Compiling PCRE to FPGA for Accelerating SNORT IDS

Abhishek Mitra, Walid Najjar and Laxmi Bhuyan
Department of Computer Science and Engineering
University of California, Riverside
{amitra, najjar, bhuyan} @cs.ucr.edu

ABSTRACT

Deep Payload Inspection systems like SNORT and BRO utilize regular expression for their rules due to their high expressibility and compactness. The SNORT IDS system uses the PCRE Engine for regular expression matching on the payload. The software based PCRE Engine utilizes an NFA engine based on certain opcodes which are determined by the regular expression operators in a rule. Each rule in the SNORT ruleset is translated by PCRE compiler into an unique regular expression engine. Since the software based PCRE engine can match the payload with a single regular expression at a time, and needs to do so for multiple rules in the ruleset, the throughput of the SNORT IDS system dwindles as each packet is processed through a multitude of regular expressions.

In this paper we detail our implementation of hardware based regular expression engines for the SNORT IDS by transforming the PCRE opcodes generated by the PCRE compiler from SNORT regular expression rules. Our compiler generates VHDL code corresponding to the opcodes generated for the SNORT regular expression rules. We have tuned our hardware implementation to utilize an NFA based regular expression engine, using greedy quantifiers, in much the same way as the software based PCRE engine. Our system implements a regular expression only once for each new rule in the SNORT ruleset, thus resulting in a fast system that scales well with new updates. We implement two hundred PCRE engines based on a plethora of SNORT IDS rules, and use a Virtex-4 LX200 FPGA, on the SGI RASC RC 100 Blade connected to the SGI ALTIX 4700 supercomputing system as a testbed. We obtain an interface throughput of (**12.9 GBits/s**) and also a maximum speedup of **353X** over software based PCRE execution.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.2.0 [**Computer Communication Networks**]: General—*Security and Protection*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'07, December 3–4, 2007, Orlando, Florida, USA.
Copyright 2007 ACM 978-1-59593-945-6/07/0012 ...$5.00.

General Terms

Design, Performance, Security

Keywords

Regular expressions, nondeterministic finite automata, intrusion detection system, deep payload inspection

1. INTRODUCTION

Increase in malicious activities using computer networks as a medium, has also resulted in an increased deployment of intrusion detection systems(IDS) to scan and intercept network packets containing signatures of such activities. The SNORT [4] software is a popular and widely used open source IDS for securing the network of an organization from malicious activities. The SNORT IDS interacts with the TCP/IP stack on a computer or a security appliance, to intercept and scan the network payload, in order to identify signatures of malicious activities such as Buffer Overflow attacks, Denial of Service Attacks, Man in the Middle attacks, etc. on the network packets and thus avoid potential contingencies. The SNORT IDS utilizes regular expression based matching in addition to string based matching for identification of malicious signatures in the network payload. The signatures of vulnerabilities and malicious activities are represented as a set of rules, which are frequently updated by the security community. The growing number of the vulnerabilities, has led to an ever growing number of rules in the SNORT ruleset. As a result, the SNORT IDS ends up expending an ever increasing number of CPU cycles for each payload, scanning through the list of rules. With the rapid enhancement of the bandwidth of network connections, viz. of the order of Tens of Gbps; it has thus been necessary to offload network processing applications to dedicated hardware [19] and free up the host processor. In fact, even the current state of the art processors such as Intel XEON Woodcrest (3.0 GHz) and Intel Itanium Montecito (1.6GHz) are unable to maintain the necessary [11] throughput while running the SNORT IDS with multiple Regular Expression Engines, thus necessitating the execution of the IDS engines in dedicated hardware. The Field Programmable Gate Array (FPGA) arena has seen rapid development in speed and silicon logic size in recent years, with the latest devices supporting multi Gigabit throughput interfaces to the host processor. Moreover the inherent flexibility of an FPGA based design, allows us to leverage them for implementing highly optimized parallel logic circuits, supporting a multitude of regular expression engines.

Table 1: Example Rules in SNORT DB 2.6

SNORT Ruleset	Regular Expression Rule	Attack Type	Implication
backdoor	pcre:"/^Netbus\s+\d+\x2E\d+/smi"	Netbus Trojan	Captures the header of the Netbus trojan i.e. Netbus followed by one or more spaces, one or more digits, character '.' and one or more digits.
web-misc	pcre:"/^[^\x3e\x3f\x26]{63}/R"	Buffer Overflow	Captures a McAfee specific buffer overflow attack sequence i.e. Any 63 characters other than >, ? or & .

Table 2: Format of a typical PCRE Rule in SNORT IDS with the optional modifiers

pcre:"/<regex>/[ismxAEGRUB]";

In this paper we present a novel method to compile PCRE Operation Codes (opcodes) directly to Very High Speed Integrated Circuits Hardware Description Language (VHDL), for parallel implementation on FPGA hardware. We implement the PCRE regular expressions from the SNORT IDS using a two stage translation process. In the first stage, the SNORT IDS rulesets are compiled using the PCRE compiler to generate PCRE opcodes. In the second stage the PCRE opcodes are translated to VHDL hardware blocks suitable for implementation on FPGA and connected together using a NFA based control logic. Our system maintains the execution semantics of the software based regular expression engine on the FPGA hardware, thus ensuring compatibility with the SNORT IDS ruleset. The interface throughput suffices for wire-speed payload scanning of even the fastest available ethernet interfaces. Our design is a compile once, NFA based design, with re-compilation necessary only for new and updated rules. We obtain more than 350X speedup with our FPGA based regular expression engine architecture when compared to a baseline state of the art CPU viz. the Intel Xeon 5160 and our design can sustain a throughput of 12.9 Gbps.

The rest of the paper is organized as follows: In Section 2, we lay out the background work related to the SNORT IDS, Regular Expressions, PCRE, and using FPGA based hardware for accelerating software code. Section 3 details on our compilation process, on how the PCRE opcodes are handled by our scripts in a two stage process to generate VHDL blocks and NFA control logic. We also discuss our parallel PCRE engine architecture implemented on the FPGA. In section 4 we detail our experimentation method and finally the results and speedup, when executing parallel PCRE engines on FPGA based hardware. Related work vis-a-vis PCRE and SNORT IDS is discussed in section 5. Finally we sum up with the conclusions and future directions in section 6.

2. BACKGROUND

The following section describes the benefits of using Perl Compatible Regular Expressions (PCRE) [6] vis-á-vis SNORT IDS. We describe the mechanism by which SNORT IDS utilizes the PCRE compiler for translating the regular expression based rules from the SNORT database and matching them on the payload using the PCRE engine. We also provide insight into the implementation of PCRE engines derived from SNORT ruleset on actual hardware viz. Virtex-4 LX 200 FPGA on SGI RASC RC 100 blade[20].

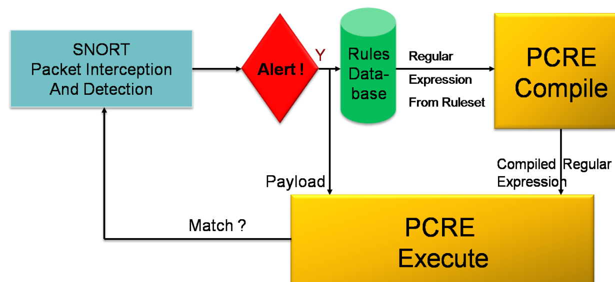

Figure 1: SNORT IDS and PCRE Engine usage on CPU

2.1 SNORT IDS

Intrusion Detection Systems(IDS) such as SNORT and BRO [5] started as string matching engines for deep payload inspection of network packets using a database of signature strings known as the rulesets. As the database of string based signatures expanded, the efficiency of the rules started to dwindle. Therefore regular expressions are increasingly used due to their advantages of expressibility and compactness. A single regular expression can possibly encompass tens and hundreds of individual string representations, and thus they have become a highly popular method for constructing signatures for IDS.

SNORT is a GPL'd IDS based on a community driven ruleset wherein the rules are updated frequently by the security community thus capturing the signatures vis-a-vis the newest vulnerabilities and malicious packets. PERL based regular expressions are being increasingly utilized for charting out the SNORT ruleset due to their compact representation, excellent expressibility and wide usage across the community. The SNORT IDS utilizes a plugin oriented architecture to enable regular expression matching as well as various additional features. Table 1 exemplifies two different PCRE rules from the SNORT database ver. 2.6. More than four thousand such rules make up the SNORT PCRE rulesets. The PCRE engine is used as a plugin by SNORT IDS to run a regular expression match on the intercepted payloads as depicted in Figure 1. Table 2 highlights the format of a typical PCRE rule in SNORT IDS with the optional PCRE specific flags. The commonly used PCRE flags include 'i' for case insensitive match, 's' for inclusion of newlines in the dot operator, 'm' for enabling anchors to match immediately following a newline, and 'x' to ignore whitespace between regular expression token.

Table 3: Occurences of important PCRE operators in SNORT DB 2.6

Regular Expression Operator	Occurences
Anchor: Match the first character "^"	3087
Anchor: Match the last character "**$**"	3
Quantifiers "**{}**"	2833
Ranged Quantifiers "**{n,x}**"	303
Negated Character Class "[^**...**]"	1575
Repetition "*****"	1492
Repetition "+"	1386
Back References "**1**,**2**,**3**,**4**,**5**,**6** "	298

2.2 PERL and PCRE

PERL is a very popular string oriented language with a rich set of regular expressions, thus making it highly suitable for creating regular expressions based rules for SNORT IDS. PCRE software consists of two parts namely the PCRE compiler and the PCRE Engine. The PCRE compiler compiles PERL based regular expressions into a set of op-codes, which would then be suitable for execution on the PCRE engine. The engine executes the regular expression represented as opcodes with a given string to recognize whether the regular expression matches the string. The PCRE engine executes a greedy quantifier matching NFA which conforms to the PERL regular expression semantics. The important PCRE operators including the anchors, quantifiers, ranged quantifiers, character classes, and their occurences in the SNORT rulesets is highlighted in Table 3 and example rule snippets demonstrating their implications in tabulated in Table 4. Regular expressions can produce different results depending on the execution engine. As an example a greedy NFA would provide a different result when compared to a non greedy execution engine. Since PERL utilizes a NFA based, greedy quantifier match strategy as default and the SNORT rules have been generated by the community to adhere to the PERL regular expression standards, thus it is extremely important for the accelerated regular expression engine in hardware to adhere to the PERL regular expression standards in order to successfully detect malicious activity. It is possible that non-conforming implementations may result in false negatives, which could result in potential security issues. As an example a greedy quantifier regular expression engine, using the regular expression /**test.*test**/ on the string "This test is testing greedy and lazy tests", would match and return "*test is testing greedy and lazy test*" while a lazy quantifier match would return "*test is test*" i.e. up until at the fourth word which is "testing".

2.3 Regular Expression Engines on FPGA

The basic buiding block of regular expression engines implemented on FPGA are finite automata. The basic FPGA logic elements in an FPGA are LUTs, which are volatile SRAM memory elements that store given data. These LUTs are connected to other LUTs on chip via routing network, thus an output of one LUT may look up values in another LUT, and through a series of such chains, the FPGA implements complex logic on its fabric. An example of a finite automata is depicted in Figure 2. The language of the depicted finite automata matches a string with even numbers of zeros. The states of the finite automata are encoded as '0' corresponding to S1 and '1' corresponding to S2. The

Table 4: Example snippets from SNORT Rules highlighting the use of PCRE operators

<table>
<tr><th>Operator</th><th>Snippet</th><th>Implication</th></tr>
<tr><td>"^"</td><td>^NetBus</td><td>Netbus at the start of line</td></tr>
<tr><td>"$"</td><td>[\x26|\s]|<b>$</td><td>Ends with either a '&', space
or End Of Line</td></tr>
<tr><td>"{}"</td><td>[\x26]"{63}"</td><td>'&' Exactly Sixty Three Times</td></tr>
<tr><td>"{n,x}"</td><td>[^\n]"{244,255}"</td><td>Any character but newline, more than 244 but less than 255 times</td></tr>
<tr><td>"[^...]"</td><td>[^\r\n]</td><td>Any character but CR,LF</td></tr>
<tr><td>"*"</td><td>[^\r\n]*</td><td>Any character but CR,LF Zero or more number of times</td></tr>
<tr><td>"+"</td><td>\s +</td><td>White Spaces One or more
number of times</td></tr>
<tr><td>"\1,\2,\3,"<br>"\4,\5,\6 "</td><td>(\x22|\x27|)
... \1</td><td>If ' " ' was matched earlier then match ' " ', if ' ' ' was matched then match ' ' '</td></tr>
</table>

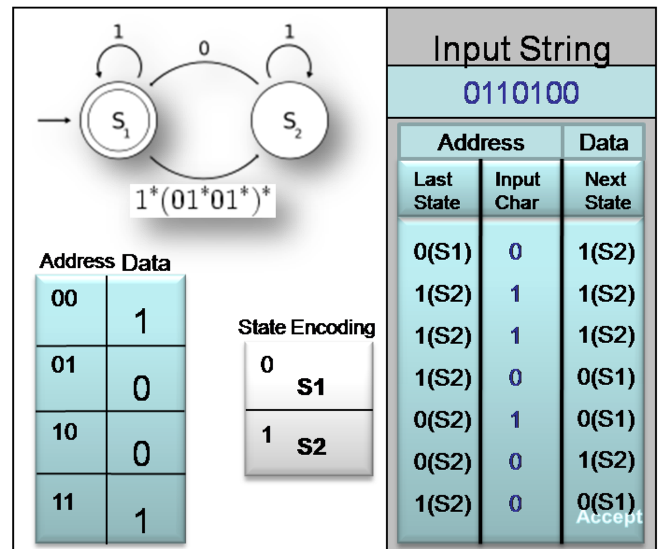

Figure 2: A Finite Automata Implemented on FPGA using LUTs.

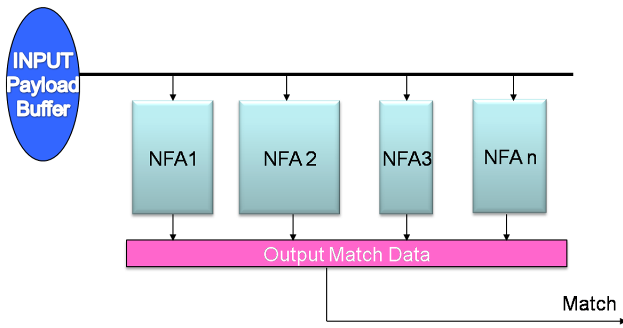

Figure 3: Multiple NFA engines executing in parallel on a FPGA.

first lookup table of size four elements is addressed by the current state of the automata and the current input data. As an example, if the automata is in state S1 and receives an input character '1', the corresponding address in the LUT is '01'. The data stored at address '01' is '0' and is routed to the next LUT on the right which contains two locations. Since the corresponding data at address '0' in the second LUT is S1 hence the finite automata selects state S1 as its next state. The gray section in Figure 2 demonstrates the state transitions as the automata processes and accepts an example string "0110100".

The software implementation of regular expression engines can only run a single instance of an engine on a CPU core, which results in limited throughput of the software based IDS. However hardware based IDS use multiple instantiations of regular expression engines on reconfigurable silicon, such as an FPGA, in order to enable a multitude of parallel engines, as shown in Figure 3, thus speeding up the system many times. Various optimizations to the engine are possible, including sharing of prefix, common subexpressions and constant matches in the hardware along with choice of engines implemented as a DFA or as a NFA.

Since the logic design on an FPGA can be updated as and when required, it makes them an ideal platform for supporting newer or updated regular expressions, and thus any modification to the SNORT IDS ruleset would result in the re-compilation of the new or modified rule only.

3. COMPILING PCRE ENGINES TO FPGA

The heart of our automated compiling system lies in the way regular expressions from the SNORT database are converted to engines on FPGA. The initial step involves invoking the PCRE compiler software which is part of the PCRE package version 7.0. We have added a mechanism for the PCRE compiler to emit compiled opcodes from regular expressions to a local database. Our AWK script parses the database and extracts the opcodes corresponding to the regular expression based rules. Thereafter the rules are processed by the opcode to VHDL script. This script first generates storage block for each opcode that matches one or more characters. A second script processes the sequence of PCRE operators, in the list of opcodes and generates a NFA based control logic in VHDL. A third script combines them to a VHDL entity that interacts with the memory interface module. Once all the regular expression engines are generated, the fourth script generates a payload buffer and a matchdata buffer. The payload buffer receives TCP / IP payload from the software and on each clock cycle retrieves a character into each of the engine through the memory interface module. The matchdata buffer is connected to the match output of each of the NFA engines, and the data is sent back to the CPU once the complete payload has streamed through the regular expression engines.

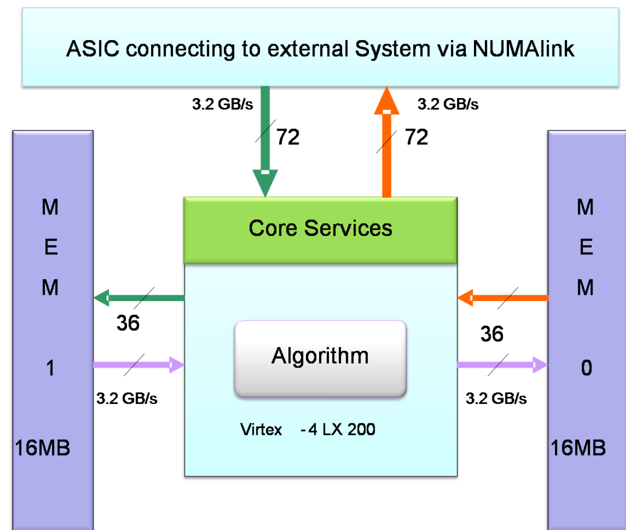

Figure 4: Organization of the RASC RC 100 Blade with the Virtex 4 LX 200 FPGA.

3.1 The SGI RASC RC100 Hardware

FPGAs allow speedup of slow sequential software by efficient hardware execution, as well as executing multiple threads in parallel, thus augmenting the host processor. We utilize the SGI RASC RC100 Blade as a proof of concept platform for demonstrating the performance of PCRE on hardware. The RASC RC 100 blade consists of two Virtex-4 LX 200 FPGAs, with 40MBytes of SRAM logically organized as two 16MBytes blocks and an 8MBytes block, as shown in Figure 4. The SRAM are 36Bit QDR devices with 4 bit parity, thus transferring 128bit data every clock cycle. The RC100 Blade is connected using the low latency NUMALink interconnect to the SGI Altix 4700 Host System, for a rated bandwidth of 6.4GB per second per FPGA. We are able to implement two hundred PCRE engines which operate in parallel on the network payload, thus achieving throughput rates of the order of 12.9 Gigabits per second. The SGI Altix 4700 allows multiple RASC Blades to be installed on the host system, thus potentially allowing for implementing of the complete SNORT Database on hardware. The Virtex 4 LX 200 FPGA is the largest FPGA chip in the Virtex-4 series, containing 200,448 logic cells. The FPGA also contains 700KBytes of Block SRAM memory on the chip, that appends the memory capacity of the distributed logic. Two rulesets in the SNORT rules database viz. the oracle and the web-client rules consist of a multitude of rules with long list of back-reference data. Such a long list of data is unsuitable for storage in the distributed logic on the FPGA, and due to its sheer size, causes overmapping of logic resources. Our compiler uses the Block RAM for storing back reference data for the aforementioned rulesets, thus mitigating the problem

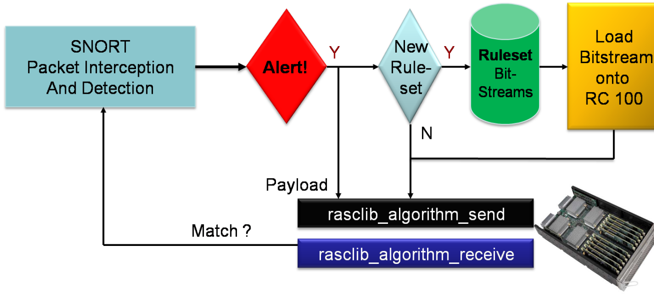

Figure 5: SNORT IDS and PCRE Engine usage on FPGA

of overmapping. Each FPGA on the RASC RC 100 Blade operates independently of each other. A set of API known as RASC Abstraction Layer manages the data transfer between the host processor on SGI Altix system and the RASC RC 100 Blade. The block diagram in Figure 5 depicts the integration of the SGI RASClib (RASC Abstraction Layer) APIs and SNORT IDS for executing PCRE matches in hardware.

3.2 Compiling PCRE rules from SNORT rules

The SNORT IDS accesses the rules by rulesets when enabling PCRE based IDS. Thus each of the rulesets are available as separate files in the available SNORT Database. As a first step of processing, our compiler script extracts all rules from the SNORT database that have a pcre field and stores them into local ruleset files for further processing. These rules contain the various regular expressions which are used by SNORT IDS.

3.2.1 PCRE Compiler

Once the PCRE rules are segregated out from the SNORT Database, they are compiled through the PCRE compiler. The PCRE compiler software is part of the PCRE package and converts PCRE regular expressions into PCRE opcodes. The PCRE opcodes are kind of a machine code that defines the structure of the regular expression engine compiled from the PCRE regular expression.

3.2.2 PCRE Opcodes

The PCRE Opcodes are the instructions for the software based PCRE engine, and are defined in the pcre_internal.h file, and is part of the PCRE package. As an example Table 5 highlights important PCRE opcodes in the SNORT database. Each of the PCRE opcodes have an equivalent hardware description, i.e. the behavior of each of the opcode may be described in VHDL code, that is suitable for hardware implementation. Our second stage scripts is invoked to first extract caseful character matching PCRE opcodes for each regular expression. The caseful character matching blocks are implemented as simple text match circuits on hardware first. These blocks receive characters from the payload memory via the NFA controller circuit. Moreover when the repetition type PCRE opcodes are encountered, the blocks additionally are endowed with a local counter to count the number of occurrences of characters. As discussed earlier, since the hardware implementation on an NFA is inherently parallel in nature, it is possible to process one new character every clock cycle.

Table 5: PCRE opcodes corresponding to aforementioned Regular Expression operators

Regular Expression Operator	**PCRE Op Code**
Match after the first character "^"	OP_CIRC 19
Match after the last character "**$**"	OP_DOLL 20
Quantifiers "**{}**"	OP_EXACT 32
Ranged Quantifiers "**{n,x}**"	OP_UPTO 30
Negated Character Class "[^**...**]"	OP_NCLASS 60
Repetition "*****"	OP_STAR 24
Repetition "+"	OP_PLUS 26
Back References "**1**,**2**,**3**,**4**,**5**,**6** "	OP_REF 62

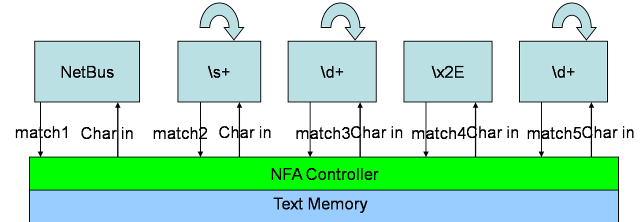

Figure 6: The NFA derived from a SNORT Rule.

3.2.3 Local Storage of Back References

The PCRE engine allow for back-references to be used primarily for the convenience of recollecting the marshalling / enclosing meta-characters which are useful for a multitude of payload containing programming language specific constructs. As an example some programming languages allow strings to be enclosed either in single quotes ' ' or double quotes " ". A backreference on the enclosing quotes would store the opening quote in the memory, and would recollect it while encountering the closing quote. Since the back references are essentially a type of addressible memory with the contents as the available backreferences, they may be stored on the FPGAs' on chip Block RAM. While generating VHDL code of the regular expression engine from the PCRE opcodes, our compiler allows utilizing the Block RAM for local storage of Back Reference thus saving on I/O costs to the SRAM on board, and allowing for faster hardware design.

3.2.4 Generating the NFA control structure

Our third stage compiler script combines the generated VHDL storage and counting blocks into a final entity that defines the actual regular expression engine on hardware. The opcode defining the regular expression operators are iteratively analysed by the script, and the control structure for the NFA is generated in VHDL. The control structure receives a new character every clock cycle and also is updated of the status of each storage and counter blocks. The NFA generating script converts the sequence of regular expression operator to a tree based hierarchical representation and thereafter the tree is parsed and converted to an extensive set of if - else statements in VHDL.

The NFA control structure is thereafter tied together with the storage and counter blocks in a single VHDL file which is tied together to the payload buffer via the memory interface module. Figure 6 depicts the internal organization of an NFA generated from a SNORT rule.

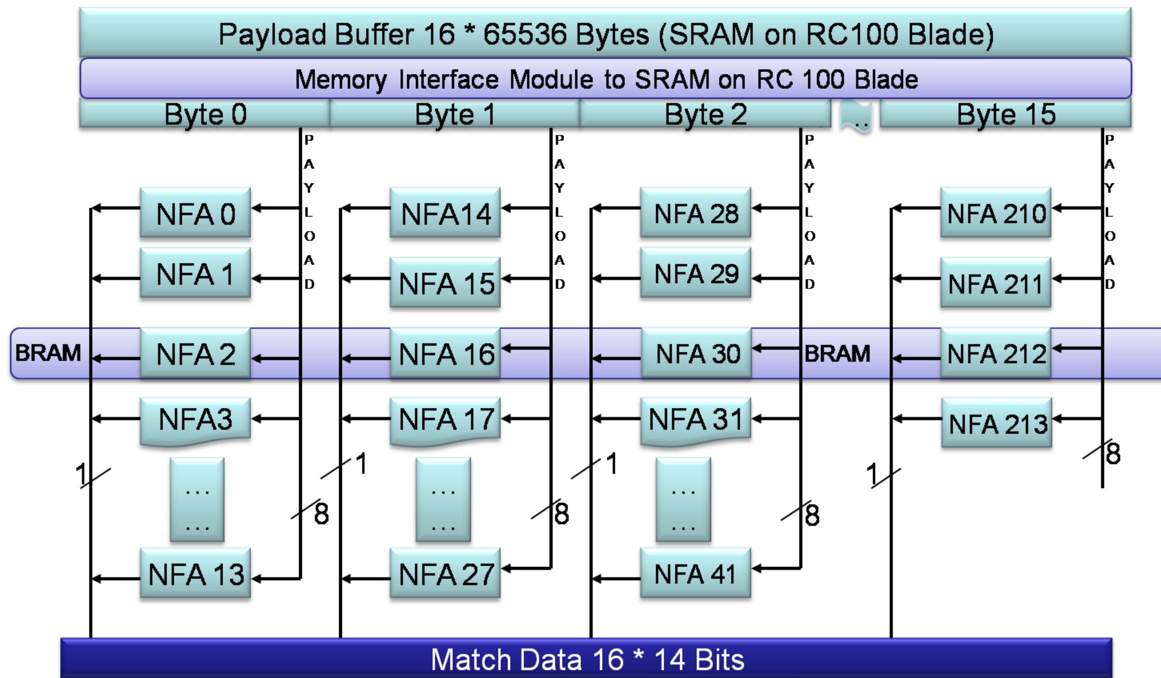

Figure 7: Architecture of parallel PCRE Engines on Virtex-4 LX 200 FPGA

3.2.5 *Common Prefix Optimization*

Since the SNORT IDS regular expressions are based on a collection of similar rules grouped by the rulesets, many of which contain regular expressions that share a common prefix. These prefixes are a potential point of design consideration which may lead to conservation of on-chip area, and are discussed in [18]. We generated an AWK script to extract common prefixes from regular expression in the oracle rule set of SNORT IDS, and compile them together into a single hardware block. The optimized design resulted in a savings of 26% area on the chip, but unfortunately that lead to a tradeoff on performance of more than 50%. As a result we chose not to enable the Common Prefix Optimizations to the regular expressions since it constrains the digital clock routes on the FPGA to operate at a lower frequency, thus affecting the throughput of the regular expression engines.

3.2.6 *PCRE engines on the FPGA*

PCRE engines from the SNORT rulesets are aggregated together in a single FPGA design, to be implemented and run in parallel on the FPGA. As depicted in Figure 7 there are two hundred and fourteen NFA engines which operate in parallel on a single FPGA chip. The NFA engines receive a character every clock cycle on the 8 bit payload line. The RASC RC 100 Blade allows for 128 bits to be retrieved from the Payload buffer each clock cycle, thus a total of sixteen separate payload lines are generated, that operate in parallel. As depicted in the architecture, the FPGA processes sixteen parallel payload threads, and each thread except thread 16 has fourteen parallel NFAs. The memory interface module is a SRAM memory controller, that interfaces to the Memory 1 on the RC 100 Blade. The Memory 1 serves as a Payload Buffer that receives the data worth sixteen payloads from the host CPU on the SGI Altix 4700. The payload buffer needs to store 16 * 65536 Bytes viz. 1,048,576 Bytes, since the maximum size of each payload is 64KBytes, and sixteen of them are processed in parallel on the FPGA. The FPGA based PCRE engines representing regular expressions with back-references are allocated memory space on the on-chip Block RAM, to store and retrieve the back-reference data. Each PCRE NFA engine generates a 1 bit output data that represents whether the payload matches the regular expression. A total of 224 bits are transferred back to the host upon completion of streaming of the packet through the PCRE engines on the FPGA. The 224 bit match data is then de-multiplexed at the host to obtain the individual match status of each of the regular expression engine.

4. EXPERIMENTAL RESULTS

Our compilation and hardware implementation of PCRE engines are compared against PCRE engines executed on software on actual SNORT rules. We run the experiment on a set of network dump of size 2.0 GigaByte. Our network dump file has been generated by running tcpdump on the SGI Altix 4700 supercomputer, over a period of time, and collecting the payload data in a single file. We load the payload file on the main memory of the Altix machine before sending it onto the RASC Blade. When evaluating the baseline software implementation of PCRE, we store the file on a ramdisk on a SGI workstation. We have created five project directories with 25, 50, 75, 100 and 200 regular expression derived from the backdoor, web-client, and spyware-put rulesets to allow us to obtain speedup data with varying number of regular expressions. Since any more than 200 regular expressions caused an overmapped FPGA, we had to limit our tests to 200 regular expressions.

For our software baseline testing, we utilize the aforementioned rulesets and create five project locations each with the

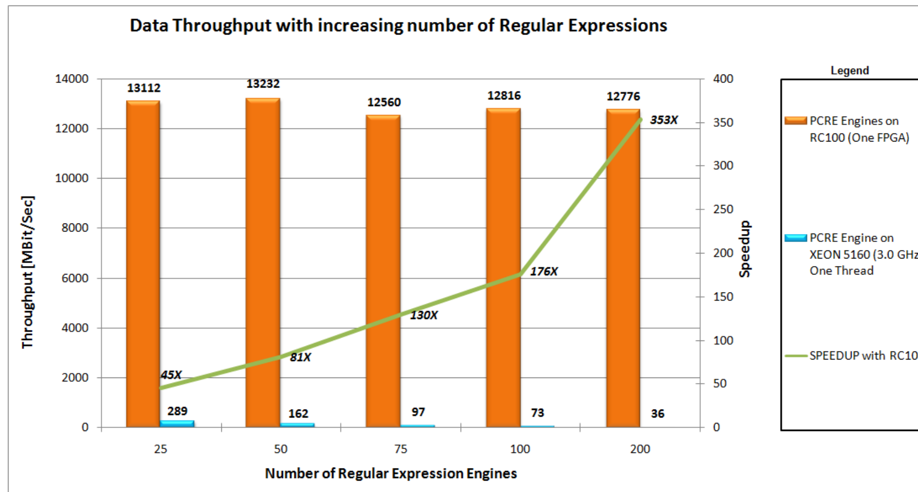

Figure 8: Speedup obtained by implementing PCRE engines on the SGI RASC RC100 Blade.

same set of rules that were implemented on hardware. Each of the project directory is accessed by a shell script that invokes pcre_compile on the regular expression and thereafter executes by invoking pcre_execute. The time measured using the benchmark does involve compilation overhead, since that duplicates the actual behavior of SNORT IDS. Each iteration of the experiment was executed 100 times to determine the average execution time, and hence the average throughput. We run the software baseline benchmarks on an Intel XEON 5160 (Woodcrest 3.0GHz 1333MHz Front Side Bus) based SGI Workstation, with 16 GB RAM. [1]

We utilize the three aforementioned rulesets for compilation into VHDL and thereafter implementation onto the FPGA hardware on the SGI RASC RC 100 Blade. The VHDL conversion of regular expression engines from each of the ruleset are collated together in a project location, along with the payload buffer and the memory interface module. We utilize the Xilinx ISE 9.1i synthesis tool namely 'XST' to synthesize the project into lower level hardware description known as the netlist. Thereafter the netlist containing the regular expressions and the payload buffer is mapped onto the Virtex-4 LX 200 FPGA, along with the SGI Core services using the 'MAP' tool. The next stage involves running the Place and Route (PAR) and Bitgen tools to finally generate a bitstream file that would configure the hardware resources on the FPGA. The bitstream file is copied to the bitstream repository, that would be accessible by the SGI Altix RASC daemon. All the FPGA compilation tools are run on the baseline SGI XEON 5160 workstation. The host code that reads the payload file, calls the RASClib APIs and receives the match data is a 'C' program. The SGI Altix 4700 at our site, utilizes two partitions, each with 32 Intel Itanium2 (Montecito 1.6GHz) processor cores, and 642 GByte main memory. The host code has been compiled using Intel C Compiler version 9.1.042, for optimum performance on the Itanium2.

In order to benchmark the performance of the hardware, we program the FPGA first with the bitstream containing 25 regular expression engines. Next we call the RASClib API's to send the 2.0 GigaByte worth of payload, to the RASC Blade, and obtain the match data for the 32,768 packets. We then repeat the experiment with each set of regular expressions, each time sending 2.0 GigaByte payload to the RASC Blade, and receiving the match data. Each iteration of the experiment is run 100 times and the average time of execution is recorded to obtain the average throughput. We include the bit-stream loading time overhead for each run of the benchmark. Since the RASC RC 100 Blade is connected to a NUMA Link interconnect on the Altix 4700 system, that is a shared memory system, the throughput data shows slight variability, which is dependent of a number of factors including the location of the memory that stores the payload, the actual system load at the time, etc.

As depicted in Figure 8, the speedup and throughput is charted for both the baseline case (cyan) bars and on the RASC RC 100 blade viz. the (orange) bars. The hardware design synthesized at a little more than 150MHz for all the five regular expression rulesets, and hence was clocked at 150 MHz. As the number of regular expressions on the hardware increased each packet was processed through many more regular expressions in parallel. Since the baseline software entails serial execution of the same payload over multiple regular expressions, the effective throughput declines with increasing number of regular expression engines. Therefore the speedup provided by the hardware increased linearly with increasing number of regular expressions when compared to the baseline software execution scenario.

5. RELATED WORK

Regular Expressions are a powerful method for generating rules and inspecting payload for IDS applications, including SNORT, BRO, and L7-filter.[4] [5][2].

Initial versions of SNORT started with string based pattern matching on a ruleset comprising of string matches. They were implemented on Software as well as improved versions were implemented on a variety of Hardware such as FPGA and ASICs. Suresh et al have documented a method to compile C code of bloom filter based text scanners to VHDL and achieve high throughput (18 Gbps) on Virtex II

[1] As discussed later, we are in the process of developing the required host code, that would help utilize the second FPGA on the RASC RC 100 Blade in a future implementation.

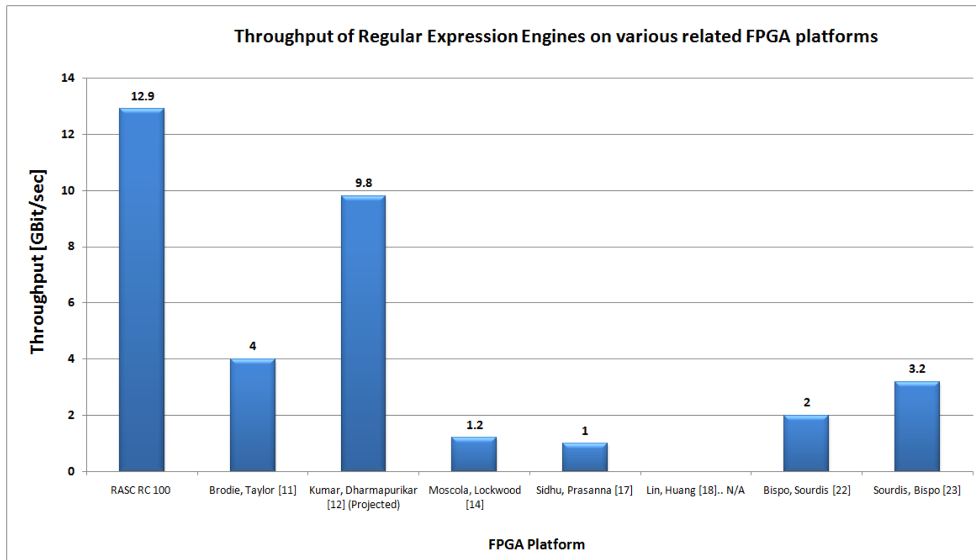

Figure 9: A comparison of throughput of regular expression engines on various related FPGA hardware.

FPGA. [3] Tan et al [1] detail on a high throughout design of the Aho-Corasick engine for string matching based IDS on Application specific silicon. Young H. Cho et al [8][9] detail out a silicon to implement a hardware based string matching coprocessor for SNORT IDS that runs at 7 Gbps. Their ASIC design provides a high performance platform for pattern matching. Baker et al [13] [7] demonstrate an FPGA implementation of the Knuth-Morris-Pratt algorithm for string matching suitable for IDS applications at 2.4Gbps

Current research initiatives have resulted in optimized Regular Expression engines in software which result in fast execution on CPUs. Yu et al suggest optimizing techniques on DFAs generated from regular expressions to reduce their execution times and achieve 50 to 700 times speedup[10]. But their method also asks for rewriting of the SNORT rulesets, which may not be supported by the community due to their adherence to PCRE standards. Kumar et al [12] have demonstrated graph theoretic algorithm to generate D2FA from DFA by combining multiple transitions in order to reduce the memory requirements of DFAs by more than 95%. Their design enhances Cisco network appliance by reducing embedded memory requirements.

FPGAs have been utilized on various Internet Firewall architectures, due to their ability to execute parallel regular expression based scanning engines. DFA based regular expression engines have been targeted towards FPGAs mainly for parallel execution on smaller FPGAs [14][15][16]. The authors utilize the JLex library to generate description of regular expressions from SpamAssassin rules. The authors also propose the use of DFAs by providing data on their compactness when compared to NFAs. Since DFAs can have only one active stage the hardware may be faster. Current research on NFAs used for Regular Expression Matching have resulted in optimization speed and area on FPGA. NFA implemented on a FPGA can process one character per clock cycle. Sidhu, Prasanna provide a highly detailed work on implementing and optimizing NFAs for use on FPGAs. In fact they propose a fast algorithm that generates the NFA on the FPGA hardware, rather than compiling it from software. [17] Brodie et al develop new hardware structures to implement FSM based regular expression engine. On their actual hardware test they obtain a 4Gbps sustained throughput on a 133MHz Virtex-II FPGA [11]. Lin, et al propose various optimization methods including prefix infix and postfix sharing of regular expressions on an older version of the SNORT ruleset.[18] Overall their methods bring about 20% reduction in on chip area, but its effect on clock speed is not discussed. Tiwari, et al propose utilizing Block RAM resources on the FPGA for storing LUT data, and hence free up LUT resources on the chip. It results in a savings of 26% power compared to LUT implementation of control logic for Finite State Machines.[21]. Bispo et al. [22] touch upon a VHDL generation scheme of NFAs from SNORT ruleset. They utilize extensive size optimization on NFAs including Prefix Sharing, Character Class Sharing and Static Pattern sharing. Their design results in a throughout of 2Gbps on a Virtex-4 FPGA, which suffice for 1GBE or slower network connections. Sourdis, Bispo, et al. [23] describe three building blocks that optimize constrained repetitions in regular expressions. They utilized their VHDL based hardware blocks to generate overall area efficient IDS systems on FPGA. The maximum throughput corresponding to their improved design is 3.2Gbps on a Virtex-4 FPGA.

As far as our contribution to the paper is concerned, direct compilation of PCRE opcodes into VHDL blocks and onto the FPGA is a novel approach to enhance throughput and as well as maintain compatibility with the community based rulesets. Figure 9 compares the throughput of various related FPGA platforms to the RASC RC 100 blade while executing regular expression engines. The performance of the RASC RC 100 blade is attributed to its high throughput connectivity to the host processor by exploiting the NUMA-Link fabric.

6. CONCLUSION AND FUTURE WORK

In this paper we described a method to directly compile PCRE opcodes generated from SNORT rules to VHDL and implement them on FPGA. We test proof-of-concept design

on a Virtex 4 LX 200 FPGA on a SGI RASC RC 100 blade. The design runs at 150 MHz and provides a throughput of 12.9 Gbps on tcpdump payload data on a number of SNORT rules ranging from 25 to 200. Our design performs between 45X up until 353X when compared to a baseline implementation on a Intel XEON 5160 CPU at 3.0 GHz.

At the moment we compile our design to only one FPGA on the RASC RC 100 blade. In the future we plan to incorporate the second FPGA too, and instantiate 200 more regular expression engines. Since each of the FPGA on the board has an independent NUMA Link interface to the host CPU, we may be able to effectively double the throughput of the hardware implementation.

RASC Abstraction Layer also provides a streaming interface to the FPGAs which would enable the host CPU to directly write to the FPGA. This could effectively enhance the throughput of the design, since the payload data is directly streamed to the NFAs rather than being written to the SRAM first and then being read by the payload memory. We plan to incorporate the streaming interface in order to speed up our design in the future.

7. REFERENCES

[1] Lin.Tan, Timothy Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," Proceedings of the 32nd International Symposium on Computer Architecture (ISCA 2005)

[2] "Application Layer Packet Classifier for Linux," Justin Levandoski, Ethan Sommer and Matthew Strait, http://l7-filter.sourceforge.net/.

[3] "Automatic Compilation Framework for Bloom Filter Based Intrusion Detection," Dinesh C. Suresh, Zhi Guo, Betul Buyukkurt, Walid A. Najjar, ARC 2006: 413-418

[4] SNORT IDS homepage, "http://snort.org"

[5] BRO IDS homepage, "http://bro-ids.org/Overview.html"

[6] Perl Compatible Regular Expressions(PCRE) library, "ftp.csx.cam.ac.uk/pub/software/programming/pcre"

[7] Zachary K. Baker, Viktor K. Prasanna, "A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs" Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004.

[8] Young H. Cho and William H. MangioneSmith, "A Pattern Matching Coprocessor for Network Security in " Proceedings of DAC 2005, June 13 - 17, 2005.

[9] Young H. Cho, Shiva Navab, and William H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," in Proceedings of 12th Conference on Field Programmable Logic and Applications 2002, pp. 452 - 461, Springer-Verlag.

[10] Fang Yu and Zhifeng Chen and Yanlei Diao and T. V. Lakshman and Randy H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems.

[11] Benjamin C. Brodie and David E. Taylor and Ron K. Cytron, " A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching, " in Proceedings of the 33rd annual international symposium on Computer Architecture, 2006.

[12] Sailesh Kumar and Sarang Dharmapurikar and Fang Yu and Patrick Crowley and Jonathan Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection", in Proceedings of the SIGCOMM '06 conference on Applications, technologies, architectures, and protocols for computer communications

[13] Zachary K. Baker, Viktor K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs" in Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays.

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

[15] John Lockwood, "An open platform for development of network processing modules in reprogrammable hardware", In Proceedings of IEC DesignCon'01, pages WB–19, Santa Clara, CA, Jan. 2001.

[16] John W. Lockwood and Naji Naufel and Jon S. Turner and David E. Taylor, "Reprogrammable network packet processing on the field programmable port extender FPX", in Proceedings of ACM International Symposium on Field Programmable Gate Array (FPGA'2001), pp 87-93, Feb 2001.

[17] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching using FPGAs", in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines April 2001.

[18] Cheng-Hung Lin and Chih-Tsun Huang and Chang-Ping Jiang and Shih-Chieh Chang, "Optimization of regular expression pattern matching circuits on FPGA," in Proceedings of the conference on Design, automation and test in Europe 2006.

[19] W. Feng and P. Balaji and C. Baron and L. N. Bhuyan and D. K. Panda, "Performance Characterization of a 10-Gigabit Ethernet TOE," in Proceedings of the 13th Symposium on High Performance Interconnects 2005.

[20] SGI Document Number: 007-4718-006 , "Reconfigurable Application-Specific Computing User's Guide", "http://techpubs.sgi.com"

[21] Anurag Tiwari and Karen A. Tomko, "Saving Power by Mapping Finite-State Machines into Embedded Memory Blocks in FPGAs," in Proceedings of the conference on Design, automation and test in Europe 2004.

[22] João Bispo and Ioannis Sourdis and João M. P. Cardoso and Stamatis Vassiliadis, "Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues.," in Proceedings of ARC 2007.

[23] I. Sourdis and J. C. Bispo and J. M.P. Cardoso and S. Vassiliadis, "Regular Expression Matching in Reconfigurable Hardware," in International Journal on VLSI and Signal Processing