Assisting Network Intrusion Detection with Reconfigurable Hardware

B. L. Hutchings and R. Franklin and D. Carver
Department of Electrical and Computer Engineering
Brigham Young University, Provo, UT 84602
hutch@ee.byu.edu

1 Abstract

String matching is used by Network Intrusion Detection Systems (NIDS) to inspect incoming packet payloads for hostile data. String-matching speed is often the main factor limiting NIDS performance. String-matching performance can be dramatically improved by using Field-Programmable Gate Arrays (FPGAs); accordingly, a "regular-expression to FPGA circuit" module generator has been developed. The module generator extracts strings from the Snort NIDS rule-set, generates a regular expression that matches all extracted strings, synthesizes a FPGA-based string matching circuit, and generates an EDIF netlist that can be processed by Xilinx software to create an FPGA bitstream. The feasibility of this approach is demonstrated by comparing the performance of the FPGA-based string matcher against the software-based GNU regex program. The FPGA-based string matcher exceeds the performance of the software-based system by 600x for large patterns.

2 Introduction

Network intrusion detection systems (NIDS) monitor network traffic for predefined suspicious activity or data patterns and notify system administrators when malicious traffic is detected so that appropriate action may be taken. NIDSs often rely on exact string matching of packet payloads to detect hostile packets and string matching is the most computationally expensive step of the detection process [2]. Accordingly, NIDS typically apply string matching only to those packets that are most suspect, and only to those sections of the packet most likely to contain the offending data. For example, Snort (a popular NIDS found at www.snort.org) [10] checks port numbers, packet headers and flags, etc., to ensure a given packet has a high likelihood of containing hostile data before performing string matching on the packet data. Unfortunately, while this strategy of data reduction makes the problem of detecting hostile packets tractable, it also means that it is likely that a malicious packet may be overlooked. This paper explores the feasibility of using reconfigurable FPGAs to perform string matching for NIDS with the end goal of performing string matching on *all* packets at network rates.

This paper discusses the design and performance of an FPGA-based regular-expression module generator that was developed entirely in Java using JHDL [1, 7]. The module generator automatically: (1) extracts strings from the Snort rule database[10], (2) generates a regular expression that matches all extracted strings, (3) synthesizes a circuit that will match the generated regular expression, and (4), generates an EDIF netlist that can be processed by Xilinx place and route software to create an FPGA bitstream.

3 Background

Three topic areas are relevant to this project:

- past work in FPGA-based string matching,
- Snort[10], an open-source NIDS that provided the test data for performance comparisons, and
- JHDL, the Java-based hardware design tool kit that was used to implement the module generator.

String matching with FPGAs String matching is not a new application area for FPGA-based systems. Indeed, some of the earliest papers in FPGA-related research areas report efforts to accelerate string matching in a variety of areas. Several references [8, 9, 4, 5, 3] describe a few of the many string-matching efforts that have been reported over the last ten years. These efforts cover the broad range of text searching, from searching general text-based databases to similarity matching of DNA databases.

Most relevant to this effort is recent work by Sidhu and Prasanna [11] to accelerate grep regular expression searches with FPGAs. Because of the need for a rapid interactive response, their approach focused on compilation strategies that could quickly convert a regular expression into an FPGA circuit. As is commonly done in software, Sidhu and Prasanna compile the regular expression into a Nondeterministic Finite Automata (NFA); however, unlike software approaches, they skip the usual step of deriving a Deterministic Finite Automata (DFA) from the NFA, and directly implement the NFA with FPGA hardware. This simplifies and speeds up the compilation process of creating regular-expression matching hardware. Each NFA uses a single FF to implement the accepting state of the preceding stage. Flip-flop-rich FPGAs provide logic and flip-flop resources well suited for this arrangement.

The module generator discussed in this paper uses the NFA-based hardware implementation strategy of Sidhu and Prasanna because of its inherent modularity. With the NFA approach, each character and metacharacter of the supported reg-ex syntax can have a corresponding, pre-compiled circuit element in a related circuit library. Using a syntax-directed approach, the module generator instances a circuit element that corresponds to each character/metacharacter it finds in the regular expression and then interconnects these elements according to the structure of the expression. The module generator also extends the previous work by automating it and augmenting it with additional metacharacters, including: "?", ".", and "[]". Overall contributions of this work include: (1) development of a fully automated module generator that can generate circuits that match arbitrarily large regular expressions, (2) exploration of various circuit optimizations that improve speed and area utilization, and finally, (3) application of this module generator to prove feasibility of using FPGAs to accelerate string matching in network security applications.

Snort, An Open Source NIDS Snort[10] is a popular NIDS that runs under most versions of Linux and Windows. Snort's basic operation is to examine all network traffic, and log intrusion events. Pattern-matching techniques are used to compare network traffic to known attacks that are specified in a rule-set. Snort is very popular because it is open-source and because of the control it affords the user over rule-set configuration. A user can easily modify the rule-set, for example, to reduce the number of patterns to improve performance or to add patterns to detect new attacks. For this project, Snort provided a model of NIDS function and, more importantly, a default rule-set that contained the test data that were used to test the automatic module generator and the circuits that it generates. The default rule-set contains patterns for detecting various attacks as well as viral exploits such as Code Red and NIMDA.

JHDL, a Java-based Design Tool JHDL [1, 7] consists of a set of Java libraries that can be used to perform programmatic structural design [7]. In its current state, JHDL is a complete structural design environment that includes debugging, netlisting and other design aids. Circuits are described by writing Java code that programmatically builds the circuit via JHDL libraries. Each circuit element in JHDL is represented as an object; these objects inherit from core classes that setup the net-list and simulation models. Circuits are created by calling the constructor for the corresponding JHDL object and passing Wire objects as constructor arguments that are connected to the ports of the circuit. Once constructed, these circuits can be debugged and verified with the JHDL simulator and design browser. JHDL emits EDIF net-lists that can be passed to Xilinx place and route software for bit-stream generation. Finally, JHDL provides run-time support for debugging the running hardware in the context of the original design using the same GUI as the JHDL simulator. JHDL is suitable for this project because it can be used to write module generators [6] that are much more complex than can be accomplished with VHDL. For example, complex circuit-generation algorithms and data structures that are much more amenable to general-purpose languages can be written in Java and combined with JHDL circuit libraries to create sophisticated module generators.

4 Technical Approach

The technical approach was to create a JHDL-based module generator capable of handling a wide range of regular expression operators, based on standard reg-ex syntax. Supporting several reg-ex operators makes the module generator easier to adapt to future research and also makes it usable for other string-matching tasks apart from network intrusion detection. However, for this project, only two regular expression operators were absolutely necessary: concatenation (implicit) and alternation (|). The module generator uses concatenation to create strings from single characters and alternation to create one large regular expression from all of the individual strings extracted from the Snort rule-set. This section will provide a brief overview of regular expression syntax and the general form of the strings that were used to test the system, will discuss how the module generator was developed in JHDL and will discuss some of the circuit optimizations that were explored.

4. Regular Expression Syntax

Regular expressions are a common way to express string matching patterns. The atomic elements of a regular expression are the single characters to be matched.

These are combined with meta-character operators that allow the user to express concatenation, alternation, Kleene-star, etc. Concatenation (implicit) is used to create multi-character matching patterns from single characters (or substrings) while alternation (|) is used to create patterns that can match any of two or more substrings. Kleene-star (*) allows a pattern to match 0 or more occurrences of the pattern in a string. Combining the different operators and single characters allows complex expressions to be constructed. For example, the expression (th(is|at)*) will match "th", "this", "that", "thisis", "thisat"', "thatis", "thatat", etc. The supported syntax also accommodates hexadecimal notation that is used to describe nonprintable characters (ASCII) in the Snort rule database. For example, the sequence `\x00` will match the null character and `\x20` will match a space.

The module generator also supports the "?" operator which will match 0 or 1 occurrences of the expression to which it is applied. r1r2? will match r1, or r1r2. It is implemented such that it matches r1:ε where ε means it matches immediately. The availability of this operator makes it possible to study the impact that sharing common string prefixes has on FPGA circuit area. For example, the expression run|running can be converted into run(ning)? which may reduce the number of comparators used to match the characters in the expression.

A simple circuit for matching the regular expression $a(b|c)?$ is presented in Figure 1 to help explain the general operation of the matching circuits created by the module generator. $a(b|c)?$ matches: "a", "ab", and "ac". The corresponding matching circuit (shown in Figure 1) consists of three character matchers (the three blocks in the figure), two OR-gates and interconnecting wires. A character matcher consists of an edge-triggered D flip-flop (FF), a character comparator, and an AND-gate. The input to the FF serves as an enable to the character matcher; the output of the character matcher becomes true on the *next* clock edge if: (1) the character input matches the compared value, and (2) the enable signal (the FF input) is a logical '1'. During circuit operation, all input characters are broadcast to all character matchers one character at a time. For the sake of illustration, assume that the input string to the circuit is "ab" and that all FFs are reset to a logical '0'. Now, 'a', the first character, is broadcast to all character matchers and on the next clock edge the output of the first character matcher becomes a logical '1'. This enables the 'b' and 'c' character matchers because this output is wired to the enables for these matchers; it also causes the Match signal to become true (the regular expression matches "a"). The second character, 'b', is then broadcast to the character matchers and on the next clock edge the output of the 'b' character matcher becomes true. This signal is OR-

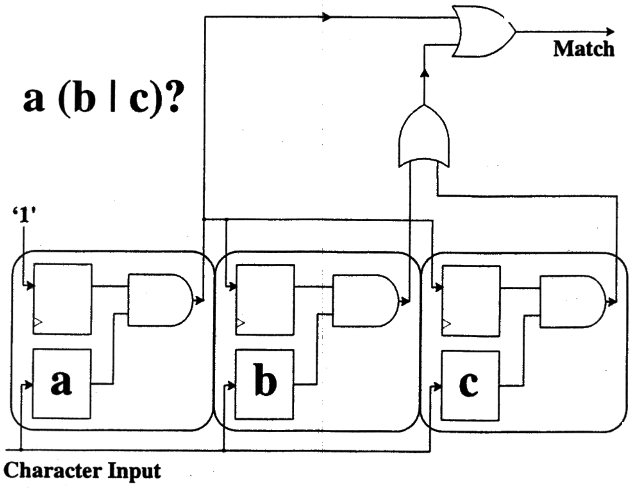

Figure 1: A Matching Circuit for $a(b|c)?$

ed together with the output from the 'a' character matcher output and generates a true value on the Match output (the regular expression matches "ab"). For additional examples, please see the paper by Sidhu and Prasanna [11].

4.2 Examples from the Snort Rule Database

Patterns from the Snort rule database are used to generate regular expressions for intrusion detection. Snort rules, in general, specify much more than just a data pattern to search for; they include information on packet headers and flags, TCP ports, etc. However, this effort is only concerned with string matching, so the parser extracts only the "content" fields from Snort rules and combines them into one regular expression. For example,

```
content:"|00|E|00|M|00|L";
content:"|00|N|00|W|00|S";
content:"R|00|I|00|C|00|H|00|E|00|D|00|2|00|0";
```

are the actual content fields from a Snort rule that detects the NIMDA virus. These three patterns are combined by the module generator into a single regular expression, shown below.

```
(\x00E\x00M\x00L  |
\x00N\x00W\x00S   |
R\x00I\x00C\x00H\x00E\x00D\x002\x000)
```

Another example is the content field from a Snort rule for detecting attempts to access the Code Red 2 back Door:

```
content:"scripts/root.exe?"
```

which is converted to the regular expression: `(scripts/root\.exe\?)` (the backslash "escapes" the "." and "?").

4.3 Operating Environment

The operating environment for this module generator differs from that described in Sidhu and Prasanna in two significant ways.

1. *Hardware can be precompiled.* Patterns are known well beforehand and compile time is not an issue for network security applications. Thus, vendor place and route tools are used to create a fully automatic regular-expression-to-bitstream tool. This is in contrast to Sidhu and Prasanna where vendor place and route software could not be used and there was no automatic path from the regular expression to matching hardware (several manual steps were reported in the paper).

2. *Matching patterns can be very large.* The entire default Snort rule-set contains content rules in excess of 15,000 characters and will continue to grow as more attacks and viruses are identified. This means that circuits must be area efficient so that the available FPGA resources can be used to match as many patterns as possible.

4.4 JHDL versus VHDL

The design and implementation of the regular-expression module generator was heavily influenced by the availability of a general-purpose language such as Java. Generally, complex module generators consist of two major parts: a front-end that performs *data processing* to determine the structural circuit details based on some user specification, and a back-end that performs *circuit generation* and emits the corresponding net-list. Except in cases where the module generator is extremely simple, a general-purpose language is better suited for implementing the front-end than a Hardware Description Language (HDL) such as VHDL. Front-end processing usually interprets user input and this often requires the sort of processing normally associated with programs written using general-purpose languages: parsing of input, error detection, construction and processing of dynamic data structures such as linked lists and trees, file I/O, etc.

For example, the front-end of the regular-expression module generator performs the following:

1. opens and parses the Snort file to extract the content fields of each rule (the strings to be matched),

2. generates a large regular expression that includes all of these strings,

3. parses the generated regular expression to create an internal representation of the expression that is used for circuit generation, and

4. optimizes the generated regular expression to extract substrings that can be shared, etc.

The module generator front-end is implemented with Java source code and the back-end uses Java source and JHDL libraries to generate a detailed structural description and EDIF net-list.

VHDL is not suitable as the sole implementation language for a module generator such as that described in this paper. Apart from the general awkwardness of writing parsers, dynamic data structures, etc., in VHDL and debugging them with a simulator, VHDL tools make it impossible to perform execution of both the front-end (data processing) and back-end (circuit generation) in a single tool or environment. For example, front-end software written in VHDL can be "executed" with a VHDL simulator, however, it is not possible to then generate circuits in a simulator. On the other hand, a VHDL synthesis tool can perform circuit generation, but it cannot execute the front-end software in order to obtain the parameters used to generate the circuit.

The only way to overcome this problem with VHDL is to take a combined strategy that uses a general-purpose language such as C++ for front-end processing and a VHDL synthesis tool as the back-end for circuit generation. However, this approach is also unsatisfactory for several reasons:

1. It requires the designer of the module generator to have expertise in both C++ and VHDL.

2. It complicates the design and debugging of the module generator because the designer will have to debug both the C++ and VHDL code.

3. It requires the end-user to install and maintain an expensive synthesis tool that in many cases only serves as a net-list translator because the front-end usually generates a structural description from the user input.

4. It requires the end-user to understand how to use the VHDL synthesizer (at least be able to interpret messages generated by the VHDL synthesizer) because the front-end will occasionally generate incompatible VHDL code due to bugs in either the front-end or the synthesizer, or due to versioning problems.

JHDL overcomes the basic problems listed above because it is based on a general-purpose language. Moreover, this general-purpose language heritage also makes it straightforward to add new functionality to the JHDL suite. The schematic viewer, simulator, and various browser windows have been developed in modular fashion and can be extended or modified to ease debugging and verification on a per application basis. For example, the default

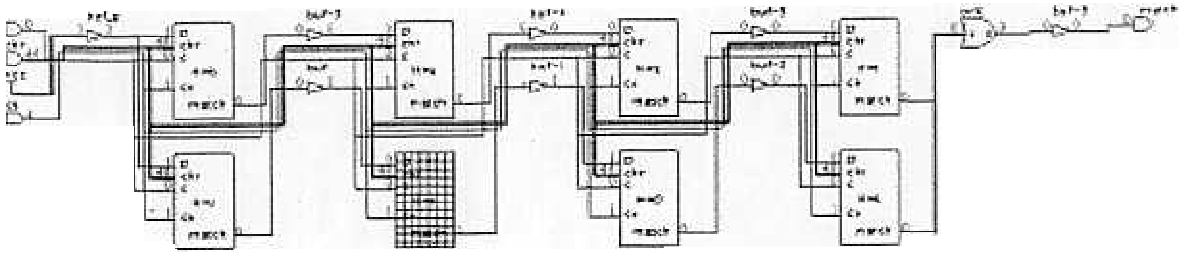

Figure 2: Modified Schematic View

schematic view was extended during this project to provide a direct visual indication of subexpression matches that occur during a debugging session. In Figure 2, the expression `(JHDL|bust)` is being matched against an incoming string of characters. In this example, the cross-hatch pattern indicates that the 'J' and 'H' characters have matched thus far. Extending JHDL tools in this manner is enabled via a standard API and a few lines of Java code.

4.5 Hardware Implementation Details

Efficient utilization of FPGA resources is important because Snort rule-sets currently contain approximately 15K characters – a figure that will grow continually. Accordingly, various optimization strategies are used to improve the utilization and performance of circuits generated by the module generator. To improve overall performance, the module generator: (1) uses mapping directives to reduce the area of the most commonly used operator (concatenation), (2) creates a pipelined fan-out tree to reduce propagation delay for character broadcasting, (3) reorganizes the regular expression to improve OR-gate performance, and (4) shares common subexpressions to further reduce area.

The module generator optimizes the concatenation operator (the most commonly used operator) by forcing a single, eight-bit character matcher to fit within a single slice by using mapping directives. The output of the matcher passes through a flip-flop and connects back into the slice via the carry chain to implement a logical AND of the match of the current character and the previous one as shown in Figure 3. This leads to a reduction in circuit area that is reported later. Minor attempts were also made at manual placement but early feedback suggested this may not be effective for this application.

With the NFA approach, incoming characters are globally broadcast to all character matchers and this can create performance problems due to very large fan-out and the resultant loading of the broadcasting wires. The module

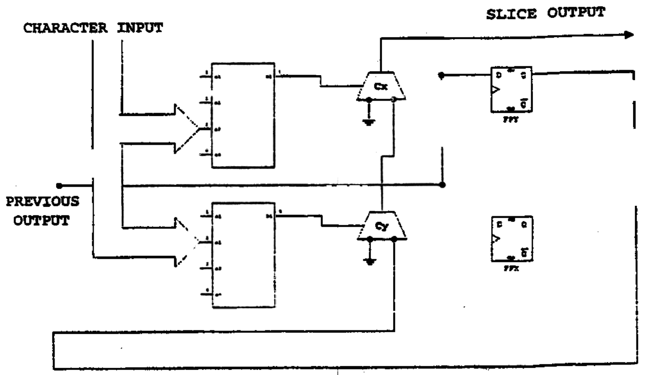

Figure 3: Virtex Slice usage

generator overcomes this by creating a pipelined broadcast tree that takes into account the fan-out requirements of the string matcher. The tree structure reduces loading on any individual wire by distributing the load across several branches of the tree. The tree is pipelined to further improve clock rate.

The module generator also reorganizes the regular expression to make it easier to implement the alternation operators as distributed OR-gates. For example, the alternation operator can be naively implemented with a single OR-gate for each occurrence of the operator. However, this will create a long serial chain of OR-gates that may result in long delays and poor performance. The module generator takes an alternative approach and transforms the regular expression from infix to postfix form. For example, the expression `(a|b|c|d)` is transformed into `(abcd |||)`. This makes it easier to detect a run of alternation operators. Each run of alternation operators is grouped together with an OR-gate of appropriate size (four-input in the example) and the module generator connects these with other OR-gates for which these runs are constituents. This parallel approach to implementing the OR-gate improves performance by reducing the combinational delay that occurs with the naive, serial approach.

Different versions of the module generator also combined common prefixes to reduce FPGA area. For example, the words "this" and "that" share the common prefix "th". This means that the two regular expressions, "this|that" and "th(is|at)", are equivalent. The module generator recognized these common prefixes by building a TRIE dictionary from the different sequences within the regular expression. Experiments confirmed the area savings, however, clock frequency was generally lower because sharing common prefixes lowers the number of flip-flops between combinational logic. Table 1 illustrates the tradeoffs for a 600 character string containing shared prefixes.

5 Performance Analysis

To determine relative performance, the circuits generated by the module generator were compared against a software-based string matcher, GNU "regex". The testing process used three computers: one that sends data consisting of mostly printable text with inserted attacks, a second that reads this data over the network and time stamps it immediately before and after passing it through either the GNU regex or FPGA-based matcher, and a third computer that reads the time-stamped data and computes overall latency. The test data varied in size and incorporated a representative sample of rules in the default Snort rule-set that target the following types of intrusion:

- attacks on webservers
- viruses such as NIMDA and Code Red
- backdoor/trojan exploits

5.1 Test Setup

Pentium 3 (750 MHz) computers running Redhat Linux version 2.3.7-10 served as the test computers. The string-matching circuits executed on a PCI-based ISI SLAAC-1V board (housed in the second computer) that contains a Virtex XCV1000 device and several FIFOs that support high-throughput DMA transfer. Global clock rate for the FPGA device was set to 33MHz because of limitations in the FPGA interface to the PCI bus – a figure that is generally much lower than the FPGA circuits can operate, as reported by the Xilinx static timing analyzer. The reported latencies include only the actual time spent string matching and ignore any delays due to TCP/IP overhead or other network delays (time stamps are computed just before, and just after the string match occurs). Overall throughput was computed by dividing the size of the data set by the average time taken to process the entire data set over ten test runs. The CPU utilization figures are calculated according to:

$$\frac{(User\ Time + System\ Time)}{Elapsed\ Real\ Time}$$

5.2 Interpretation of Results

From the results in Tables 2 and 3 it can be seen that for small test cases, software and hardware performance are approximately the same. However as the size of the regular expression increases, the hardware based implementation outperforms the software-based version by more than 600x in the worst case. Throughput for the circuit-based string matchers requires one clock cycle per character of input ($O(1)$ in regular-expression length); larger regular expressions simple acquire more circuit area to exploit more parallelism to process the longer string in the same amount of time. Throughput for the circuit-based string matchers is thus independent of the length of the regular expression and the small variances seen in the tables are the result of experimental error. In contrast, software-implemented string matchers require more time to process each incoming character as the regular expression grows in length. An examination of the two tables shows that, for the middle range of the tests (844 - 2689 characters), software slowdown relative to the size of the regular expression is roughly linear ($O(n)$ in regular-expression length) although it gets worse at 4971 characters. Another item of interest is the relatively low and relatively constant CPU utilization of the hardware based implementation which is independent of the size of the data being searched. This occurs in the hardware-based test setup because the CPU is primarily moving data, leaving the computationally-intense string matching operations to the FPGA-based matcher. CPU utilization and throughput data from Table 2 are also plotted in Figures 4 and 5.

5.3 FPGA Utilization Data

Tests also included the computation of FPGA area utilization for several string matchers of different sizes. Data are organized into two sets: one set shows the performance/utilization data with manual mapping turned on (as discussed in the Technical Approach), the other set shows performance/utilization data with manual mapping turned off (Table 5). As is typical for many FPGA-based designs, clock frequency for the hardware string matchers drops with increased utilization. This tends to be especially true for circuits like the string matcher which require lots of

Table 1: Comparison of approaches for Regular Expression Implementations

Implementation	Slices	Flip-Flops	LUTs	Routed	Frequency (MHz)
NFA(non-shared, 660 chars)	674	661	1347	7946	43.219
NFA(shared, 660 chars)	556	527	1092	6405	34.483

Table 2: FPGA vs. software regex performance for a 1MB data set sent in 1kB chunks

Size of Regular Expression (# of non-Meta characters)	Hardware Latency (ms)	Software Latency (ms)	Hardware Throughput kB/s	Software Throughput kB/s	Hardware CPU Utilization	Software CPU Utilization
47	< 1	< 1	390	432	33.9%	11.2%
435	< 1	3.2	340	197	33.6%	67.6%
844	< 1	37.6	381	23.5	34.3%	91.9%
1,420	< 1	104	284	8.90	28.4%	96.3%
2,689	< 1	240	291	4.63	24.7%	98.3%
4,971	1.2	970	331	0.99	43.7%	99.4%

Table 3: FPGA vs. software regex performance for a 10MB data set sent in 16kB chunks

Size of Regular Expression (# of non-Meta characters)	Hardware Latency (ms)	Software Latency (ms)	Hardware Throughput kB/s	Software Throughput kB/s	Hardware CPU Utilization	Software CPU Utilization
47	< 1	< 1	862	884	19.3%	11.8%
435	< 1	50.9	870	278	18.1%	97.3%
844	< 1	602	824	24.0	16.3%	98.3%
1,420	< 1	978	826	14.9	19.3%	99.6%
2,689	< 1	1930	838	7.58	20.1%	99.6%
4,971	7.38	8400	784	1.72	38.5%	99.8%

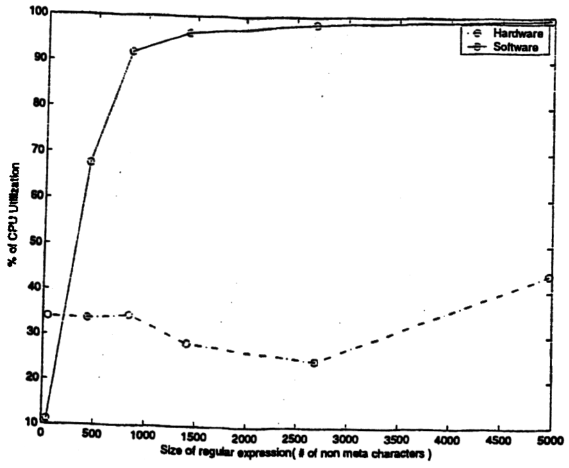

Figure 4: CPU Utilization

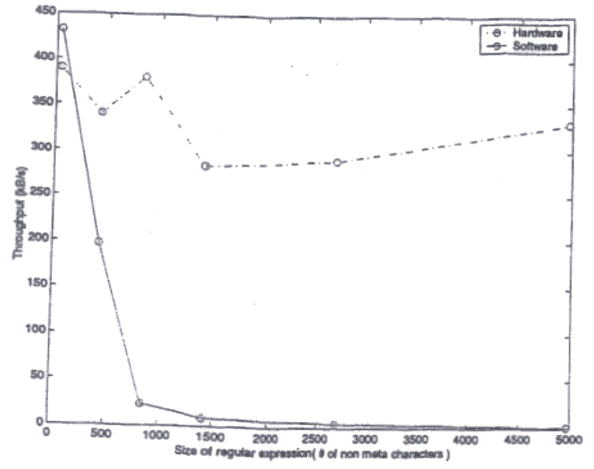

Figure 5: CPU Throughput

global routing resources (string matching requires broadcasting of character data throughout the FPGA).

The results obtained by turning off manual mapping are somewhat surprising. Turning off manual mapping dramatically increases the amount of circuitry by at least 50% or more but reduces clock rate for the largest regular expressions. Further study of this issue will be required to fully understand the tradeoffs that manual mapping provides, however, for the time being, these data indicate that the module can either optimize for clock rate or area depending on whether the number of patterns or overall performance is the major concern.

6 Conclusions

FPGA-based string matching appears to be a suitable candidate for use in NIDS. Inclusion of FPGA-based matchers in these systems may reduce the CPU workload significantly and make it possible to examine more data in more packets than is possible with software-only approaches. Presently, the rules in NIDS like Snort must be carefully written to examine packet data payloads only after everything else has been tried; hardware-based matchers like those described in this paper should ease this restriction considerably. This is not to imply that accelerating string matching as discussed in this paper is a panacea; network security is a very complex problem and string matching represents only a small part of the overall solution. Still, hardware-based pattern matchers that are capable of inspecting packet data at network data rates for most or all of the traffic on the network should prove to be very useful.

7 Future Work

The work reported here proves basic, technical feasibility of using reconfigurable FPGAs to accelerate string matching for network security applications. Future work will probably focus in two general areas: circuit and technology optimizations, and additional applications in network security. In these early stages of the project, the focus was on achieving error-free operation with moderate levels of performance. The generated circuits are only moderately pipelined and there is a lot of room for clock rate improvement. Virtex-2 devices also present opportunities for performance enhancement and optimization strategies for these devices will be studied. Beyond string matching, there are opportunities for applying circuit-based optimizations to other parts of the networking and security problem. For example, packet reassembly was performed using software for this project. It *may* make sense to implement the TCP/IP stack directly on the FPGA in order

Table 4: FPGA utilization statistics for various sized designs

Size of Regular Expression (# of non-Meta characters)	# of Slices Required	# of Slices per Character	XCV1000 Part: Maximum Operating Frequency (MHz)	XCV1000 Part: % of available slices used	XCV2000e Part: Maximum Operating Frequency (MHz)	XCV2000e Part: % of available slices used
99	157	1.59	99.3	1%	126.3	1%
506	863	1.71	63.5	7%	85.8	4%
1,005	1,423	1.42	62.2	11%	83.0	7%
2,008	2,331	1.16	49.9	18%	82.2	12%
4,003	4,375	1.09	43.5	35%	88.6	22%
8,003	10,309	1.28	30.9	83%	52.5	53%
16,028	20,116	1.26	N/A	N/A	49.5	79%

Table 5: Comparison of optimized mapping scheme vs. standard XILINX mapping

Size of Regular Expression (# of non-Meta characters)	Optimized Mapping Scheme: # of Slices Required	Optimized Mapping Scheme: # of Slices per Character	Optimized Mapping Scheme: Maximum Operating Frequency (MHz)	Completely Unmapped: # of Slices Required	Completely Unmapped: # of Slices per Character	Completely Unmapped: Maximum Operating Frequency (MHz)
99	157	1.59	99.3	256	2.56	90.6
506	863	1.71	63.5	1,349	2.70	62.4
1,005	1,423	1.42	62.2	2,271	2.27	51.9
2,008	2,331	1.16	49.9	3,845	1.92	52.7
4,003	4,375	1.09	43.5	7,486	1.96	61.1
8,003	10,309	1.28	30.9	13,450	N/A	N/A

to achieve higher levels of performance. Finally, FPGAs that include an embedded processor present an interesting target of opportunity; they have the potential to provide an integrated, scalable approach that allows a mix of software and hardware to be dedicated to problems in network security. Software may be used for less regular, control-specific processing while the hardware may be used for performing computationally intensive tasks for large, regular data sets.

8 Acknowledgements

The authors would like to thank the many students in the Reconfigurable Logic lab for their help in this project. Many thanks also go out to the folks at Los Alamos National Laboratories and ISI-East (Maya Gokhale, Brian Schott and others) for suggesting this topic to us at a SLAAC retreat.

9 References

[1] P. Bellows and B. L. Hutchings. JHDL - an HDL for reconfigurable systems. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 175–184, Napa, CA, April 1998.

[2] C. J. Coit, S. Staniford, and J. McAlerney. Toward faster string matching for intrusion detection or exceeding the speed of snort. In *DARPA Information Survivability Conference and Exposition II, 2001 Proceedings*, volume 1, pages 367–373, 2001.

[3] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. The Teramac configurable custom computer. In J. Schewel, editor, *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing.*, pages 201–209, Philadephia, PA, October 1995.

[4] P. W. Foulk. Data-folding in SRAM configurable FPGAs. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 163–171, Napa, CA, April 1993.

[5] B. Gunther, G. Milne, and L. Narasimhan. Assessing document relevance with run-time reconfigurable

machines. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 10–17, Napa, CA, April 1996.

[6] S. Hemmert and B. L. Hutchings. An application-specific compiler for high-speed image morphology. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages to–appear, Napa, CA, April 2001.

[7] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A cad suite for high-performance fpga design. In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, page n/a, Napa, CA, April 1999. IEEE Computer Society, IEEE.

[8] D. P. Lopresti. Rapid implementation of a genetic sequence comparator using field-programmable gate arrays. In C. Sequin, editor, *Advanced Research in VLSI: Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 138–152, Santa Cruz, CA, March 1991.

[9] D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–177, Napa, CA, April 1993.

[10] Martin Roesch. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conference, LISA '99*, Seattle, WA, November 1999. www.usenix.org/events/lisa99/full_papers/roesch/roesch_html/.

[11] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages to–appear, Napa, CA, April 2001.