

Deep Network Packet Filter Design for Reconfigurable Devices

YOUNG H. CHO

and

WILLIAM H. MANGIONE-SMITH

University of California, Los Angeles

Most network routers and switches provide some protection against the network attacks. However, the rapidly increasing amount of damages reported over the past few years indicates the urgent need for tougher security. Deep packet inspection is one of the solutions to capture packets that can not be identified using the traditional methods. It uses a list of signatures to scan the entire content of the packet; providing the means to filter harmful packets out of the network. Since one signature does not depend on the other, the filtering process has a high degree of parallelism. Most software and hardware deep packet filters that are in use today execute the tasks under Von Neuman architecture. Such architecture can not fully take advantage of the parallelism. For instance, one of the most widely used network intrusion detection systems, Snort, configured with 845 patterns, running on a dual 1-GHz Pentium III system, can sustain a throughput of only 50 Mbps. The poor performance is due to the fact that the processor is programmed to execute several tasks sequentially instead of simultaneously. We designed scalable deep packet filters on field programmable gate arrays (FPGAs) to search for all data independent patterns simultaneously. With FPGAs, we have the ability to reprogram the filter when there are any changes to the signature set. The smallest full pattern matcher implementation for the latest Snort NIDS fits in a single 400k Xilinx FPGA (Spartan 3 - XC3S400) with a sustained throughput of 1.6 Gbps. Given a larger FPGA, the design can scale linearly to support a greater number of patterns as well as higher data throughput.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General—*Security and protection*; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Security, Design

Additional Key Words and Phrases: Firewall, Network Intrusion Detection, String Filter, Virus, Worm

1. BACKGROUND

Today, any computer connected to the Internet is in danger of being attacked by hackers. A recent series of malicious attacks resulted in large economical damage. Worms such as “MS Blaster” and “SoBig-f” caused a tremendous amount of damage; it was estimated that “SoBig-f” alone accounted for \$29.7 billion in damages

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0001 \$5.00

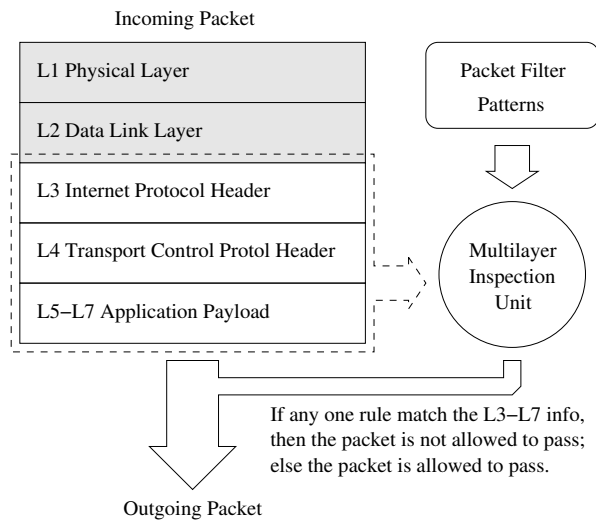


Fig. 1. Deep Packet Inspection

worldwide [Gaudin 2003].

In attempt to limit the damage, network appliances have built-in firewalls. Most of these firewalls only have the ability to examine network packet headers. However, application-level attacks are usually hidden in the payload of the packets. Therefore, these attacks can pass through the firewalls undetected [Allen et al. 2001].

1.1 Advanced Network Intrusion Detection and Prevention

The purpose of a network intrusion detection and prevention system (NIDS/NIPS) is to protect computers from network attacks [McHugh et al. 2000; Allen et al. 2000]. Advanced forms of NIDS use an in-depth packet scanning technique called deep packet inspection (DPI). As shown in figure 1, the DPI system searches for the patterns in all layers of the packet. Given a pre-defined signature set, the DPI system is able to detect previously undetectable attacks. For example, an open source NIDS, Snort, can be configured to detect the “Sobig-F” worm by searching for the specific 76 byte pattern in the payload [Roesch 1999; Cho et al. 2002; Cho and Mangione-Smith 2004].

Effectively, the DPI based system performs packet header classification and pattern search at every alignment of packet payload. Layers 3 and 4 consists of standardized protocol fields while layers 5 through 7 are comprised of various application specific data. The protocol fields can be used to classify network packets. In order to increase the accuracy of packet identification, application specific data is also scanned for a set of patterns. Although this system offers higher flexibility and accuracy in detecting attacks, computationally intensive pattern search tasks dramatically increase system cost per performance [Viacom Inc. 2001].

Fast packet header classifier is an essential part of deep packet inspection. The most intuitive classification method is to directly compare header fields against a known set of attributes. Since the locations of header fields are predefined, compar-

isons can be done using discrete comparators, content addressable memory (CAM), and hash functions. Hashing functions along with memory can be the most space efficient system. Although hashing can introduce false positives, sometimes it is the most practical solution that can accelerate the classification. Without an efficient header classification, dynamic inspection engines may process the packet payload unnecessarily. Therefore, a more efficient and accurate DPI system can be built using header classifier running in conjunction with the payload pattern matcher. Since the main focus of this paper is architecture and design techniques for high performance reconfigurable pattern matching engine, the rest of the paper will not address the design or the effects of header classifier.

Inspection of packet payload requires much more flexibility and computation. It is often the case that patterns must be matched at every byte offset. In practice, several attack signatures share the same class of header. Therefore, multiple patterns need to be matched against the payload of each packet. Scanning N different patterns of length L bytes over payload of P bytes requires $N \times ((P - L) \times L)$ byte comparisons in the worst case. In Von Neuman architecture, each operation can translate to multiple instructions of memory load, shift, compare, and branch. Given this requirement, the payload inspection task requires much more processing resources than the header classifier. Therefore, in most DPI NIDS, this task is usually the performance bottleneck of the entire system. One can drastically increase the system performance by accelerating the payload inspection process.

1.2 Commercial Systems

There are a few commercially available NIDS which incorporate the DPI processes. Companies such as Netscreen, SourceFire, and CISCO produce firewalls that employ a form of deep packet inspection [Dubrawsky 2003]. These products use one or more processors running rule-based NIDS software. It is difficult for the sequential processors to filter the network traffic above gigabits per second [Roesch 1999; Karagiannis 2001]. Therefore, most of the DPI system uses some form of heuristics to approximate the scanning.

Some products use high-speed network security co-processors that have rule-based deep packet filter accelerators. These co-processors can be configured as packet classifiers, network monitors, and rule-based firewalls. These devices are rated to support network bandwidth on the order of gigabits per second under favorable conditions. Among these, StrataSwitch II from Broadcom and ClassiPI from PMC Sierra are popular chips for 1+ gigabits per second network.

StrataSwitch II is capable of filtering at the line-rate up to 10 gigabits per second on all of the ports with an average packet size of 80 bytes. The layer 2 to layer 7 processing is done in the Fast Filter Processor using a set of customizable filter rules. Due to its efficient static field comparator, the chip can be programmed to be an effective packet classifier. However, due to the task that requires dynamic pattern searches, it is a poor candidate as a deep packet filter. The performance of the filter also decreases as the number of rules grow. For a few rules, the co-processor is able to sustain the gigabits per second bandwidth, but for hundreds or even thousands of rules, the chip is not even be able to support fast Ethernet. Other limitations with StrataSwitch is its fixed 80 byte search window [Broadcom Inc. 2001].

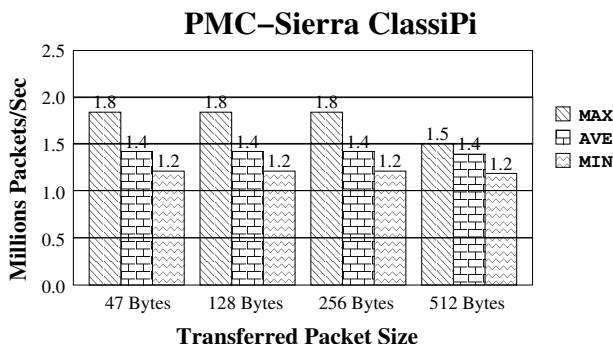


Fig. 2. Benchmark for ClassiPi deep packet Inspection Chip

ClassiPi is another device that can be used to perform pattern searches over all the layers of the packet. Like StrataSwitch II, it uses a set of rules to filter network traffic. According to the company benchmark shown on figure 2, the PM2329 chip is able to scan several thousands of Ethernet addresses, IP-addresses, and other TCP header information with an average throughput rate of 1.4 million packets per second. With an average packet size of 256 bytes, the bandwidth of the network can reach 2.8 Gbps. However, its throughput reduces to 800 Mbps when four 12-byte strings searched using a 112-byte search window.[PMC Sierra Inc. 2001]. The reason for the performance drop is the sequential instruction execution. Like most general purpose processors, these co-processors are based on the Von Neuman architecture.

1.3 Academic Research Systems

Researchers have looked into several applications, such as network processors [Iliopoulos and Antonakopoulos 2000], switches, routers [Braun et al. 2001], and protocol wrappers [Fallside and Smith 2000; Braun et al. 2002; Dowd et al. 1997; Sinnappan and Hazelhurst 2001]. Projects such as PLATO of the Technical University of Crete [Dollas et al. 2001], FPX of the Washington University [Lockwood 2001], and Myrinet FPGA nodes from Myricom [Sivilotti et al. 1998; Cho 2000] are reconfigurable design platforms.

Recently several academic researchers have demonstrated high performance pattern matching engines using FPGAs. Some implementations used programmable embedded memories in FPGAs to build a fast pattern matcher. Washington University implemented a pattern detector using hashed index memory lookup. With the use of Bloom filters [Bloom 1970], their circuit can detect a large number of patterns at 600 Mbps using embedded memories. Since bloom filter uses several hash functions to detect patterns, it is possible to have false positives. Also, the system cannot tell which exact pattern is detected with bloom filter alone. Therefore, all detected patterns need to be explicitly compared against a set of strings to further refine the result [Dharmapurikar et al. 2003; Lockwood et al. 2003].

There have been several efforts to match strings using content addressable memories (CAM) [Gokhale et al. 2002; Yu et al. 2004; Yusuf and Luk 2005]. The Granidt project of Los Alamos National Laboratory implemented another type of

pattern search system using CAM [Gokhale et al. 2002]. Yu of UC Berkeley used Ternary CAM (TCAM) to extend the pattern matching functionality to allow variable length [Yu et al. 2004]. CAM essentially performs pattern matching as part of its function. However, it is very expensive in terms of power and size. Therefore, most designers using FPGAs focus on optimizing the bandwidth and size of the fixed logic implementations [Singaraju et al. 2005].

Building discrete logic version of the pattern matcher using FPGA started in 2001. Sidhu and Prasanna mapped Non-deterministic Finite Automata (NFA) for regular expression into FPGA to perform fast pattern matching [Sidhu and Prasanna 2001]. Franklin and Hutchings implemented a pattern search engine in JHDL [Franklin et al. 2002]. We used chains of 8-bit decoders to build fast pattern match engines that can sustain a bandwidth of 3.2 Gbps [Cho et al. 2002]. Moscola and Lockwood translated regular expressions into deterministic finite automata (DFA) showing that, most DFAs can be reduced to a fast and compact circuit [Moscola et al. 2003]. Sourdis mapped a similar design with a deeper pipeline to increase the filtering rate up to 10 Gbps [Sourdis and Pnevmatikatos 2003]. Follow-up works by Cho, Clark, Baker, and Sourdis made a contribution in reducing the size of the design by eliminating duplicate logic [Cho and Mangione-Smith 2004; Clark and Schimmel 2004; Baker and Prasanna 2004; Sourdis and Pnevmatikatos 2004] and by integrating hashing with memory based matching techniques [Cho and Mangione-Smith 2005; Papadopoulos and Pnevmatikatos 2005; Sourdis et al. 2005]. These improvements allowed the large pattern set contain over 20,000 characters to fit into a single FPGA with look-up-tables (LUTs) to byte ratio below 1. These designs are all constructed using parallel pattern matching engines, therefore they maintained high throughputs irrespective of payload content.

In order to experiment and compare the designs, most academic systems used signature sets for an open source NIDS called Snort. The signature set contains over 2,000 fixed string patterns of network attacks and is updated regularly.

1.4 Snort Intrusion Detection System

Snort is an open source software NIDS. It is able to perform traffic analysis, IP packet logging, protocol analysis, and payload content search. Furthermore, Snort can be configured to detect a variety of abnormal packet behaviors, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and OS fingerprinting attempts.

Like the commercial systems, Snort uses a set of rules to filter the incoming packets. The Snort signature database consists of patterns for the known attacks. The simple rule structure allows flexibility and convenience in configuring Snort. However, its performance is inversely proportional to the number of rules in the database. Because it is an open source system, many academic researchers have used its signature set to perform experiments on their own system.

1.4.1 Packet Inspection Signatures. Snort uses a detection engine that utilizes a modular plug-in architecture. There are three primary subsystems that make up Snort: the packet decoder, the detection engine, and the logging and alerting subsystem. Snort maintains its detection rules in a two dimensional linked list of headers and options.

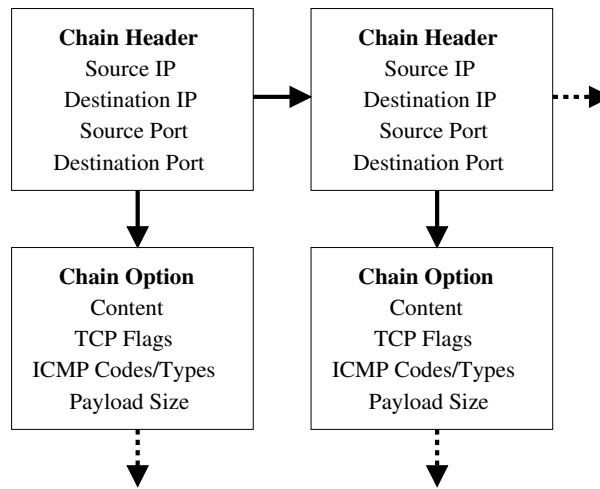


Fig. 3. Snort Rule Chain Logical Structure

As shown in figure 3, the rule chains are compared against incoming packets. First, the network packet headers are sequentially compared with the Chain header information. When there is a match, the search routine recursively searches through the Chain options, including the patterns in the payload. After the comparisons, the packet header is subsequently compared with the rest of the Chain headers. The detection engine checks only those options that have been set by the rule parser during the run-time. The first rule that matches a decoded packet in the detection engine triggers the action specified in the rule definition and returns.

The structure of a rule consists of a command keyword for handling the matching packet, the header information, and various options for other patterns such as the content string for the payload. Snort can use one or more rule files as its input. Each rule file can contain more than one rule signature with the format shown as following.

```
Action Protocol SrcIPAddr/Port Direction DstIPAddr/Port Options
```

The first part of the rule corresponds to the packet header information. The second part corresponds to the pattern search options applied to the application payload. The most computationally intensive option is called ‘content.’ This option is the key to better packet filtering in deep packet inspection firewalls.

The following rule is a signature that is used by Snort to detect ‘Code Red’ worm.

```
alert TCP 128.142.4.10 any -> 198.162.10.1 80 (msg: ‘IDS552/web-iis_IIS
ISAPI Overflow ida’; dsize: >239; flags: A+; uricontent: ‘.ida?’;
classtype: system-or-info-attempt; reference: arachnids,552;)
```

When the signature is loaded on to Snort, the system will alert the administrator if the packet under examination has matching protocol, IP addresses, ports, and other packet characteristics describe within the parenthesis. The above rule will cause the system to specifically search for the pattern “.ida?” in all the payloads of the packets that match the header specifications in the rule signature.

1.4.2 *Performance Bottleneck.* Snort is designed to run on general purpose Von Neuman processors. On the Celeron 733 system, the peak performance of Snort with 105 signatures sustained a maximum of 50 Mbps regardless of the contents of the network traffic [Cho et al. 2002]. For 845 rules, we found that a dual 1-GHz Pentium system is needed to sustain the same bandwidth. Under normal network traffic conditions, the filtering bandwidth of 50 Mbps may be sufficient for a fast Ethernet network. However, if the aggregate bandwidth of the network is above 50 Mbps, the packets are dropped by Snort. This result is mainly due to payload inspection tasks of Snort. Although this dynamic payload inspection task is computationally intensive, it can be divided into several independent tasks that can run in parallel. In light of this, we develop a hardware deep packet filter using FPGA to overcome the performance bottleneck.

1.5 Paper Organization

The rest of the paper is organized as follows. In section 2, we discuss the basic design of reconfigurable pattern matching engine. Then we leverage on the reconfigurability of FPGA logic and algorithm to reduce the size of the overall system in section 3.

2. HARDWARE PAYLOAD INSPECTION

The Snort signatures contain information to search through all layers of the network packet. Header information in layers 3 and 4 are used for classifying the packets. Then an exhaustive pattern search is performed on the payload (layers 5-7) to filter out all the malicious attacks. [Roesch 1999]. Since payload inspection is the performance bottleneck of our application, we present a fast reconfigurable hardware pattern matcher and index generator to accelerate the system.

2.1 Reconfigurable Pattern Matcher

The hardware pattern match unit is a chain of comparators that compare the incoming data against predefined patterns.

The pattern matching is shown in figure 4. Since the logic we use is reconfigurable, the byte comparators can be implemented with a 8-to-1 bit decoder. The chain of the comparator unit can be instantiated in parallel to increase the bandwidth of the pattern matching unit. We will step through the inspection processes in the 4-byte matching unit to introduce and verify the performance scalability of the design.

2.1.1 *Scalable Design.* The data throughput of the design can be increased by widening the width of the data input. We present an efficient method to linearly scale the pipelined pattern matcher.

Figure 5 is simplified example of the design. The squares outside of the content match unit are 8-bit registers. The solid squares show the initial state while the dotted line squares are the subsequent future states of the registers. The values in the shaded squares represent “don’t care” while the rest of the squares contain valid ASCII characters. The squares within the content match unit represent 8-bit comparators. As shown in the figure, the four 8-bit comparators work in parallel to match four consecutive bytes of data simultaneously. Then their results are passed to the 1-bit registers located below all the comparators for timing purposes.

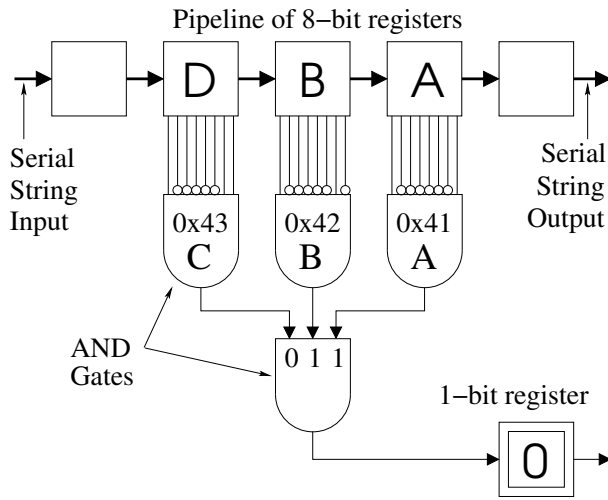


Fig. 4. Simple String Search Engine for “ABC”: Since the input substring is “ABD”, the comparator detects no match.

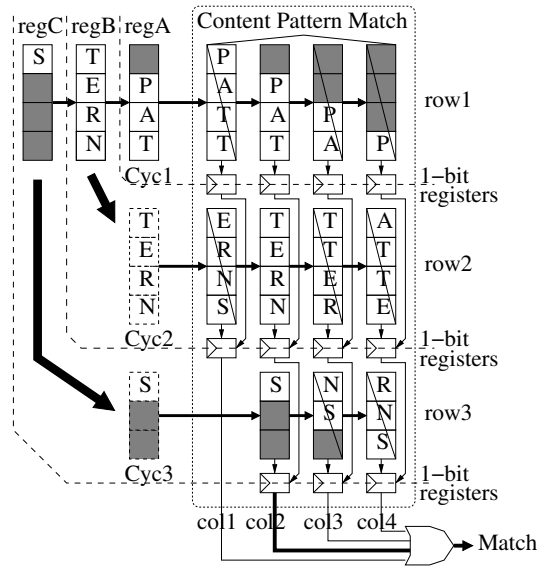


Fig. 5. Pattern Match Example

Output from the 1-bit register controls the subsequent 1-bit comparison result. The content of the register A, B, and C represent the string stream ‘PATTERNS.’ Since the registers are pipelined, we can separately observe different parts of the design during each clock period. The matching stages are indicated as clock cycles 1, 2, and 3.

During cycle 1, substring ‘PAT’ is compared against four different strings with different byte offsets in row 1. Since all of the 1-bit registers are initialized as 0, the

ACM Journal Name, Vol. V, No. N, Month 20YY.

result will not be latched even if there is a matching pattern in rows 2 and 3. For our example, the comparator in row 1-column 2, indicates the match. This value is latched through the 1-bit register to enable the next row of 1-bit registers.

In the following clock cycle, the substring in register B is latched into register A. The substring is then matched with all the patterns. However, the only comparator results that can be latched are all the comparators in row 1 and the comparator in row 2, column 2. The purpose behind enabling all the row 1 registers is because of a possibility that the pattern may have actually started with the content in register B during cycle 1. The matching pattern in row 2, column 2, enables the 1-bit register in row 3, column 2 for the cycle 3.

Finally, with the content of register C in register A in cycle 3, substring ‘S’ is compared with comparators in row 3. The match in the column 2 comparator sends 1 to the OR gate which indicates the match of the substring ‘PATTERNS.’ This signal then can be used to alert the user or to drop the packet.

All the contents for the rules are compared in parallel. Therefore the design size will grow proportionally to the number of the rules. Because each rule forms an independent pipelined unit, the performance of the design remains approximately the same regardless of the design size. Since all the comparators compare the incoming stream of data with a fixed value, it is unnecessary to keep the pattern values in a register. Rather, the effect of comparator can be achieved by using a 32-bit AND gate with inversion in the input lines corresponding the zero bits of the bit pattern. Therefore the hardware requirement is much smaller than implementing comparators.

2.1.2 Parallel Search Engines. Snort consists of over 2,000 different patterns. In hardware, each of these patterns can be made into independent pattern matchers. Figure 6 is a block diagram of our design for Snort NIDS. Each rule unit implements the logic for a single Snort rule signature. Packet data is passed to the units through a 32-bit bus.

With closer inspection, we find that the design resembles the Snort software execution path. The header information of each packet is compared with the predefined header data. If the header information matches the rule, the payload is sent to the content pattern match unit where the dynamic pattern is searched. However, unlike the software, all the rule chains are matched in parallel to achieve much higher performance.

In addition to the parallel rule units, the four bytes of data are matched in each stage of the pipeline to increase its throughput. As it will be presented in the next subsection, the parallel instances of the search string are used to match four patterns of different byte alignment.

If the packet is malicious, the system is configured to raise an identification flag. In a system with sufficient buffer memory, the flag can be used to either alert the user or drop the packet. Otherwise, the flag can be used to corrupt the rest of the packet payload going through the pipeline; thereby causing the network to drop the packet.

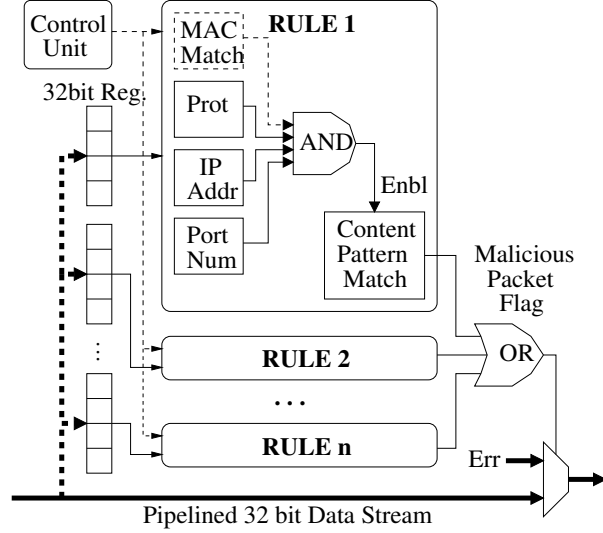


Fig. 6. Parallel Datapath of NIDS in Reconfigurable Hardware

2.2 Signature Index

The inspection module for each pattern produces a 1-bit output to indicate the match at that clock cycle. It may be sufficient in some applications to simply indicate the match, with identification accomplished in software. However, it is often more desirable to produce the corresponding signature index number.

A small index encoder module can be written in VHDL as a chain of CASE statement. However, a VHDL with thousands of CASE statements do not translate efficiently in most synthesis tools. Therefore, naively written VHDL of a large encoder is almost always the critical path of the entire system.

2.2.1 Simple Index Encoder. For the purpose of generating a compact hardware, we assume that only one input pin will be asserted at any clock edge. With this assumption, the address encoder can be built using the combinations of outputs from the binary tree of OR gates like the one in figure 7.

Based on the natural characteristic of the binary tree, we determine that each index bit should be asserted if any of the odd nodes on the corresponding level of the tree is asserted. For example, a four bit index encoder for a 15-input encoder is written as equations 1 through 4.

$$Index_3 = a_1 \quad (1)$$

$$Index_2 = b_1 + b_3 \quad (2)$$

$$Index_1 = c_1 + c_3 + c_5 + c_7 \quad (3)$$

$$Index_0 = d_1 + d_3 + d_5 + d_7 + d_9 + d_{11} + d_{13} + d_{15} \quad (4)$$

A static pattern search unit working with a dynamic search unit would most likely cause only one of the input pins to be asserted at any cycle. However, there is still

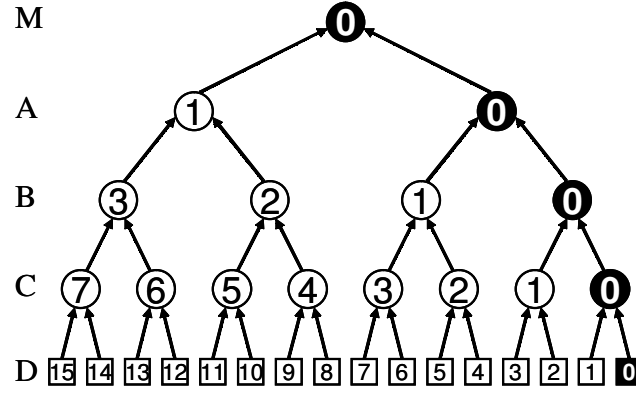


Fig. 7. A binary OR tree - logic equations indices are derived using the OR tree. All the unused gates are trimmed off to save hardware resource.

a possibility that a search engine will detect more than one pattern at one time. In this situation, the previously described index encoder may not be sufficient. One solution to the conflict is to divide the set into multiple sets; each subset containing non-overlapping patterns. Then each sub-set pattern search unit can have its own index encoder. In some case, a limited output pin count may make it infeasible to have multiple index output. Therefore, it is often desirable to assign priority to each pattern index.

We have developed two methods for assigning index priorities. The first method uses software pre-processing to assign priority numbers to the patterns. The other method directly modifies the hardware to implement index priority.

The problem of simultaneous pattern detection is only seen when two or more input pins are asserted at the same time. When multiple input pins are asserted, our encoder applies a bit-wise OR to all the index numbers. Therefore, at the output pin, we will see the combined output of two or more indices which may not give any indication of detected indices. By pre-processing and reassigning the index orders, one can effectively give priority of one pattern over another.

For all the patterns that can assert encoder inputs at the same time, one can assign index numbers to satisfy equation 5; which applies bit-wise OR to all the indices where I_n is an index number with an higher value of n indicating the higher priority.

$$I_n | I_{n-1} | \dots | I_0 = I_n \quad (5)$$

Once all the indices are assigned for the overlapping patterns according to the desired priorities, the indices can be assigned to the rest of the patterns. In this method, there is a limitation place on the size of prioritized pattern set. The maximum number of indices for each set is equal to the number of index output pins. There may be several independent sets of overlapping patterns in a realistic configuration, but their number of simultaneously overlapping patterns are usually less than four. Therefore, for most sets, this technique is sufficient. The advantage of using this method is that no additional gates are needed.

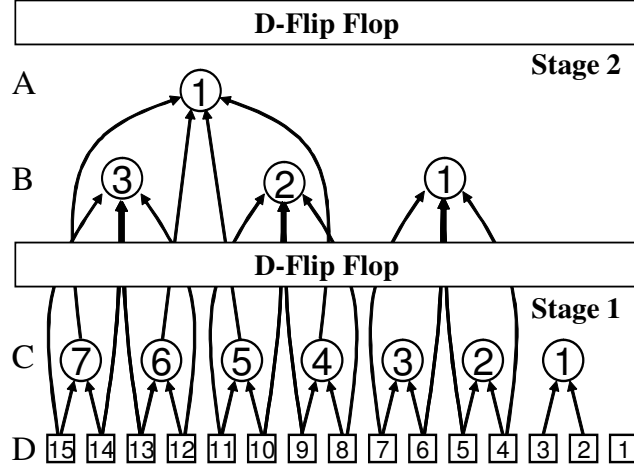


Fig. 8. Highly pipelined binary OR tree - two 2 input gates are collapsed into one 4 input gates to reduce the gate count. Pipelining is applied for single gate level of critical path.

2.2.2 Hardwired Priority Index Encoder. For situations where strict priorities are needed, we must modify the binary OR tree to enforce priority. In a binary OR tree implementation described in the previous subsection, the most significant bit of an index is “1” when any of the D nodes under A_1 node is asserted. Since the higher numbered D nodes have the priority over the lower, the output of A_0 node need not be considered. Therefore, the most significant bit value is assigned as the output of A_1 . For the next address bit, we consider branches with nodes that are immediate children of A_1 and A_0 . We can deduce that the second index bit is “1” if the output of B_3 is asserted. But this time, we also find that the index bit is “1” if the output of B_3 is “1” while none of the D nodes under A_1 is asserted; we only need to check that A_1 is “0” to guarantee that none of its children nodes are asserted. Using this method, equations for less significant bits can be constructed. We apply this method for a 15-bit input to extract index bit equations 6 to 9.

$$Index_3 = a_1 \quad (6)$$

$$Index_2 = b_1 \cdot \bar{a}_1 + b_3 \quad (7)$$

$$Index_1 = c_1 \cdot \bar{b}_1 \cdot \bar{a}_1 + c_3 \cdot \bar{a}_1 + c_5 \cdot \bar{b}_3 + c_7 \quad (8)$$

$$Index_0 = d_1 \cdot \bar{c}_1 \cdot \bar{b}_1 \cdot \bar{a}_1 + d_3 \cdot \bar{b}_1 \cdot \bar{a}_1 + d_5 \cdot \bar{c}_3 \cdot \bar{a}_1 + d_7 \cdot \bar{a}_1 + \quad (9)$$

$$d_9 \cdot \bar{c}_5 \cdot \bar{b}_3 + d_{11} \cdot \bar{b}_3 + d_{13} \cdot \bar{c}_7 + d_{15}$$

With registers at the output encoded address bits, the critical path has a maximum of $(\log n)-1$ gate delays where n is the number of the input pins. Since pattern search structures are pipelined after every gate, this long chain of gates becomes the critical path.

Each LUT in most FPGAs are usually paired with a D-flipflop. Therefore our design of the encoder inserts additional pipeline registers. The figure shows the

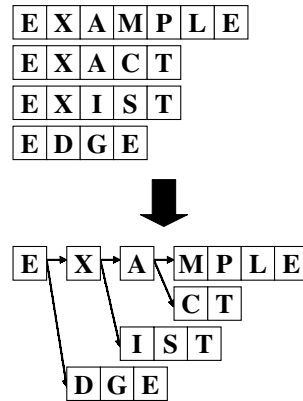


Fig. 9. An example of Aho and Corasick's Keyword Tree: 6 bytes are optimized away.

pipelined OR tree with most 2-two input gates replaced by a single four input gate followed by D-flip flop. The logic for index equation is also further pipelined to maintain a single level of gate between the pipelined registers.

3. OPTIMIZING RECONFIGURABLE LOGIC

3.1 Discrete Logic Optimizations

The main design challenge for FPGA pattern matchers with large signatures is the design size. There are 2,207 signatures with over 22,000 bytes of pattern in the Snort rule set as of March 2004. Since a single byte comparator requires at least three four input look-up-tables (LUT) and one D-flip flop (DFF), resource requirements for a full system is at least 66,000 LUTs for 8-bit and over 260,000 LUTs for a 32-bit version [Cho et al. 2002]. Mapping this on to 20,000 LUT FPGAs would require more than 13 FPGAs.

To approximate the resource usage for the basic design, we attempted to compile the entire Snort pattern set using the Xilinx FPGA tools. Unfortunately, during the mapping stage of the design compilation, Xilinx ISE tool ran out of memory (2 GB maximum for 32-bit Operating System) after 24 hours of compilation. We tried to compile the design for the Altera FPGA, but ended with the same result. We could not exhaustively test for the limitations of the CAD tools because of the lengthy compilation time. Even if the design compilation is possible, using multiple FPGAs is not practical. Therefore, we sought for opportunities to reduce the hardware size by leveraging on the reconfigurable architecture and the data-specific optimizations.

3.1.1 Aho and Corasick Keyword Tree. Aho and Corasick's keyword tree [Aho and Corasick 1975] is used in many software for efficient pattern search. In fact, Snort NIDS applies the fast pattern search techniques based on the algorithm [Desi 2002]. We can apply the same concept in hardware to reduce the size of the pattern search engine.

As shown in figure 9, the keyword tree is a way to store a set of patterns into an optimized tree of common keywords. In software, the tree not only reduces the amount of required storage, but it accelerates the matching process by narrowing

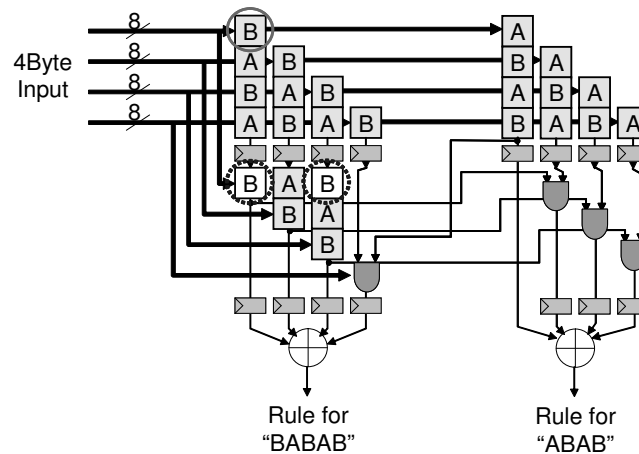


Fig. 10. Reusing Common 8-bit Comparators: All the common 8-bit comparators in the same alignment are combined.

down the number of matching patterns as the algorithm scans through the keyword tree.

A similar benefit can be achieved when it is applied to hardware design. The search pattern set of our design can be formed into keyword trees. Once the keyword tree is generated, it is converted into the pattern search tree with the same function as the parallel. The comparator and register in the engine can be seen as memory usage in software. Therefore, by applying this technique, the size of the design can be reduced by allowing more than one pattern to reuse comparator logic. As mentioned in a previous section, the bandwidth can be scaled by widening the width of the input. Constructing the keyword trees in the scaled version of a pattern matcher requires that the length of each keyword be in multiples of the datapath width.

3.1.2 Byte Decoders. Each byte comparator made with a combination of two 4-input LUTs. Since several comparators are generated as parallel units, many duplicate comparators can be reused. Eliminating duplicate logic at the gate level is an obvious optimization technique. Therefore, most synthesis tools for ASIC and FPGA compilers already have options to apply such techniques. However, automatically detecting duplicate logics are difficult and time consuming process. For NIDS application, this technique can be much easily applied to the circuit by the design generator to aid the synthesis tool to yield the most area efficient design. Clark and Schimmel [Clark and Schimmel 2003] were first to apply this technique in building their NIDS. Many other NIDS designers have applied similar techniques to reduce the size of their design [Cho and Mangione-Smith 2004; Sourdis and Pnevmatikatos 2004; Baker and Prasanna 2004].

Figure 10 illustrates how the optimization technique is applied to the circuit. The output of the “B” comparator in the first pipeline stage is reused for two other comparators in the second pipeline stage. After eliminating all duplicates in the figure, we find that we can build a functionally equivalent design with only eight

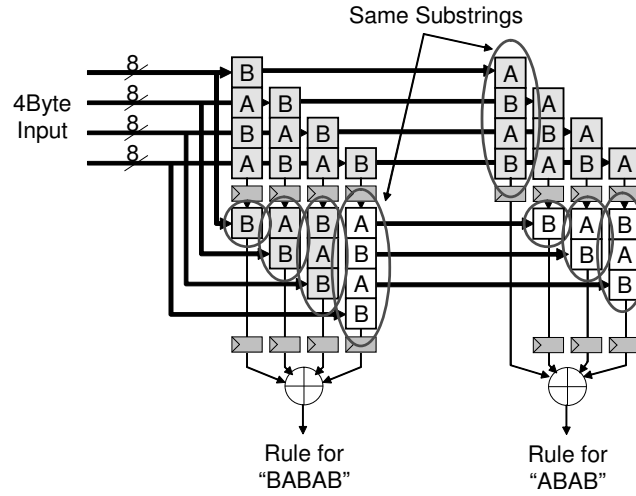


Fig. 11. Reusing the Common Substrings: All the matching substrings with same alignment are combined.

8-bit comparators. As with the example, we found that combining comparators in to a single decoder module reduced the design size down to 25 % of the original design.

3.1.3 *Common Substring Comparators.* Filtering the stream of data through wide bus requires inspection of the same data at each byte alignment of the bus. Therefore, the 4-byte bus datapath contains four modified 1-byte inspection modules for each target pattern in the signature set. As a result, each module consists of several 1 to 4 byte wide comparators in each stage of the pipeline.

The design requires that all comparators for the same byte alignment to be connected to the same input pins. Since all the 4-byte inspection modules run in parallel, some comparators are used to check for the same substring. Therefore, the duplicate comparators can be removed without losing any design functionality.

The first step of eliminating duplicate comparators is to divide every target string into a set of 1 to 4-byte segments. Then a set of unique segments is extracted from the original set. All the unique string segments are then translated into comparators eliminating the duplicate components. Finally, the outputs of the comparators are forwarded to the corresponding pattern match modules.

Figure 11 shows that there are four pairs of duplicate comparators for 4-byte pattern inspection modules for “BABAB” and “ABAB”. Therefore, an extra comparator for each of “ABAB”, “BAB”, “AB”, and “B” can be deleted. Since the output of the previous pipeline stage is forwarded to enable the pin of the next stage, no additional logic is required for this reduction.

By consistently applying this optimization, the size of logic area for the entire Snort rule set is decreased yet another 50 % the previous design. Another benefit of this optimization is reduction of VHDL source code. This is important to note especially because we found through experience that large source code required much larger memory and slowed down the compilation time by factor of 2 to 10.

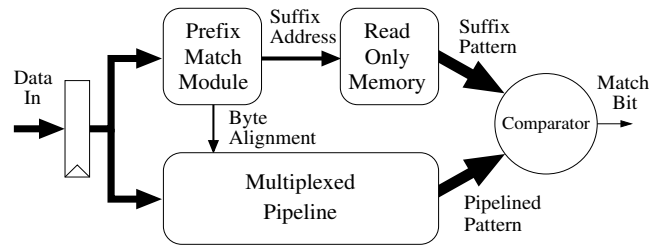


Fig. 12. Block diagram of a memory based pattern search engine

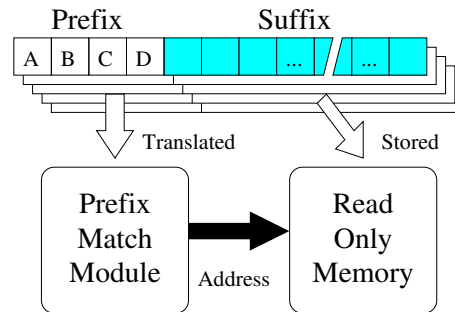


Fig. 13. Pattern prefix and suffix

3.2 Using Embedded Memories

In this section, we present an extension to the architecture that allows designers to make trade-offs in uses of LUTs and Embedded memories to build pattern matching engines. By combining previous architecture with this extension, one can maximize the resource utility of any given FPGA.

Figure 12 succinctly presents the extension. First, it uses the decoder based filter to “pre-screen” the first part of the pattern, which we call “prefix.” Once the prefix is identified, it is converted in to an index which loads the rest of the pattern called “suffix” to the pipeline comparison circuit. Then, rest of the input patterns are compared against the “suffix” at once.

The index generated by the prefix search engine points to an address into a ROM where the suffixes are stored as illustrated in figure 13. For multiple byte datapath, the alignment information of the matching prefix is used to realign the data for correct comparison.

3.2.1 Pattern Partition. In order for a given design to function correctly, one must partition the patterns and map them into the ROMs. The most important condition in partitioning the rule set is that all the prefixes in each partition must be unique. For different length prefixes, the tail end of the longer prefixes in the partition must not match the shorter prefixes. Otherwise, more than one prefix match can occur. If there are more than one prefix detection, the module can not determine which suffix to read from the ROM.

Since every clock can potentially produce a valid index, suffix comparison must be done at most once per cycle. Thus, any prefixes that can trigger two possible

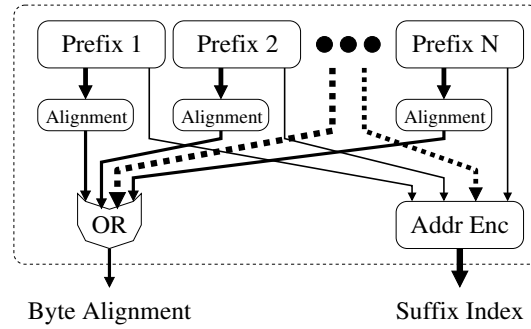


Fig. 14. Prefix match module

alignments must be assigned to different ROMs. For a single byte datapath, this constraint does not cause any problem, since the data is always at zero alignment. However, certain prefixes can match two different alignments in a multiple datapath design.

For instance, let's assume that in the 4-byte datapath, a prefix match module was configured with a prefix "ABAB". If the incoming data started with "ABAB," the alignment for the prefix could be either 0 or 2. Therefore, depending on the datapath and the lengths of the prefixes, conditions must be formed to test every pattern in the set to allow for only up to one index detection at each cycle.

Consider further that in a 4-byte datapath with fixed 4-byte prefixes, all prefixes in the partition must meet the following three criteria. (1) Byte 1 of the prefix can not be equal to byte 4. (2) The substring from byte 1 to 2 cannot equal substring byte 3 to 4. (3) Substring from byte 1 to 3 cannot equal substring byte 2 to 4. Other alignment constraints are possible, but 1 and 4 byte designs are the most likely to be built.

The role of the prefix match module is to match incoming data with a prefix of patterns configured in the datapath. For multiple byte datapath, it must also generate an alignment offset of the corresponding suffix.

The pattern search engine of the prefix match module is equivalent to the decoder based pattern search engine presented earlier. As shown in figure 14, the circuit for detecting all byte alignment is also generated. Our software synthesis tool guarantees that only one prefix in a given subset will be detected at each clock cycle, it does not have to consider the priority of matching patterns as with the decoder filter.

Once the suffix is read from the ROM, the subsequent data is pipelined and shifted to the lineup at the comparator as in figure 15. Based on the length of the longest pattern and ROM latencies, the number of pipeline stages are determined. The shifters are made with a single level of multiplexers or they are pipelined to multiple levels, depending on the width of the input bus. Since a single byte wide datapath has only one alignment, no shifters are necessary.

In addition to suffix data, the ROM must store the length of each pattern. The length is decoded at the comparator to enable the comparators of the indicated length. Then the memory output is compared with the byte aligned packet data.

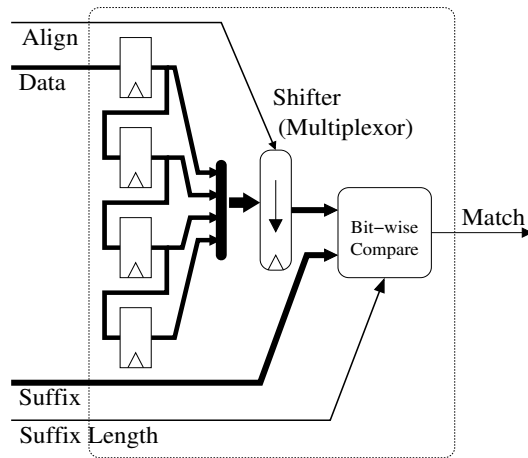


Fig. 15. Suffix comparator: the incoming data is shifted to the corresponding alignment. Then the suffix is compared against aligned pattern.

If the incoming data matches the suffix, the prefix index is forwarded as the output of the circuit.

We note here that unlike the decoders used in discrete gate design, the 1-byte XOR comparators are made of more than one LUTs. Depending on the synthesis tool and resource availability, the number of LUTs varied from five to eight. But, since only one set of comparators is used by all of the patterns stored in the ROM, the average gates per byte can be much less than the full decoder implementation.

3.2.2 Data Specific Memory Configuration. Since memory modules are not standardized among different FPGA manufacturers, describing a generic and efficient memory module in VHDL is difficult. Even if the generic VHDL modules are created, most vendor-specific compilers usually do not effectively map them as memory. Instead, memory modules are usually transformed into combinational logic that may waste a large amount of resources. In order to most effectively use the embedded memory, a target specific VHDL generator is necessary.

Most FPGA vendor tools have memory primitive templates that can be used to correctly configure the built-in memory. A primitive template is chosen based on the dimensions of the pattern set for the best utilization. For the pattern search application, the memory configuration with the widest data bus is the best because of the long length of the patterns. Once the template is chosen for a given pattern set, its suffixes are processed and written in to the multiple primitive modules. These modules are instantiated and connected within a top memory module to hide the distribution of the memory content over multiple modules.

After the patterns are partitioned into ROMs, each set may contain suffixes of varying lengths. When the set is stored in the ROM, the memory utilization tends to be low due to the fixed width of the memory; which is as wide as the longest suffix entry in the set. There are a few ways to modify the memory to improve its utilization. Since our goal is to minimize the logic resource, we present a simple modification to the memory that can greatly increase utilization.

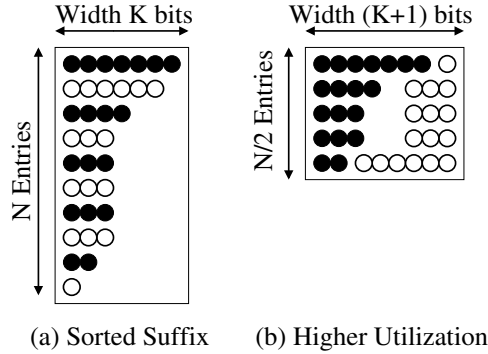


Fig. 16. Rearranging data to increase memory utilization

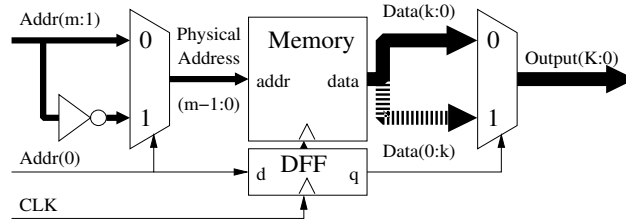


Fig. 17. Increasing the memory utilization

While experimenting with dividing the patterns, we found that the majority of sets were mid-size suffixes. Most of the sorted suffixes look similar to figure 16a. When these patterns are stored directly into the memory, nearly half of the memory is wasted. Our modification improves the utilization by filling in the empty spaces with the valid data.

We begin by sorting the patterns in the set by their length. Then all the even entries are sequentially stored from the first entry of the memory to the last. Then all the odd entries are flipped in terms of bit sequences and stored from the last entry to the first as shown in figure 16b. This process effectively stores the odd entries into a transposed memory.

In order to correctly read the rearranged memory entries, a wrapper logic is necessary. Figure 17 is a block diagram of the wrapper logic. At the address input of the memory, all the bits, except for the least significant bit (LSB), are passed to the actual memory. The LSB is used to determine whether the memory is even or odd. If the address is even, the rest of the address bits are unchanged and passed on as a physical address. Otherwise, the address bits are first inverted and then passed on to the memory. Likewise, the output of the memory is connected to a 2-to-1 multiplexer with the LSB connected to its select pin. When the LSB indicates even entry, the normal output is selected. If odd entry is called, the output with the reversed bit order would be selected. The address assignment of each suffix is a very important factor in high memory utilization. Therefore, the dimensions and the constraints of the memory should be considered during the pattern partitioning process.

3.2.3 Balancing Resource Usage. Since every pattern can be considered independent, patterns can be divided into two groups. One group can be built only using the LUTs while the other can be built with embedded memories and supporting logic. The mix of architectures can best utilize the resource since most FPGAs today have LUTs as well as embedded memories. Most applications do not allow such flexibility of using memory and LUTs interchangeably. However, for pattern matching, we can build a design to improve better FPGA utilization by using architectures presented in this section.

4. PATTERN MATCHER IMPLEMENTATION

In practice, the rule set is constantly updated to guard the network from the new attacks. Thus, the filter also needs to be either programmable or reconfigurable. In order to build high-speed hardware that is reconfigurable, we chose FPGAs as the design platform. The efficient pipeline structure and the simplicity of our design algorithm allow the FPGA to process at a fast clock rate.

4.1 Methodology

We automate the design process to produce and compile the code whenever the signature set is updated. We wrote structural VHDL templates of highly-pipelined modules based on the architecture describe in the previous sections. The signature set is used to generate logic and parameters for the template to generate a custom VHDL source code. The resulting structural VHDL files are efficient due to the predefined design templates. The design modules are instantiated using an optimized datapath template to build the system.

To simplify experiments and compilation processes, we wrote an interactive data manipulation tool with macro and script capabilities. This tool is used to pre-process multiple rule files and manipulate data for producing experimental designs. Several VHDL generators are interfaced with the main code. Then several scripts are written to generate different versions of VHDL for identifying the best data manipulation steps.

By trying different combinations of VHDL codes, we found that breaking up the single module into several smaller modules resulted in significantly faster total loading and compilation time. Since the design has a parallel structure, the modification was not difficult. However, this approach also ran into a memory problem when there were too many modules. Through additional manual trials, we determined the size for the modules that gave a reasonable load and compilation time. We also learned that restructuring modules in hierarchical fashion improved the speed of compilation. Thus, our VHDL generators were modified to pre-process all data to determine data dependencies and hierarchical structure.

Source code is produced with the net list of the design. The net list is processed and mapped on to a specified FPGA with the “place and route” tool. Although the “place and route” tool supports several optimization options, we do not use these options for our design for quicker compilation. Since the design is highly pipelined and parallel, there is much freedom for the “place and route” tool. Therefore, despite the minimal compilation options, the CAD tools produce the designs with high run-time performance.

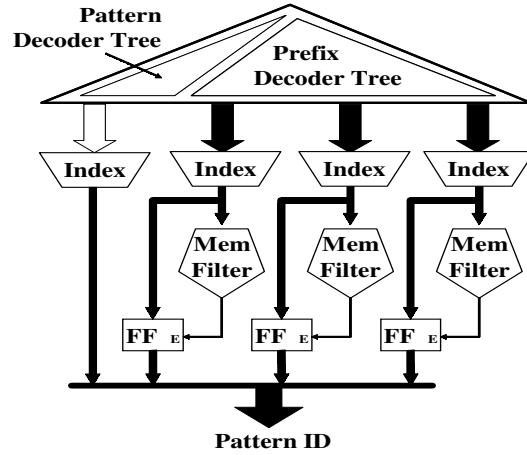


Fig. 18. Combining Decoder Tree with Memory based Pattern Search Module.

4.2 Design Platform

Other than the memory based filter, all source code is written using generic structural VHDL that would easily map to 4-input LUTs and 1-bit DFFs used in most FPGAs today. Therefore, our tools produce hardware structures that can run at high frequency in most FPGAs. Due to available resources and support, all our fast DPI implementations use Xilinx FPGAs. Since embedded memories defer drastically between different FPGA vendors, our memory based filter uses Xilinx 18 Kb block RAM template to generate efficient designs.

4.3 Area and Performance

The timing and area report generated by the compiler is used to determine the accurate logic area and system performance. For our purpose, we do not include static data inspection units. However, we believe that all logic improvements will impact the size of these smaller units.

Progression of our LUT only design, which we will refer to as decoder architecture, can be seen through our sample Snort NIDS data set from March 2004. Pre-processing the data to reuse substring comparators, our design mapped onto about 120,000 LUTs from an estimated 260,000 LUTs. This result showed promise because it was the first of its kind that mapped on to a single FPGA at the time of its implementation, namely XC2VP125. Once we enhanced the VHDL generator to implement 8-bit comparator reuse, we found further area reduction. According to the compilation report from the ISE mapper, total gate usage was under 22,000 gates in Xilinx FPGA. This result indicated that our design can be mapped onto a much smaller FPGA. Then, we extended our design with a priority address encoder. Our final design successfully mapped onto 27,702 Xilinx LUTs in Spartan 3 XC3S2000 FPGA with a maximum throughput of 8.0 Gbps. By combining many LUTs, we effectively reduced the amount of connection as well as the average wire length.

Initially, we implemented the filters using only the decoder architecture with

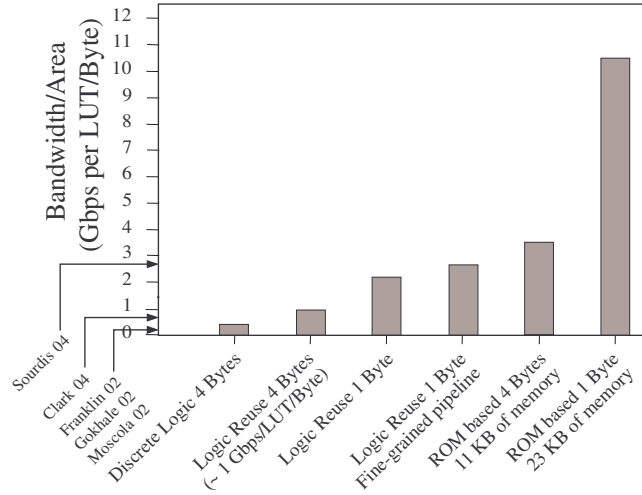


Fig. 19. Area and Performance Improvement with subsequent data-specific and FPGA resource specific optimizations.

input data width of 1-byte, 4-bytes, and 8-bytes. Since this design only uses discrete gates, the number of gates is larger than the memory based design. Accordingly, none of the built-in memories in FPGA were used. Also, none of the gates in this design had to consider a large fixed components such as block RAM. Since the design has fewer FPGA CAD placement constraints, it is able to run at a faster clock rate. Depending on the application and the available FPGA devices, this design choice may be preferable.

As we mentioned earlier, the patterns assigned to the same ROM must have unique prefixes. Given the fixed number of available memory modules, some patterns may not partition into any one because of the above constraint. The only way to configure such patterns is by converting them into decoder filters. Due to larger resource requirements for the generic XOR based comparators, a smaller set of patterns often is more efficiently implemented as a hardware decoder.

With these modifications, we generate the same pattern search design that took advantage of the best of both architectures. The block diagram of this compact design is shown in figure 18. Since the prefixes for patterns stored in the memory are essentially a set of shorter patterns, they are combined with the rest of the patterns to yield an efficient keyword tree.

Since large amounts of patterns are stored in the memory, the overall design requires much less LUTs than the decoder based filter. As a result, the full filter successfully placed and routed into a smaller Spartan 3 - XC3S400 device. The system uses a total of 4,415 LUTs with a clock rate of 200 MHz in a XC3S400 device and runs at 271 MHz in Virtex-4 LX15 device. The speed up in the Virtex 4 device is due to a higher degree of freedom in placing the components in the larger FPGA.

In the instances where the pattern detection rate is relatively low, it may be

ACM Journal Name, Vol. V, No. N, Month 20YY.

Method	Scale Factor	FPGA	Speed Grade	Bytes	Clock (MHz)	BW (Gbps)	LUT per Byte	Mem (Kbit)
<i>Decoder</i>	1	V4LX200	-11	300	533	4.26	1.01	0
<i>Decoder</i>	1	V4LX200	-11	600	497	3.97	0.88	0
<i>Decoder</i>	1	S3-1500	-4	20800	250	2.00	0.81	0
<i>Decoder</i>	4	S3-2000	-4	19021	250	8.00	1.40	0
<i>Decoder</i>	8	S3-5000	-4	19021	250	16.00	2.52	0
<i>Memory</i>	1	S3-400	-4	32384	200	1.60	0.21	184
<i>Memory</i>	1	V4LX15	-11	32384	271	2.17	0.21	184
<i>Memory</i>	4	S3-2000	-4	6805	250	8.00	0.90	90
Related Work								
<i>Singaraju</i>	2	V2-30	-7	16347	N/A	2.36	0.70	0
<i>Yusuf</i>	1	V2-8000	N/A	19715	N/A	2.50	0.60	0
<i>Baker</i>	1	V2P100	N/A	8263	224	1.79	0.35	0
<i>Clark</i>	1	V2-8000	N/A	17537	233	1.86	1.70	0
<i>Sourdis</i>	1	V2-3000	N/A	18031	335	2.68	0.97	0
<i>Franklin</i>	1	VE-2000	N/A	8003	50	0.40	2.58	0
<i>Gokhale</i>	1	VE-1000	N/A	640	272	2.18	15.19	24

Table I. Comparing area and performance results of FPGA based deep packet filters.

sufficient to indicate a match signal without identifying the pattern. Software can do a thorough search of the database to determine which pattern was detected. For this design, all the index encoder for the decoder based matcher can be replaced with a tree of OR gates to reduce the amount of gates. Due to our efficient index encoder design, only 755 LUTs can be reclaimed from the first design and 65 LUTs for the memory based implementation.

Summary of our implementation area and performance results are shown on figure 19 and table I. We built our parallel deep packet filter for the Snort rules from January 2004 with 1,625 unique strings. By applying several hardware logic optimizations, what was once over 230,000 LUTs now fits on to a single Spartan 3 FPGA using at most 0.21 LUTs per byte of pattern. Furthermore, we successfully placed and routed different filters that sustain a data throughput of 1.6 Gbps to 16.0 Gbps to show that the architecture is scalable given more logic resources.

5. CONCLUSION

The benchmark details of software and some hardware deep packet inspection applications reveal performance shortcomings. The poor performance of many of the current NIDS is mainly due to the sequential execution of a large set of pattern matching tasks. Therefore, existing software or hardware may not be fast enough to filter the networks with bandwidth in the order of gigabits per second.

Today, Gigabit Ethernet networks are becoming less expensive thus much more prevalent in local area networks. In order to protect such 1+ Gbps network from malicious attacks, we designed a high-performance scalable pattern matching architecture for FPGAs. The smallest of the designs that we have implemented fits on an FPGA that cost less than 6 US dollar while sustaining raw throughput of 1.6 Gbps.

Furthermore, we have shown through several implementations of pattern matchers that the architecture is linearly scalable in performance given hardware resources.

Although the purpose of our architecture is to enable high-performance NIDS, the pattern matcher can be used to accelerate other applications especially in the field of networking and biosequencing. Given a generic reconfigurable platform such as FPX of Washington University [Lockwood 2001] or Myricom FPGA node [Sivilotti et al. 1998], we can explore different applications of high-performance pattern matcher with the same hardware.

REFERENCES

- AHO, A. V. AND CORASICK, M. J. 1975. Efficient String Matching: An Aid to Bibliographic Search. In *Communications of the ACM*. ACM Press, 333–340.
- ALLEN, J., CHRISTIE, A., FITHEN, W., MCHUGH, J., PICKEL, J., AND STONER, E. 2000. State of the practice of intrusion detection technologies. Tech. rep., Carnegie Mellon Software Engineering Institute. Jan.
- ALLEN, J., CHRISTIE, A., FITHEN, W., MCHUGH, J., PICKEL, J., AND STONER, E. 2001. Code Red Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. Tech. rep., Carnegie Mellon Software Engineering Institute. Aug.
- BAKER, Z. K. AND PRASANNA, V. K. 2004. A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA.
- BLOOM, B. H. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. In *Communications of the ACM*. ACM.
- BRAUN, F., LOCKWOOD, J., AND WALDVOGEL, M. 2001. Reconfigurable router modules using network protocol wrappers. In *11th Conference on Field Programmable Logic and Applications*. Springer-Verlag, Belfast, Northern Ireland, 254–263.
- BRAUN, F., LOCKWOOD, J., AND WALDVOGEL, M. 2002. Protocol wrappers for layered network packet processing in reconfigurable hardware. *IEEE Micro* 22, 1 (Jan.), 66–74.
- Broadcom Inc. 2001. *Strada Switch II BCM5616 - Integrated Multi-layer Switch*. Broadcom Inc.
- CHO, Y. H. 2000. Optimized Automatic-Target-Recognition Algorithm on Scalable Myrinet/Field Programmable Array Nodes. In *Proc. Thirty Fourth Asilomar Conference on Signals, Systems, and Computers*. IEEE, Pacific Grove, CA.
- CHO, Y. H. AND MANGIONE-SMITH, W. H. 2004. Deep Packet Filter with Dedicated Logic and Read Only Memories. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA.
- CHO, Y. H. AND MANGIONE-SMITH, W. H. 2005. A Pattern Matching Co-processor for Network Security. In *IEEE/ACM 42nd Design Automation Conference*. IEEE/ACM, Anaheim, CA.
- CHO, Y. H., NAVAB, S., AND MANGIONE-SMITH, W. H. 2002. Deep Network Packet Filter Design for Reconfigurable Devices. In *12th Conference on Field Programmable Logic and Applications*. Springer-Verlag, Montpellier, France, 452–461.
- CLARK, C. R. AND SCHIMMEL, D. E. 2003. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *International Conference on Field Programmable Logic and Applications (FPL)*. Lisbon, Portugal, 956–959.
- CLARK, C. R. AND SCHIMMEL, D. E. 2004. Scalable Parallel Pattern-Matching on High-Speed Networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA.
- DESI, N. 2002. Increasing Performance in High Speed NIDS: A look at Snort's Internals.
- DHARMAPURIKAR, S., KRISHNAMURTHY, P., SPROULL, T., AND LOCKWOOD, J. 2003. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*. IEEE Computer Society Press, Stanford, CA.

- DOLLAS, A., PNEVMATIKATOS, D., AND ASLANIDES, N. 2001. Rapid prototyping of a reusable 4x4 active ATM switch core with the PCI pamette. In *12th IEEE International Workshop on Rapid Prototyping*. IEEE, Monterey, CA, 17–23.
- DOWD, P. W., MCHENRY, J. T., PELLEGRINO, F. A., CARROZZI, T. M., AND COCKS, W. 1997. An FPGA-Based Coprocessor for ATM Firewalls. In *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*. IEEE, Napa Valley, CA.
- DUBRAWSKY, I. 2003. Firewall Evolution - Deep Packet Inspection. *Infocus*.
- FALLSIDE, H. AND SMITH, M. J. 2000. Internet connected FPL. In *10th Conference on Field Programmable Logic and Applications*. Springer-Verlag, Villach, Austria, 48–57.
- FRANKLIN, R., CARVER, D., AND HUTCHINGS, B. L. 2002. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*. IEEE, Napa Valley, CA.
- GAUDIN, S. 2003. Virus Damage Worst on Record for August 2003. *Cyber Atlas*.
- GOKHALE, M., DUBOIS, D., DUBOIS, A., BOORMAN, M., POOLE, S., AND HOGSETT, V. 2002. Granidt: Towards Gigabit Rate Network Intrusion Detection Technology. In *12th Conference on Field Programmable Logic and Applications*. Springer-Verlag, Montpellier, France, 404–413.
- ILIOPOULOS, M. AND ANTONAKOPOULOS, T. 2000. Reconfigurable network processors based on field programmable system level integrated circuits. In *10th Conference on Field Programmable Logic and Applications*. Springer-Verlag, Villach, Austria, 39–47.
- KARAGIANNIS, M. 2001. *How to create a Stealth Packet Scrubber using Hogwash*. Hogwash.
- LOCKWOOD, J., MOSCOLA, J., KULIG, M., REDDICK, D., AND BROOKS, T. 2003. Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware. In *Military and Aerospace Programmable Logic Device (MAPLD)*. NASA Office of Logic Design, Washington DC.
- LOCKWOOD, J. W. 2001. Evolvable Internet hardware platforms. In *Proceedings of the 3rd NASA/DoD Workshop on Evolvable Hardware*. Department of Defense, Long Beach, CA, 271–297.
- MCHUGH, J., CHRISTIE, A., AND ALLEN, J. 2000. Defending Yourself: The Role of Intrusion Detection Systems. *IEEE Software Magazine* 17, 5 (Sept.), 42–51.
- MOSCOLA, J., LOCKWOOD, J., LOUI, R., AND PACHOS, M. 2003. Implementation of a Content-Scanning Module for an Internet Firewall. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA.
- PAPADOPOULOS, G. AND PNEVMATIKATOS, D. 2005. Hashing + Memory = Low Cost, Exact Pattern Matching. In *International Conference on Field Programmable Logic and Applications (FPL)*. Tampere, Finland, 39–44.
- PMC Sierra Inc. 2001. *PM2329 ClassiPi Network Classification Processor Datasheet*. PMC Sierra Inc.
- ROESCH, M. 1999. Snort - Lightweight Intrusion Detection for Networks. In *USENIX LISA 1999 conference*. USENIX, <http://www.snort.org/>.
- SIDHU, R. AND PRASANNA, V. K. 2001. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA.
- SINGARAJU, J., BU, L., AND CHANDY, J. A. 2005. A Signature Match Processor Architecture for Network Intrusion Detection. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA.
- SINNAPPAN, R. AND HAZELHURST, S. 2001. A Reconfigurable Approach to Packet Filtering. In *11th International Conference on Field Programmable Logic and Applications*. Springer-Verlag, Belfast, Northern Ireland.
- SIVILOTTI, R., CHO, Y., SU, W., COHEN, D., AND BRAY, B. 1998. Scalable Network Based FPGA Accelerators for an Automatic Target Recognition Application. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA.
- SOURDIS, I. AND PNEVMATIKATOS, D. 2003. Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. In *13th Conference on Field Programmable Logic and Applications*. Springer-Verlag, Lisbon, Portugal.

- SOURDIS, I. AND PNEVMATIKATOS, D. 2004. Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Napa Valley, CA.
- SOURDIS, I., PNEVMATIKATOS, D., WONG, S., AND VASSILIADIS, S. 2005. A Reconfigurable Perfect-Hashing Scheme for Packet Inspection. In *International Conference on Field Programmable Logic and Applications (FPL)*. Tampere, Finland, 644–647.
- Viacom Inc. 2001. *Firewall Questions and Answers*. Viacom Inc.
- YU, F., KATZ, R., AND LAKSHMAN, T. 2004. Gigabit Rate Packet Pattern-Matching Using TCAM. In *12th IEEE International Conference on Network Protocols*. IEEE, Berlin, Germany.
- YUSUF, S. AND LUK, W. 2005. Reconfigurable network processors based on field programmable system level integrated circuits. In *10th Conference on Field Programmable Logic and Applications*. Springer-Verlag, Tampere, Finland.