figure 8.

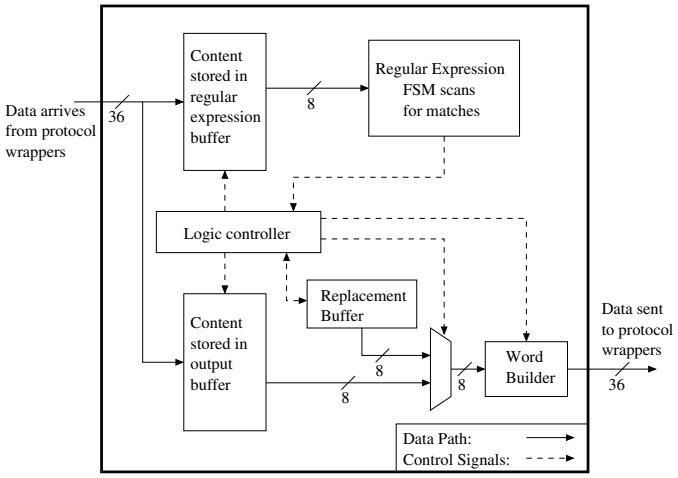

Figure 8. FPsed search-and-replace module

4.1 Logic Controller

The logic controller is the most complex entity in the design. It controls parallel, dual-ported memory buffers, the regular expression machine, the replacement buffer and the word builder using control signals that it generates. Like the FPgrep controller, the FPsed logic controller has three main phases that operate in parallel: (1) *Receiving Packets*, (2) Processing Packets, and (3) Outputting Packets.

Receiving Packets: As packets come into the module from the protocol wrappers, they are first written into the two parallel 36-bit wide dual-port memory buffers. The lower 32 bits written into the memory buffers are the incoming data. The upper four bits are used to store the four control signals from the protocol wrappers (start of frame, start of IP headers, start of IP payload, and end of frame). The write-side address lines are shared by the two memory buffers, as is the write-enable signal. This ensures that the contents of the two buffers are identical. The address lines are controlled by a simple counter. Each time a new word is written to memory, the counter is incremented to address the next location.

Processing Packets: The bytes that are input to the RE state machine come from the read side of one of the dual-port memories. Only the payload data are sent through the RE state machine for processing. To address the memory, a counter is used to step through the memory one byte at a time. The bytes are sent to the RE state machine and processed. Control signals from the RE state machine tell the controller which address to read next using the following rules:

1. If no control signals are returning from the RE state machine (i.e. not *running*), the controller increments the byte address by one.
2. If while *running* the controller receives an *accepting* signal from the RE state machine, it stores the address of the byte that created the *match* and to increments the byte address by one.
3. If the controller receives a *resetting* signal from the RE state machine, one of two things can happen:
   (a) If while *running*, the controller did not receive an *accepting* signal, it backs up the byte address and begins processing data immediately proceeding the byte that previously *started* the machine.
   (b) If an *accepting* signal was received (a *match* was found), it backs up the byte address and begins processing data immediately proceeding the byte that caused the machine to *match*. Also, to remember where the *match* has occurred, the *starting* byte address and the ending byte address of the string are stored in two fifos (the *start_fifo* and the *end_fifo*).

When the end of a packet is reached, the controller sends a reset signal to the RE state machine to *reset* it.

Outputting Packets: The output process examines bytes from the read port of the second dual-port memory. It can only output data that are completely done being processed by the RE state machine (none of the RE state machine pointers reference the data). As the output process steps through the available bytes, it checks the previously mentioned *start_fifo* and does the following:

1. If the current output address is not stored in the *start_fifo*, then the byte is sent to the word builder and the byte address is incremented by one.
2. If the current output address is stored in the *start_fifo*, then the following occurs:
   (a) The byte is not sent to the word builder. Instead a signal is sent to the replacement buffer to begin outputting a replacement.
   (b) The byte address for the output process is assigned the value stored in the *end_fifo*.

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Kelvin Smith Library @ CASE. Downloaded on July 24,2021 at 16:43:52 UTC from IEEE Xplore. Restrictions apply.

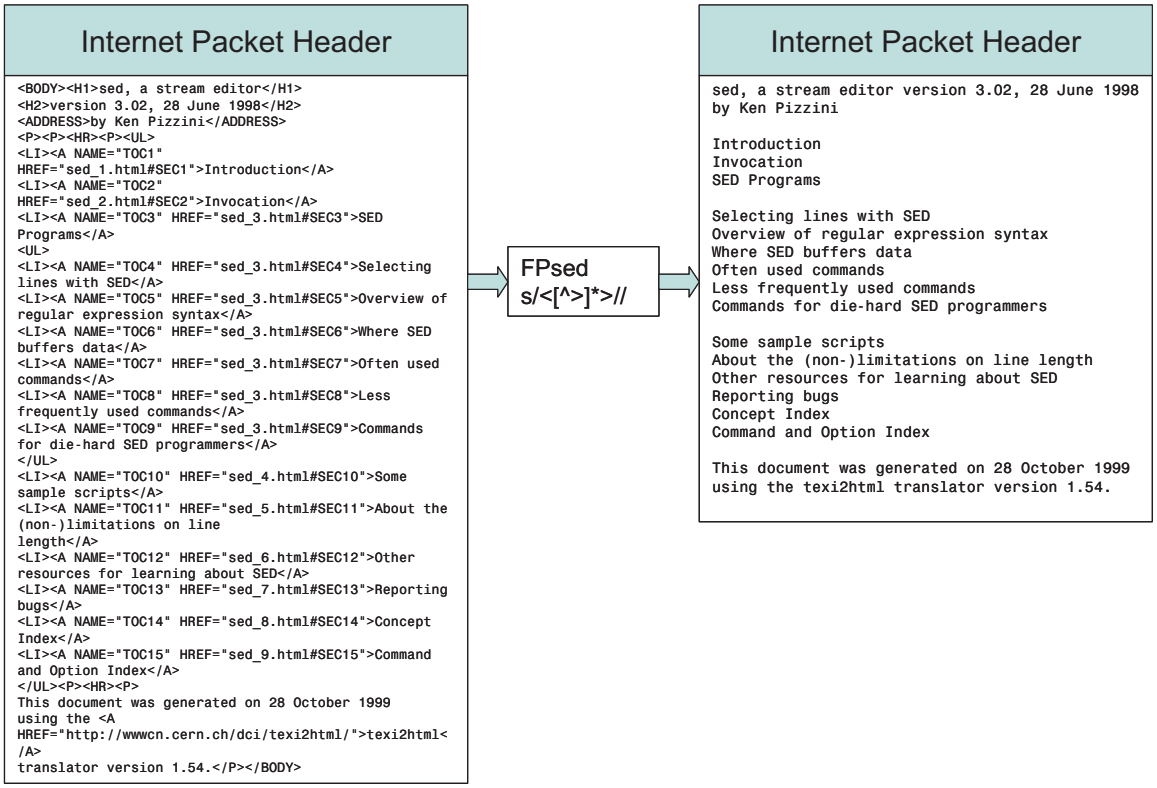

Figure 9. Example of FPsed used to strip HTML tags from a packet payload

(c) The output process waits for a done signal from the replacement buffer before processing additional bytes.

Finally, as data are read from the memory and output, the four control bits are used to assert the appropriate signals back to the protocol wrappers.

4.2 Generating the Hardware

The system was designed to be easily reconfigurable when new search terms were desired. To accomplish this, a complete design flow from specification to hardware bitstream was implemented. The design flow begins with an input specification in common RE syntax. The specification contains a list of REs and their corresponding replacement strings. The syntax and a couple examples can be seen below:

syntax:

$s/expression/replacementstring/$

Example of stripping out HTML tags:

$s/<[^>]*>//$

Example of a profanity blocker:

$s/(P|p)rofa(N|n)ity/*******/$

Each RE in the specification is parsed and sent through JLex [14] to get a representation of the DFA required to match the RE. The JLex representation is subsequently processed and converted into a VHDL representation. Next, a top-level entity is generated to connect the DFAs with the static components of the circuit. Finally, the design flow proceeds to synthesize, place and route, and program the FPGA.

5 Results

Several versions of the content editor were synthesized with the protocol wrappers into the RAD of the FPX. One module was designed to replace computer viruses as they traversed across a streaming UDP-based Internet connection. Another module was developed to remove profanity from packet payloads and were tested in the lab using a UDP-based chat client. A third module was developed to remove HTML tags from text. An example of the transformation performed by the HTML filter is shown in Figure 9.

The following sections describe the device utilization and estimated throughput of the content-editing modules on a Xilinx Virtex XCV2000E-8 part.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Kelvin Smith Library @ CASE. Downloaded on July 24,2021 at 16:43:52 UTC from IEEE Xplore. Restrictions apply.

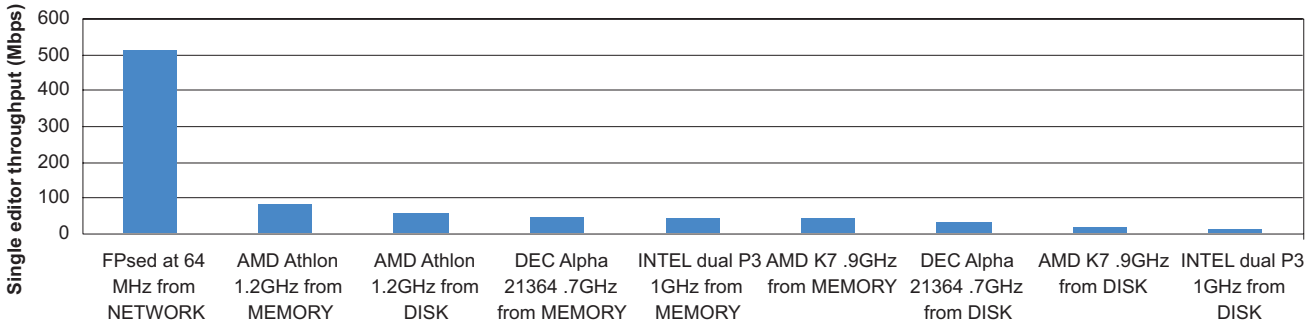

Figure 10. Comparison of hardware and software throughput

5.1 Device Utilization

The utilization of FPGA resources for two different modules is shown in Tables 1 and 2. Table 1 shows the device utilization for a module 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 editor with the protocol wrappers and all the necessary infrastructure.

Table 1. Device utilization for infrastructure and protocol wrappers

Resources	XCV2000E Utilization	Utilization Percentage
Logic Slices	2399 out of 19200	12%
Flip Flops	2870 out of 38400	7%
Block RAMs	19 out of 160	11%
External IOBs	142 out of 512	27%

Table 2. Device utilization for FPsed module with single content editor

Resources	XCV2000E Utilization	Utilization Percentage
Logic Slices	2922 out of 19200	15%
Flip Flops	3223 out of 38400	8%
Block RAMs	21 out of 160	13%
External IOBs	142 out of 512	27%

5.2 Throughput

A single content-editing RE module for the HTML filter was synthesized for the Virtex XCV2000E-8 FPGA that implements the RAD on the FPX platform. The FPGA was placed and routed using the Xilinx backend tools to run at 64 MHz. This provides a throughput of 64 MHz * 8 bits/byte = 512 Mbps.

An experiment was run to mimic FPsed's functionality with software running on four different computers. One computer, a dual Intel Pentium 3 operating at 1 GHz running a Linux 2.2 kernel, achieved 13.7 Mbps when the sed program (version 3.02) read data from disk. To ensure that disk I/O was not a bottleneck, the same program was run completely from memory and achieved 32.72 Mbps. This is approximately 16x slower than the FPsed hardware. Another computer, an Alpha 21364 operating at 667 MHz running Linux kernel 2.4, was able to search and replace data at 36 Mbps when the input was read from disk, and 50.4 Mbps when the input was run completely from memory. The fastest computers were 10x slower than the hardware. Throughput results for all four computers are shown in Figure 10.

Further increase in the throughput on the FPX platform was achieved by instantiating multiple content editors in parallel and dispatching incoming packets to an available editor. With 4 parallel editors, the throughput can increase four-fold to 2.048 Gbps. This gives the hardware a 40x advantage over software.

6 Conclusion

This paper has described an implementation of a streaming content search-and-replace module for an Internet firewall. The module is capable searching packet payloads and actively replacing the content. The content editor was implemented on the FPX platform and tested using a UDP-based chat client over the Internet. The hardware solution was found to be about 64 times faster than similar software solutions when using parallel content editors.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Kelvin Smith Library @ CASE. Downloaded on July 24,2021 at 16:43:52 UTC from IEEE Xplore. Restrictions apply.

7 Acknowledgements

This work was supported by a grant from Global Velocity. John Lockwood is a consultant to and co-founder of that company. The authors would like to thank Todd Sproull and David Lim for their development of control software and contribution to the FPX hardware platform.

References

[1] James Moscola, John Lockwood, Ronald Loui, Michael Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Napa, CA, USA), Apr. 2003.

[2] J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field programmable port extender (FPX) for distributed routing and queuing," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, (Monterey, CA, USA), pp. 137–144, Feb. 2000.

[3] 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.

[4] 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.

[5] 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.

[6] M. Roesch, "Snort - lightweight intrusion detection for networks," in *13th Administration Conference, LISA'99*, (Seattle, WA), Nov. 1999.

[7] 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.

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

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

[10] F. Kuhns, J. DeHart, A. Kantawala, R. Keller, J. Lockwood, P. Pappu, D. Richards, D. Taylor, J. Parwatikar, E. Spitznagel, J. Turner, and K. Wong, "Design of a High Performance Dynamically Extensible Router," in *DARPA Active Networks Conference and Exposition (DANCE)*, (San Francisco), May 2002.

[11] 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.

[12] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. N. Reading, MA: Addison-Wesley, 1980.

[13] G. Davies and S. Bowsher, "Algorithms for pattern matching," *Software Practice and Experience*, vol. 16, no. 6, pp. 575–601, 1986.

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

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Kelvin Smith Library @ CASE. Downloaded on July 24,2021 at 16:43:52 UTC from IEEE Xplore. Restrictions apply.