Deep Packet Filter with Dedicated Logic and Read Only Memories

Young H. Cho and William H. Mangione-Smith
{young,billms}@ee.ucla.edu
University of California, Los Angeles
Department of Electrical Engineering
Los Angeles, California 90095

Abstract

Searching for multiple string patterns in a stream of data is a computationally expensive task. The speed of the search pattern module determines the overall performance of deep packet inspection firewalls, intrusion detection systems (IDS), and intrusion prevention systems (IPS). For example, one open source IDS configured for 845 patterns, can sustain a throughput of only 50 Mbps running on a dual 1-GHz Pentium III system. Using such systems would not be practical for filtering high speed networks with over 1 Gbps traffic. Some of these systems are implemented with field programmable gate arrays (FPGA) so that they are fast and programmable. However, such FPGA filters tend to be too large to be mapped on to a single FPGA. By sharing the common sub-logic in the design, we can effectively shrink the footprint of the filter. Then, for a large subset of the patterns, the logic area can be further reduced by using a memory based architecture. These design methods allow our filter for 2064 attack patterns to map onto a single Xilinx Spartan 3 - XC3S2000 FPGA with a filtering rate of over 3 Gbps of network traffic.

1 Introduction

The recent emergence of application-level network attacks clearly indicate that inspecting the network packet header alone is insufficient to protect computers from network intrusion. In the near future, it may become necessary that all firewalls employ some sort of deep packet filters to detect application level attacks in the packet payloads.

Deep packet filters are designed to not only examine headers but also the payloads of packets. Therefore, a security system that incorporates a deep packet filter offers better protection from attacks than traditional firewalls. For example, traditional firewall have not been effective in differentiating network packets containing denial of service (DoS) attacks such as "Code Red" worm from normal packets. However, a deep packet inspection system can be configured to detect "Code Red" worm by searching for a string pattern ".ida?" in the payload [2].

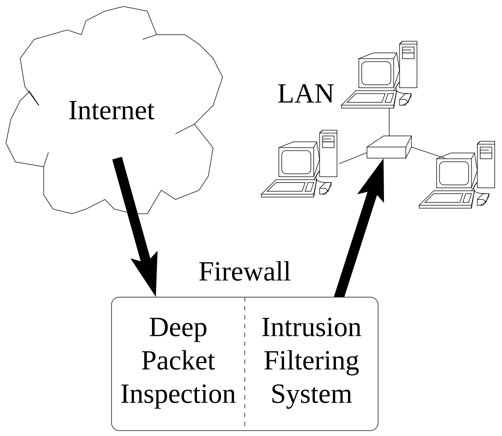


Figure 1. Firewall with Deep Packet Inspection

Packet header inspection can be done efficiently with relatively low cost since the locations of packet header fields are defined by protocol standards. However, the payload contents are not constrained by any standard definitions. Therefore, payload search and analysis tends to be more complex and expensive.

Today, companies such as Netscreen, SourceFire, and Cisco produce firewalls that employ some form of deep packet filtering [5]. Most of these systems use one or more general purpose processors running signature-based packet filtering software. It is difficult, if at all possible, for a single software deep packet inspection system for 500 realistic patterns to sustain a bandwidth of 100 Mbps or more. Few co-processors such as PMC-Sierra ClassiPi and Broadcom Strata II are reported to support such speeds. According to their specifications, the bandwidth on the order of giga-bits-

*This research work was supported by NSF Grant #CCR-0220100.

per-second can not be achived for even the most common attack pattern set with one of these co-processors.

1.1 Deep Packet Filters

In deep packet filtering, the patterns must be compared at every byte alignment of a payload during the string search process. Thus, the throughput of a search algorithm running on a sequential processor decreases as the number of byte comparison increases. One can overcome such performance limitation by translating the algorithm into an efficient parallel processing design.

Since the signature set for intrusion is constantly updated, re-programmability is essential for a deep packet filter system. Considering the criteria for the application, FPGAs are an excellent design platform because of their ability to quickly map and re-map parallel hardware designs onto the same device.

Our contribution in this paper are two effective design techniques for building a compact deep packet filter in reconfigurable devices. The first method is used to compress the hardware size (compared to previous designs) by reusing the subcomponents of reconfigurable discrete logic (RDL) filter [13]. The second method utilizes built-in memory to further reduce the footprint of the filter logic.

In order to place our work in perspective, section 2 briefly presents other reconfigurable string search implementations. In section 3, we describe our original deep packet filter and its results. Then we present design technology that reduce the design footprint by effectively eliminating duplicate logics. In section 4, we propose a ROM based architecture which further reduces the area especially well with larger rule set. Finally, we conclude with a summary of our design and comparison of our work with other similar projects.

2 Related Work

Currently, there are a few fast pattern search algorithms implemented in FPGA to meet the performance requirements of high-bandwidth network filters. Sidhu and Prasanna mapped Non-deterministic Finite Automata (NFA) for regular expression into FPGA to perform fast pattern matching [11]. Using this method, Franklin et. al compiled patterns for an open-source NIDS system into JHDL [6].

Then Washington University translated regular expressions into deterministic finite automata (DFA) showing that, in practice, most DFA optimizes to compact and fast hardware [9]. Due to the parallel nature of the hardware, these designs maintained high performance regardless of the size of the patterns.

The Granidt project of Los Alamos National Laboratory implemented a fast re-programmable deep packet filter using content addressable memories (CAM) and the Snort rule set [7]. Our implementation of the same application uses RDL to build fast pattern match engines. Our match engine is a parallel pipelined series of reconfigurable lookup tables (LUT) that can sustain a bandwidth of 2.8 Gbps [2]. Sourdis mapped a similar design with a deeper pipeline to increase the filtering rate [12] while Clark made some area improvements [3].

All of the above efforts have experimented only with a small subset of the Snort rules containing fewer than one hundred different patterns. If all of the current Snort rule set was compiled using any of the above designs, the resulting hardware would be too large to fit in to a single FPGA. In the light of this, a research group from Washington University fit a pattern detector with most of the Snort rules into a single FPGA (Xilinx XCV2000E) using index lookup technique. With the use of Bloom filters [1], they detect the patterns at 600 Mbps with some false positives. However, due to the nature of the algorithm, identifying the detected pattern or false positive would eventually require a string comparison process [4, 8].

3 Reconfigurable Discrete Logic Filter

Our system is composed of several inspection units that simultaneously compare incoming data with all the signatures defined in the signature set [2]. Due to the regular structure of the design, the logic area can be reduced by applying a number of optimization techniques. By reducing the logic area we can map a large number of signature onto a single FPGA.

As with the other projects described in the previous section, our implementation uses the high-level software rule signature from Snort to build a reconfigurable deep packet filter. A rule contains information to search through all layers of network packets to detect a particular attack. When a packet contains a targeted header, an exhaustive pattern search is performed on its payload to confirm a detection of an attack [10].

We use the Snort rules to automatically generate deep packet filtering modules in VHDL. This intermediate form of the hardware description can be compiled and mapped on to FPGAs by most vendor tools.

3.1 Brute Force Method

The core processing unit of our deep packet filter is shown in figure 2. Since our design is made with reconfigurable logic gates, byte comparators are simply an 8 to 1 bit decoder. From this point on we will refer to such pipelined string of comparators as 1-byte inspection module.

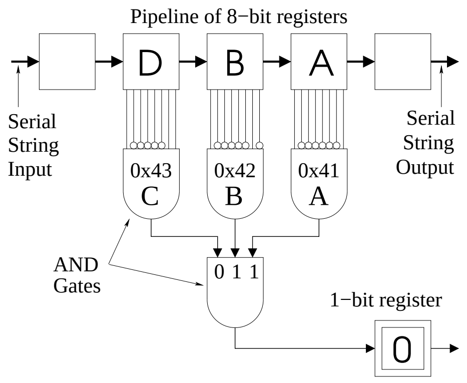


Figure 2. Simple String Search Engine for "ABC": Since the input substring is "ABD", the comparator detects no match.

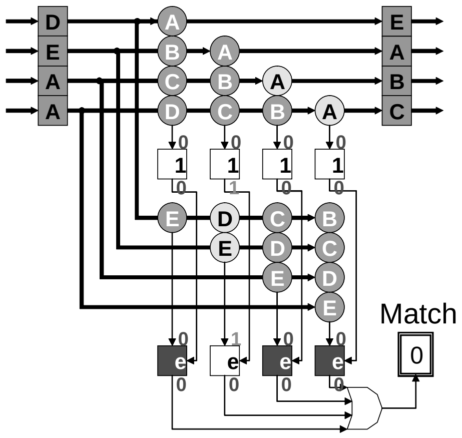


Figure 3. Parallel String Search Engine for "ABCDE": "ABC" was matched on the previous cycle, enabling the registers to allow signal from "DE" comparator to latch; on following clock cycle, Match signal would indicate a match.

The processing unit is easily scalable by simply widening the bus and adding duplicate inspection modules for each byte alignment. For our initial research [2], we generated inspection modules for a four byte wide bus as shown in figure 3. In place of 1-byte inspection modules, substring inspection modules are used to examine every alignment of incoming data.

For each signature, we generate a series of 4-byte inspection modules. These modules are connected in parallel, as in figure 4, to compare incoming substrings with all the patterns at every cycle. Therefore, performance of the filter is only dependent on the bus width and clock rate.

In our initial experiment, we used 105 of the most common attacks described in the Snort rule set to construct a deep packet filter. This early system was mapped, placed, and routed on to an 20,000 gate Altera EP20K FPGA with a critical path of 11.685 ns. At 90 MHz with 32-bit wide bus, our system can filter 2.88 Gbps of data regardless of the size of the patterns or the packet length [2].

3.2 Logic Reuse Techniques

As mentioned in section 2, the main challenge facing most reconfigurable deep packet filters are the large logic resource requirements. There are 2,064 signatures in the Snort rule set as of November 2003. Since there are a total of 21,830 bytes in the signature set, we can estimate that implementing the whole set would require over 230,000 logic gates. Mapping such design using 20,000 gate FPGAs would require 12 FPGAs. On the other hand, it may be possible to map the system on two largest Xilinx Virtex II Pro (XC-2VP125) FPGAs.

Instead of attempting to map such large design on to several FPGAs, we worked to optimize our design shrink its design footprint. During the analysis, we found that some of the rules look for the same string patterns but with different headers. By combining all the signatures with the same payload content, we can eliminate duplicate patterns. Through simple pre-processing, we reduced the number of total rules from 2,064 down to 1,519 rules containing unique patterns.

In order to verify our resource usage, we attempted to generate our original design for the entire Snort rule set. 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 with Quartus II, but ended with the same result. We could not exhaustively test for the limitations of CAD tools because of the lengthy compilation time; instead we began to make improvements to the logic thus reducing the source code and its complexity.

3.2.1 Shared Substring Comparator

Filtering a stream of data through a multiple byte bus requires inspection of the same data at different byte alignments of the bus. Therefore, as shown in figure 3, our 4-byte wide 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 through 4 byte substring comparators in each stage of the pipelined registers.

Since all 4-byte substring inspection modules are connected in parallel, there are multiple comparators that check for the same substring. Since all the input pins of the com-

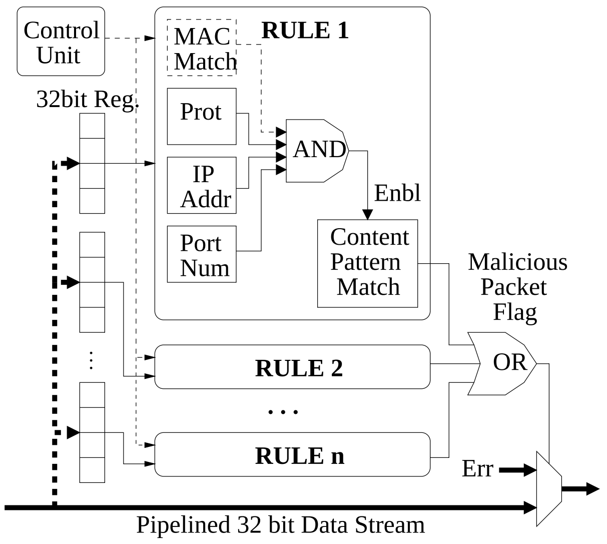


Figure 4. Parallel Deep Packet Filter

parators are connected to the same bus, some comparators are, in fact, exact duplicates of each other.

The first step of eliminating duplicate comparators is to break 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 unique string segments are then translated into comparators eliminating duplicate components. Finally, the outputs of the comparators are forwarded to modules that use them.

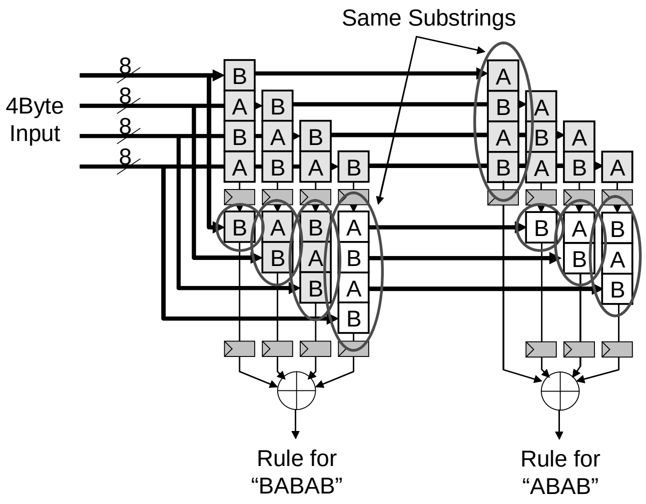


Figure 5. Reusing Common Substrings: All the matching substrings with same alignment are combined.

As an example, figure 5 shows that there are four pairs of duplicate comparators for 4-byte substring inspection modules for "BABAB" and "ABAB". From inspection, we find that extra comparators for substrings "ABAB", "BAB", "AB", and "B" can be eliminated. Since the output of the previous pipeline stage is forwarded to enable the next stage, no additional logic is required for this area improvement.

By applying this optimization over the entire system, the total logic area shrank to half of its initial size. This process also reduced the size of the hardware description source code which effects the compilation time.

3.2.2 Shared Byte Comparators

Within the multiple byte comparators, we found another opportunity to reuse the substructures of the system. Each byte comparators are constructed using a tree of 8 to 1 bit decoders. By applying the same technique we used in section 3.2.1, we can eliminate duplicate 1-byte comparators in the substring modules.

First, we had to determine the optimum size for the 1-byte comparator. We deduced that reducing the size of the subcomponent will increase its use in comparators. However, the side effects of the smaller subcomponent is increased number of wires to route. Therefore, smaller subcomponents will reduce the size of the basic modules but larger number of connections will force the compiler to duplicate these modules to distribute the fanout.

Since the Xilinx LUT has 4-inputs, we initially produced each comparator as 4-bit subcomponents. At most, only sixteen different comparators were mapped for each 4-bit input. As expected, each subcomponent was reused many times. In fact, in order to accommodate the output load, the Xilinx compiler automatically generated duplicate logic. The result did not give a significant area improvement from the results of reusing substring comparators.

In order to reduce the amount of wires, we increase the size of the subcomponent to 8-bit. Due to the internal structure of Xilinx, for some 8-bit gates, the mapper can use two 4-bit LUTs eliminating a need for an extra LUT to connect the two. Consequently, the design has reduced to one third of its original size.

To illustrate the effect of this technique, figure 6 shows that the output of the "B" comparator in the first pipeline stage can be reused for two other comparators in the second pipeline stage. By eliminating all duplicates, the example design in figure 6 only needs a total of eight 8-bit comparators. Thus, the total logic requirement for the example is reduced to 25 percent of the original design.

For systems with a large set of signatures, each alignment requires 8-bit comparators of all combinations. Effectively, 8-bit to 256-bit address decoder are placed at each alignment. Each of the decoded bits are then used as inputs for unique substrings described in the previous section.

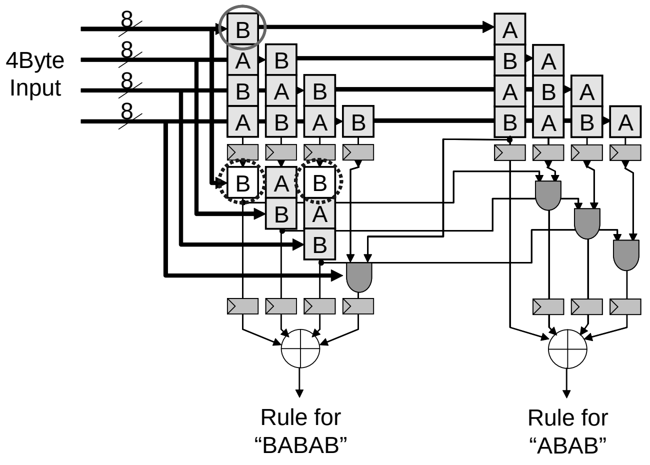


Figure 6. Reusing Common 8-bit Comparators: All the common 8-bit comparators in the same alignment are combined.

3.3 Signature Index

The inspection module for each pattern produces a 1-bit output to indicate whether its target string is detected at that clock cycle. Our initial design ORed all of these bits together. Such indications maybe sufficient in some applications, but it is much more desirable to produce a corresponding signature index number for further analysis. Therefore we have designed an address encoder for flag bits.

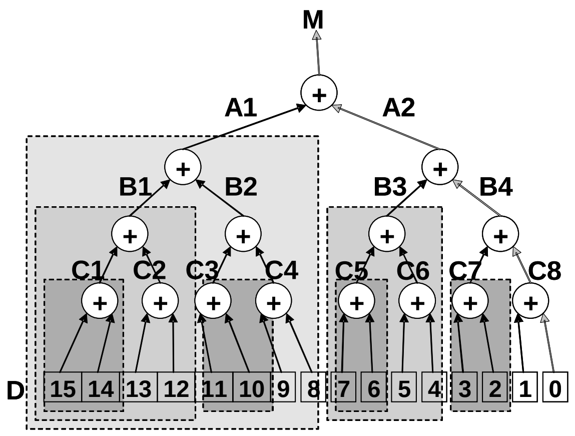
3.3.1 Prioritized Address Encoder

Since there are 1519 unique inspection modules for the most current Snort rule set, the address encoder must produce an 11-bit output. Producing a 1519-bit to 11-bit output is not a trivial task because there are so many input bits. Not only that, more than one inspection module may match at the same time. Therefore, our address encoder must have a priority mechanism.

For simplicity, we wrote our address encoder in VHDL using a long chain of CASE statements. When we attempted to compile our encoder for the entire rule set, the size of the mapped logic exceeded reasonable amount of gates. Due to an iterative chain of combinational logic, the latency for each bit of the address became too long to be effective. Also, the complexity introduced by the compiler made pipelining difficult. Instead of optimizing such poor result, we present an efficient hardware design of a priority address encoder.

3.3.2 Pruned Priority Binary Tree

Our new pattern identification mechanism can be represented as an OR binary tree as shown in figure 7. We found that we can use branches of our binary tree to efficiently generate priority based addresses.



$$
\begin{aligned}
Bit3 &= A1\\
Bit2 &= B1 + B3 \bullet \overline{A1}\\
Bit1 &= C1 + C3 \bullet \overline{B1} + C5 \bullet \overline{A1} + C7 \bullet \overline{A1} \bullet \overline{B3}\\
Bit0 &= D15 + D13 \bullet \overline{C1} + D11 \bullet \overline{B1} + D9 \bullet \overline{B1} \bullet \overline{C3} + D7 \bullet \overline{A1}\\
&\quad + D5 \bullet \overline{A1} \bullet \overline{C5} + D3 \bullet \overline{A1} \bullet \overline{B3} + D1 \bullet \overline{A1} \bullet \overline{B3} \bullet \overline{C7}
\end{aligned}
$$

Figure 7. Pruned Priority Address Encoder: M, A2, B4, C8, and D0 are not needed to encode address bits, thus pruned off. Bit3 through 1 are logic equations for prioritized address.

We know that the root of the binary tree has two branches. For the sake of description, we can label these child nodes "A1" and "A2", as shown in figure 7. The definition of binary tree leads us to calculate that the most significant bit of encoded address must be "1" if the tree under A1 contains a flag that is "1".

Since higher flags have the priority over lower flags, the output of A2 need not be considered. Therefore, the most significant bit value is assigned as the output of A1. For the next address bit, we consider branches with nodes that are immediate children of A1 and A2; we label root nodes of these branches "B1" through "B4". Once again, we can deduce that the second address bit is "1" if the output of B1 is "1". But we also find that the second address bit is "1" if the output of B3 is "1" while no higher flags are "1"; in the binary OR tree we only need to check that A1 is "0" to verify that higher flags are not "1". We extract subsequent address bits using same procedure.

To save area, unused nodes of the OR tree are deleted. Starting with the main root node, all nodes on the lower edge of tree can be pruned off. With registers at the output encoded address bits, the critical path has maximum of (log n)-1 gate delays where n is the number of the input bits.

3.4 RDL Filter Results

More efficient deep packet filters can be produced after applying the above improvements. To aid in the design process, a tool with macro and script capability was developed. 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. Many scripts were 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 allowed significantly faster compilation. Since our design has a highly parallel structure, such modification was not difficult. However, this approach also ran into a memory problem when the code was broken into too many modules. Thus, by trying out many more models, we found an approximate size for each module that gave an effective load and compile 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 valid hierarchical structure.

The restructured datapath with logic improvements reduced the size of the source code down to one third of the initial size. As a result, the entire Snort rule set can be completely compiled on to an FPGA in about 3 hours. Successive iteration times are shortened when previously compiled files are used as guide files for the new version.

3.4.1 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 unit.

After pre-processing the data to reuse substring comparators, our design mapped onto about 120,000 logic gates. This result showed promise because it was first of its kind that mapped on to a single FPGA, 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, the 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.

We extended our design with a priority address encoder. Our final design successfully mapped onto 26,607 Xilinx logic gates which placed and routed onto 15,202 Slices of Spartan 3 XC3S2000 FPGA.

By combining many logic gates, we effectively reduced the amount of connection as well as the average wire length. However, since this new system is much larger than our initial design, performance only improved slightly with maximum clock rate of 100 MHz. Therefore, with a 32-bit datapath, our system is capable of filtering at a line speed of 3.2 Gbps.

4 Read-Only-Memory Based Filter

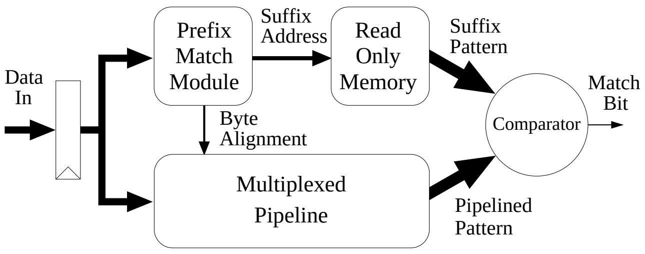


Figure 8. Block diagram of Read-Only-Memory based inspection unit

The memory based filter (Figure 8) is a pipelined datapath with a prefix match module, ROM, multiplexor, and XNOR based comparator. First, the prefix module matches the initial part of the incoming data, which we call prefix, with a set of patterns it is configured to detect. At the same time, the starting alignment of the matching sub-pattern which we call "prefix" is also determined. Identity information is then directly mapped as an address into a ROM where the rest of the pattern, called the suffix, is stored as shown in figure 9. The incoming data and suffix are compared at a corresponding alignment to determine whether an exact match has occurred.

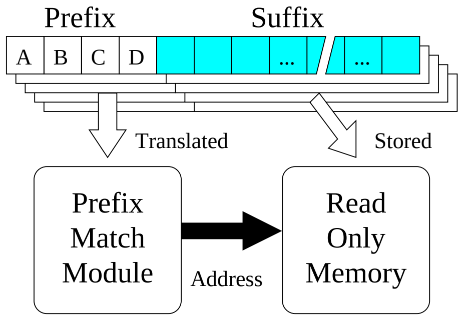


Figure 9. Pattern Prefix and Suffix

Although this datapath requires more logic than a single RDL based pattern matching module, the amount of logic area per pattern can be less when large number of suffixes are stored in the memory.

4.1 Pattern Selection

In order for the given design to function correctly without additional logic, one must partition the patterns into sets. Each set is then executed on dedicated hardware. An obvious rule to consider is that all the prefixes of the pattern set for a single datapath must be unique within the set. Otherwise, when incoming data contains prefix of two or more patterns at the same time, the module would not be able to determine which suffix to read from the ROM. No other rules are needed for datapath with one byte input. In fact, even the alignment information would be unnecessary since all would have the zero alignment.

Although the prefix width can be of any size, for simplicity, our implementation designates the length of the prefix to be equal to the width of the input bus. In order for the design to work with a wider bus, a few additional rules need to be applied.

Since we intend to perform exact prefix comparisons, patterns that are shorter than the width of the prefix would not benefit from this design. Therefore, we select only the patterns that are longer than the prefix length.

The ROM can only output a suffix of a single pattern at each rising edge of the clock. Thus, any pattern prefix that can cause detection of more than one alignment must be sorted out. The only pattern prefixes that can trigger such an event are the ones with matching beginning and ending sequence. For instance, if the prefix match module was configured with a prefix “ABAB”, the alignment for the incoming data “ABAB” could be either 0 or 2. This is because the second sub-string “AB” could be the actual starting point of the pattern. Therefore, valid four byte prefixes must meet the following three conditions. (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.

Finally, no more than one prefix in the same pattern set should trigger a detection. Although all prefixes are unique, certain incoming data can trigger two different byte alignment detection of two different patterns. To avoid such detection, the three conditions described in the previous paragraph must be applied for every two prefixes within the set.

4.2 Prefix Match Module

The role of the prefix match module is to identify a given incoming data fragment with a prefix of patterns configured in the datapath. It also needs to determine the byte alignment of a possible starting point. One may use some sort of hash function to use small amount of logic resource. However, for simplicity, we modified the RDL design generator to build VHDL modules for exact pattern matcher for all the prefixes.

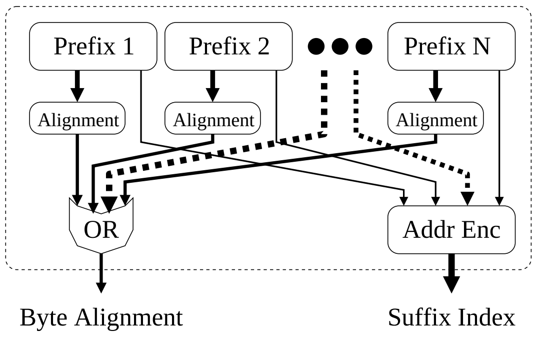


Figure 10. Structure of prefix match module

Each prefix is compared separately at each alignment equivalent to how patterns were mapped in figure 3. Therefore, the match signals from all alignment chains can be encoded to provide alignment information. Since preprocessing guarantees that only one prefix in a given subset will be detected at each clock cyle, it does not have to consider the priority of matching patterns as with the RDL filter. As shown in figure 10, the logic that handles alignment and the address encoding is much simpler and compact.

4.3 FPGA ROM Module

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 module are created, most vendor specific compilers usually do not know to map them on to the memory. Instead, memory modules are translated into some form of combinational logic that use unnecessarily large amount of resources. In order to most effectively use the embedded memory, a target specific VHDL generator is necessary.

FPGA memory modules can be configured to have different data bus width and entries. 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. In general, 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.

4.3.1 Improving Memory Utilization

When pattern sets are derived from the entire signatures, each set may have suffixes of varying lengths. If any one of these set is mapped directly onto a single ROM, memory utilization tends to be very low. There are many ways to modify the memory to improve its utilization. However, more logic is needed to build a more efficient memory. Since our goal is to minimize the logic resource, we present a simple modification to the memory that can double the memory utilization.

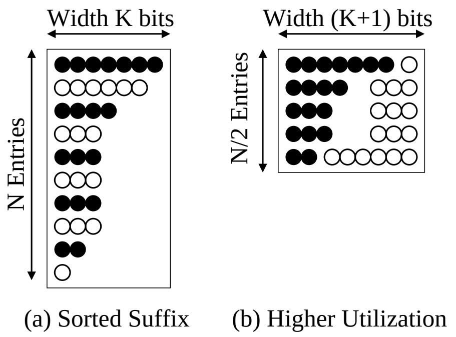


Figure 11. Rearranging data to increase memory utilization

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 11a. When these patterns are stored directly into the memory, nearly half of the memory is wasted. Our modification is aimed improve the utilization by filling in the empty spaces with the valid data. To do this, all the even entries are stored top to bottom as with a normal memory. Then all the odd entries are stored from the bottom to the top as shown in figure 11b; effectively reversing the bit-wise dimension of the entire memory. With this simple data rearrangement, the number of entries are reduced to half of the original memory.

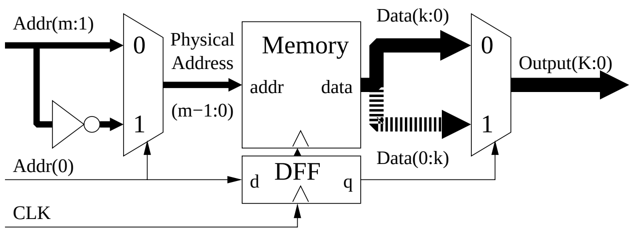


Figure 12. Wrapper logic is applied to the memory to improve its utilization.

In order to correctly read the rearranged memory entries, a small amount of wrapper logic is necessary. Figure 12 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 multiplexor 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.

4.4 Suffix Comparator

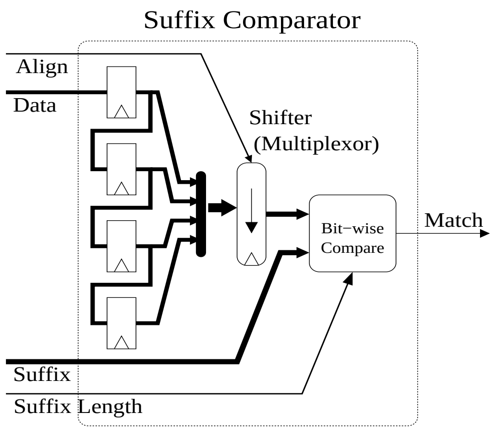


Figure 13. Incoming data is pipelined then aligned to compare against suffix.

Once the suffix is read from the ROM, the subsequent data is pipelined and shifted to the lineup at the comparator as in figure 13. 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 multiplexors or they are pipelined to multiple levels, depending on the width of the input bus. In addition to suffix data, ROM must store the length of each pattern. The length is decoded at the comparator to only compare the valid bits. When the incoming data is lined up with the corresponding suffix from the ROM, they are bit-wise compared according to length information.

If the rest of the incoming data matches the suffix, the module indicates the match. When the match signal is asserted, the address that was used on the ROM can be forwarded as an output to specify the triggered rule.

4.5 ROM Design Implementation

Although ROM based filters can be implemented in any FPGAs, we choose to use Xilinx Spartan 3 FPGA due to its low price. Block memory capacity of a Spartan 3 family chip is fixed at 18-kbit per memory unit. The Xilinx library contains template of many types of memory modules for a 18-kbit block memory in VHDL. Using these templates allow the Xilinx compiler to correctly use the built-in memory of the target FPGA. A single 18-kbit block memory can be configured as ROM of 1-bit by 16384 entries as well as 72 bit by 256 entries. Thus, we consider the dimension of the pattern sets, memory utilization, and average resource usage per signature before using a particular template.

Set	Signatures	Bytes	% of Total
1	495 rules	6,805 bytes	36%
2	212 rules	2,776 bytes	15%
3	134 rules	1,828 bytes	10%
4	94 rules	1,056 bytes	5%
5	64 rules	774 bytes	4%
Total	999 rules	13,239 bytes	70%

Table 1. Five largest pattern sets that satisfy the ROM constraints make up 70 percent of the entire Snort rules.

As with the RDL design, we implement 32-bit ROM based filters using the most current Snort signature set consisting of 1519 unique patterns. Before we generate the VHDL modules, the patterns are selected and divided using the rules defined in section 4.1. We found that 89.5 percent of all patterns implemented in the RDL based design can be reimplemented in the ROM based design. However, smaller pattern sets tend to not benefit from the ROM based method because of its large generic datapath. Thus, we implement the five largest pattern sets which totals 70 percent of the entire Snort rule set as shown in table 1. We implement each set into two different filter modules, one using the RDL method and the other with the ROM method.

4.5.1 Area and Performance

For our implementation, there is no benefit of using the improved ROM design described in section 4.3.1 for all the sets except for the largest one that contains 495 rules. This is because the rest of the pattern sets have less than 256 entries, which is supported by the memory configuration with the widest data bus. Therefore, only the largest ROM was converted to a higher utilization ROM.

Table 2 shows the resulting area in terms of D-flip flops, LUT4, and block memory for the implementations using the two methods. As expected, the largest pattern set benefited

	RDL			ROM		
Set	**Dff**	**Lut4**	**Mem**	**Dff**	**Lut4**	**Mem**
1	9,936	11,883	0	5,682	6,136	5
2	4,703	5,795	0	2,924	3,269	4
3	3,246	4,150	0	2,171	2,478	4
4	2,121	2,633	0	1,587	1,837	3
5	1,561	1,993	0	1,259	1,468	3

Table 2. Xilinx Spartan III XC3S2000 FPGA resource usage for pattern sets described on table 1. Capacity of a single block memory in Spartan III is 18 kbits.

the most from using the ROM based design. Overall results show that, for the functionally equivalent filter, the ROM design uses about half of the RDL design.

We implement the ROM design to run at the same performance as the RDL design. Even with a simplified address encoder, the critical paths for the ROM based designs still existed in the address encoder. Because of variability in of place and route process, performance measurement is only as good as the constraints that are placed on the design. Since the purpose of this section is to validate the effectiveness of the ROM design methodology, we successfully compile them with master clock constraints of 100 Mhz which, in turn, yields total bandwidth of 3.2 Gbps.

5 Comparison of Results

To evaluate of our design methods, on table 3, we compare our area and performance results with previous work of other projects. Two approaches described in this paper has the best gates per byte ratio, allowing the entire Snort rules to map on to a single Xilinx XC3S2000 FPGA. From experience in implmenting larger rule sets, we learned that the address encoder is system performance bottleneck. The latency of the address encoder is dependent on the size of its address space and its internal pipeline stages. Increasing the internal pipeline stages of the address encoder will increase performance of our design; but a better solution maybe obtained by dividing the rule set into number of smaller rule sets with smaller address space.

6 Conclusion

We have devised two effective methods of designing dynamic pattern search engines in FPGA.

The first method uses RDL to detect dynamic patterns in a stream of data. This design method, applied without any optimization, became too large to fit in a single FPGA. By reusing redundant logic within the entire design, the RDL

Design	Device	BW (Gbps)	# of Patrn	# of Bytes	Total Gates	Mem (kb)	Gates/ Byte
Cho-Msmith RDL8 w/Reuse*	***Spartan3 1500***	***2.00***	***1625***	***20800***	***16930***	***0***	***0.81***
Cho-Msmith RDL8 w/Reuse	***Spartan3 1500***	***0.80***	***1519***	***19021***	***15356***	***0***	***0.81***
Cho-Msmith ROM32 based	***Spartan3 2000***	***3.20***	***495***	***6805***	***6136***	***90***	***0.90***
Clark-Schimmel[3] RDL8	Virtex 1000	0.80	1512	17537	19698	0	1.10
Cho-MSmith RDL32 w/Reuse	***Spartan3 2000***	***3.20***	***1519***	***19021***	***26607***	***0***	***1.40***
Cho-MSmith RDL64 w/Reuse	***Spartan3 5000***	***6.40***	***1519***	***19021***	***47933***	***0***	***2.52***
Franklin et al. [6] Reg. Expression	VirtexE 2000	0.40	1	8003	20618	0	2.58
Clark-Schimmel[3] RDL32	Virtex2 8000	4.89	1512	17537	54890	0	3.10
Cho et al. [2] RDL32 Original	Altera EP20K	2.88	105	1611	17000	0	10.55
Gokhale et al. [7] CAM based	VirtexE 1000	2.18	32	640	~9722	24	15.19
Sourdis et al. [12] RDL	Virtex2 6000	8.06	210	2457	47686	0	19.41
Moscola et al. [9] Quad DFA	VirtexE 2000	1.18	21	420	~14660	0	34.90
Sidhu et al. [11] NFA Reg. Expr.	Virtex 100	0.46	1	29	1920	0	66.21

* Highly pipelined – maximum of 1-level gate delay between registers.

Table 3. Comparing area and performance results of FPGA based deep packet filters.

design ultimately shrunk down to less than one tenth of its initial hardware size.

The second method uses built-in memory in the FPGA and XOR based comparators. We show that this method further reduces the area when it is applied to a large pattern set.

We built our deep packet filter for the recent set of Snort rules (1519 to 1625 unique patterns) using RDL based architecture. Then we converted large subsets of the rules using ROM based method. As a result, what was once over 230,000 gates now fits on to a single Spartan 3 FPGA using at most 1.4 gates per byte of pattern.

In addition to small footprint, our implementations of the pattern inspection module are capable of filtering network traffic at 3.2 Gbps. Since the critical path for the implementation was determined to be in the address generation, its performance can be increased either by breaking up the patterns in a set or by deepening the address generator pipeline.

Our preliminary results using pipelined version of address encoder indicate that the design design can be accelerated to run at 8 Gbps without using any additions look-up-tables.

References

[1] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. In *Communications of the ACM*. ACM, July 1970.

[2] Y. H. Cho, S. Navab, and W. H. Mangione-Smith. Deep Network Packet Filter Design for Reconfigurable Devices. In *12th Conference on Field Programmable Logic and Applications*, pages 452–461, Montpellier, France, September 2002. Springer-Verlag.

[3] C. R. Clark and D. E. Schimmel. Scalable Parallel Pattern-Matching on High-Speed Networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004. IEEE.

[4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*, Stanford, CA, August 2003. IEEE Computer Society Press.

[5] I. Dubrawsky. Firewall Evolution - Deep Packet Inspection. *Infocus*, July 2003.

[6] R. Franklin, D. Carver, and B. L. Hutchings. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, Napa Valley, CA, April 2002. IEEE.

[7] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards Gigabit Rate Network Intrusion Detection Technology. In *12th Conference on Field Programmable Logic and Applications*, pages 404–413, Montpellier, France, September 2002. Springer-Verlag.

[8] J. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks. Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware. In *Military and Aerospace Programmable Logic Device (MAPLD)*, Washington DC, September 2003. NASA Office of Logic Design.

[9] J. Moscola, J. Lockwood, R. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2003. IEEE.

[10] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX LISA 1999 conference*, http://www.snort.org/, November 1999. USENIX.

[11] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2001. IEEE.

[12] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. In *13th Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, September 2003. Springer-Verlag.

[13] D. L. Steven A. Guccione and D. Downs. A Reconfigurable Content Addressable Memory. In *IPDPS 2000 Workshop*, Cancun, Mexico, May 2000. IEEE Computer Society Press.