Literature Review: Packet Classification and Inspection Techniques on FPGAs

Timothy Whelan

July 21, 2010

Introduction

As information systems increase in size and complexity the task of data acquisition and processing in and of such systems becomes ever more arduous and computationally expensive. To cope with the sheer size of the tasks required new and faster techniques and systems are required to be developed. Such environments are here demonstrated in a discussion of processing of packets collected from a network telescope. Network telescopes collect large numbers of network packets which can then be analysed to extract information about various security events that occur around the Internet. Identification of events of interest to an analyst and subsequent detailed inspection of events often requires searching through an over-whelming collection of packets for a relatively small portion that is useful. Such tasks have led to the need for methods to quickly discriminate between packets based on certain key criteria, such as source IP address for example. Systems that are able to perform this function on-the-fly have found a use in deep network packet inspection where packets crossing the edge of a network are inspected at all network layers for certain key phrases or strings. As will be shown, Field Programmable Gate Arrays (FPGA's) are well suited to such tasks due to their architecture and the nature of their operation which allow them to provide a good balance between the speed and ease of reconfiguring the devices for different problem parameters. Lastly a brief look is taken at other possible methods for fast data processing such as the use of Graphics Processing Units (GPU's) and application specific integrated circuits (ASIC's).

Description of network telescopes

A network telescope resides on a portion of IP space which should receive very little legitimate internet traffic [11]. Network telescopes passively collect data from networks and the data collected can be used to observe events on the In-

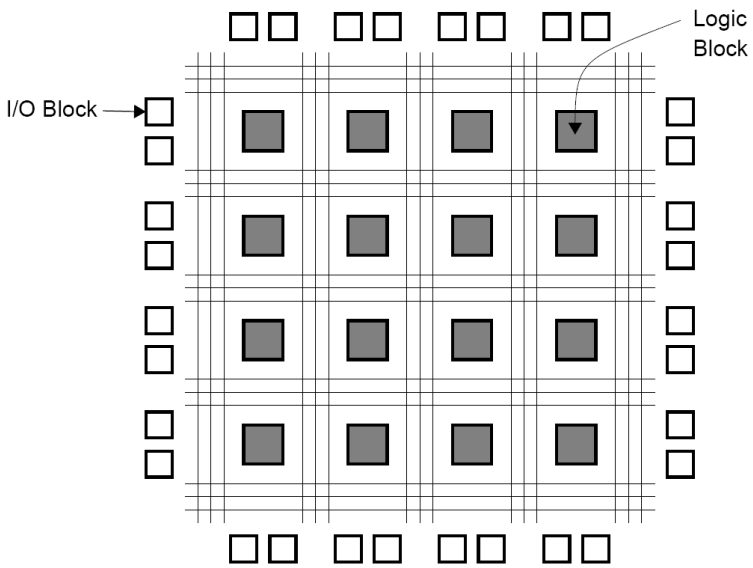

Figure 1: Structure of an FPGA

ternet. As can be seen in [21], useful attributes of network traffic one might measure include source and destination IP addresses and destination port. Reasonably, one might wish to extend this list to include source port addresses and protocols used in the packet if they can be identified.

Description of FPGA's

As described by the Altera Corporation in [1] FPGA's can fulfil any programmable function that could be implemented on an application specific integrated circuit (ASIC) but also provide the ability to reconfigure the FPGA to provide altered functionality. Whilst many other pieces of hardware exist that can be programmed to perform a certain function, and these hardware configurations can also be reprogrammed, FPGA's have become important in industry as the only field programmable logic devices that can provide a very high logic capacity on a single chip {fpds}. Simple programmable logic devices (PLD's) typically execute product and sum instructions on two levels of programmability; an AND-plane that AND's inputs together in an effective multiplication and an OR-plane that may or may not be programmable and which OR's its inputs [4, 9]. PLD's operate on two-state logic levels but FPGA's are designed to be able to operate using multi-level logic. The consequence of this design is that the complexity of the computational circuitry that can be programmed on an FPGA greatly exceeds that which can be placed on a PLD.

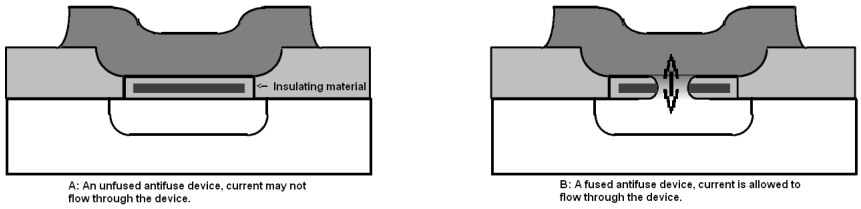

Figure 2: An antifuse semiconducting device shown before being programmed (fused) in A and after being programmed in B

Memory in FPGAs

The structure of an FPGA consists of configurable logic blocks with various interconnects between the configurable logic blocks as shown in figure 1 [9]. To provide functionality to an FPGA one specific interconnects there by linking the different logic components. FPGA's retain their functionality in three ways; the instructions programmed on an FPGA are stored in antifuse configurations, EPROM/EEPROM or SRAM blocks [9]. What follows is a brief description of three memory technologies used in FPGA's as illustrated in [9].

Antifuses

Antifuses are semi-conductor components that provide electronic insulation until a sufficiently high voltage is placed across the device at which point the insulating ability of the components of the device is destroyed – the antifuse behaves as a typical fuse but is meant to be 'fused' to allow current to flow through the device. Antifuses however cannot be 'unfused' hence once the antifuse block of an FPGA has been programmed it cannot be reprogrammed [9]. Figure 2 is a diagrammatic sketch of an antifuse.

Static Random Access Memory

To allow the FPGA to be reprogrammed they are provided with blocks of static random access memory, or SRAM. SRAM bit construction is shown in figure 3. When a write instruction, logic 1, is placed on the gate of the control transistor data can flow into the inverter system which maintains any logic level placed in it. When logic 0 is placed on the gate of the control transistor then no change can occur to the logic placed in the inverter system while it is supplied with power. Should power to the SRAM bit be lost however then the data written to the SRAM bit will be lost. Therefore a weakness of FPGA's that are built using SRAM blocks of memory is that each time the device is powered on it must be reprogrammed.

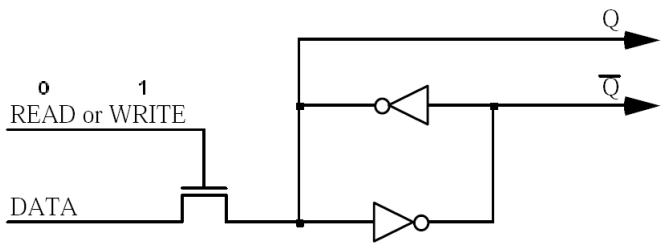

Figure 3: A single bit of SRAM

Erasable Programmable Read Only Memory (EEPROM)

The weakness of using volatile SRAM in FPGA's can be mitigated by making use of EPROM, a diagram of which can be seen in figure 4. In short, a voltage is placed across the gate structure of an EPROM bit thereby inducing a change in the internal structure of the semi-conductor. This change in structure can be reversed by shining UV light on the device which causes the device to revert back to its original state. The advantage of using EPROM over SRAM as the memory that stores the functionality of the FPGA is that the change in the device's structure does not need to be maintained by supplying power to the device which means that the device need not be programmed upon powering up but rather only once when the program logic is first transferred to the device.

General FPGA structure

The general structure of FPGA's is that of many configurable logic blocks (CLB's) that are interconnected by routing lines of different types that perform slightly different routing functions [22, 9]. Figure 4 shows the structure of a Xilinx Spartan-II FPGA's CLB slice, a portion of the CLB. There are two such slices on a CLB [22]. What follows is the structure of the Xilinx Spartan-II family of FPGA's taken from the Spartan-II databook [22]. This description, while for a specific class of FPGA, serves to demonstrate the general principles on which FPGA's operate.

As can be seen in figure 4, a CLB slice is composed of a look up table (LUT), carry and control circuitry and a D-type flip-flop which can store the output from the carry and control circuitry. This configuration is termed a logic cell (LC) and a CLB may have one or more LC's that together make up a slice of a CLB. Multiple slices then make up a CLB. To program the FPGA with certain functionality the FPGA design software, normally supplied by the FPGA vendor, maps the high level logic designed by the FPGA programmer into multiple single-step functions and these functions are then mapped onto the LUT. The LUT configurations are stored using one of the memory technologies discussed above. LUT's function by specifying the required output for all inputs to the LUT. The results from CLB's are then routed around the FPGA using the

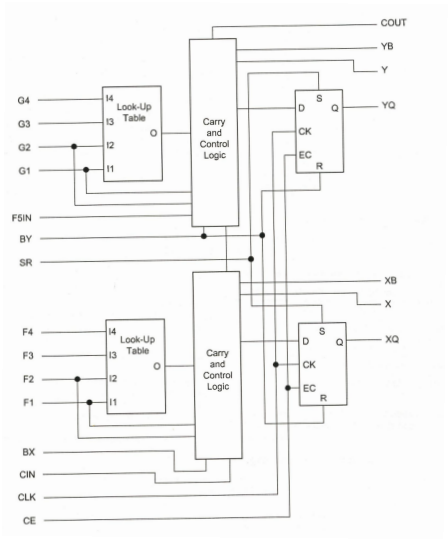

Figure 4: Spartan-II CLB slice

available routing paths placed on the FPGA. Routing on the FPGA is provided to perform various different functions such as:

- routing between LC's in a CLB to minimise routing delays for functions requiring more than one LUT to describe,
- direct routing paths between adjacent CLB's for fast binary arithmetic between adjacent CLB's,
- routing channels through general routing matrices that form meeting points for horizontal and vertical routing paths that also connect adjacent routing matrices – general routing matrices specify whether or not connections exist between the lines entering the general routing matrix.

Example FPGA applications

FPGA's are suitable for a number of high speed processing applications, especially ones where components can operate in parallel. What follows is a partial list of applications of FPGA's:

1. FPGA's have been successful platforms on which to implement artificial neural networks [25],
2. [7]demonstrates a system developed to implement the Advanced Encryption Standard encryption algorithm at a rate of 150Mbs,

3. FPGA's have proved a viable replacement for dedicated digital signal processing (DSP) chips and ASIC's [8],

4. Central to the motivation for this literature review is the work done in using FPGA's to perform packet classification and deep packet inspection[6, 3, 12, 18, 24]. As can be seen in the sources mentioned, deep packet inspection also often involves string matching within the payloads of packets.

The focus of the rest of this paper will be on FPGA based methods for packet processing such as packet classification and deep packet inspection.

FPGA Methods for Packet Processing

As mentioned before, the motivation for this literature review is to examine and assess work carried out in the field on network packet processing particularly on FPGA platforms. The literature reviewed decomposes packet processing into two separate actions: packet classification, typically based on packet headers, and deep packet inspection, typically string matching within packet payloads to identify data being transferred in the packet [17, 18].

Packet classification

To understand the desire to research improved methods for processing of packet headers one can examine the uses of packet headers and the decisions that can be taken pertaining to the handling of a packet once its headers have been inspected. Song and Lockwood mention the use of network packet headers in NIDS in [17]. The rules used in the Snort IDS typically include a 5-tuple of packet headers, source and destination IP addresses and port numbers and the protocol being used [17]. The IDS also uses pattern matching, as described later in this review, when matching rules but uses the header processing to provide a context in which the pattern matching results can be better utilized [17]. Ravindran et al describe packet routing and forwarding as another application that benefits from improved methods in header processing [14].

There are many techniques for packet classification based upon multiple header fields of a packet and a description of these techniques can be found in [13, 20]. A brief range of these methods are related below.

Linear Search

Suppose that each filter in a set of filters were sequentially compared with the headers of an inspected packet until either a filter is found that matches the packet's header fields or all filters are unsuccessfully matched to the filter; this

would constitute an exhaustive, linear search for filters that match the packet headers [13]. This approach to packet classification is straight-forward and reliable {Nottingham, taylor} but provides poor performance [20] and makes poor use of memory [13, 20] with O(N) memory requirements and performance decreasing proportionally to the size of the filter set [20]. Taylor does demonstrate a means of reducing the memory requirements of a filter set when using exhaustive search techniques but the computational behaviour of exhaustive linear searches still make it prohibitively slow for applications requiring high throughput though it can be used in conjunction with other algorithms in a more complex system [13, 20]. Linear search techniques do lend themselves well to systems that can provide parallel processing [13]. Taylor suggests that if a linear search is at one end of a spectrum of searching techniques, with linear memory and search computational complexities, then ternary content addressable memory would be at the opposite end of the spectrum with O(1) computational complexity. Ternary content addressable memory is discussed further below.

Ternary Content Addressable Memory (TCAM)

TCAM is a type of memory that can store a "don't care" state as well as a 1 or 0 bit value in the memory [17] hence it is a tri-state memory as described by the word "ternary" in its name. The operation of CAM memory is practically the converse of conventional memory; a desired piece of data contained in memory is specified and what is returned is the address or a list of multiple addresses where the content resides in memory [23]. TCAM is also optimised to search for content in memory at multiple locations in parallel [20, 23] and all these features of TCAM memory make it an obvious tool in packet classification based on header processing as indicated in the related work mentioned in [17] which will be discussed later to illustrate a hybrid algorithmic approach to the problem of packet classification. However, TCAM technology has significant drawbacks as described by Taylor [20]:

- TCAM memory has a greater cost per bit than other memory technologies
- TCAM memory suffers from inefficient use of storage space because it cannot store arbitrary ranges of numbers but instead such ranges need to be converted into '<x' and '>y' descriptions (where x and y are bit string prefixes) and extra resources are required to store the "don't care" value
- TCAM memory makes use of more transistors than other memories leading to greater power consumption
- TCAM memory does not scale very well when input to the TCAM is long.

The above two techniques perform exhaustive searches over the entire filter set, albeit with vastly differing memory and time requirements. The following techniques perform restructuring or either the filters and/or the incoming packet's fields to transform the packet classification problem into another class of problem and leverage observations that can be made about filter sets.

Filter	Destination	Source
F_1	0*	10*
F_2	0*	01*
F_3	0*	1*
F_4	00*	1*
F_5	00*	11*
F_6	10*	1*
F_7	*	00*

Figure 5: Example filter set (taken from [19])

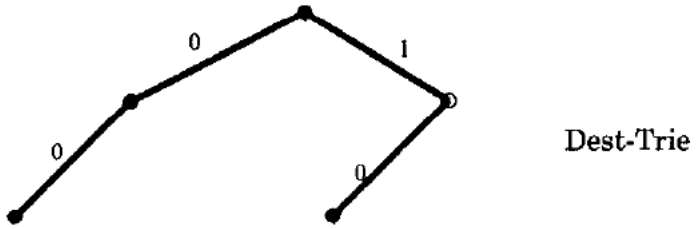

Figure 6: A) Dest-trie representing the source addresses in the filter set. The open circle represents an invalid destination address prefix. [19]

Grid-of-tries

This approach is described more formally in [19] but a brief description is provided below. Suppose an example filter set is as in figure 5 (note that this initial set represents only source and destination addresses).

Firstly, a tree structure is constructed based upon the destination addresses of the filters (figure 6a). This enables any incoming packet to be tracked to a node in this tree (the dest-trie). The leaf nodes of the dest-trie contain pointers to similar trees that represent decision trees for source addresses relevant to the destination address represented at the originating leaf nodes of the dest-trie as shown in figure 6b.

The grid-of-tries structure described above can be optimised in many ways and Srinivasan et al go on to describe some of these optimisations. Notably, the memory requirements of this structure can be reduced by not repeating filters in multiple source-tries and search costs can be reduced by placing pointers from one source-trie node to nodes in other source-tries so that when a search down one source-trie branch fails a pointer can be followed to another source-trie and the search can continue without having to perform redundant searches along higher branches before progressing beyond what was already processed at the

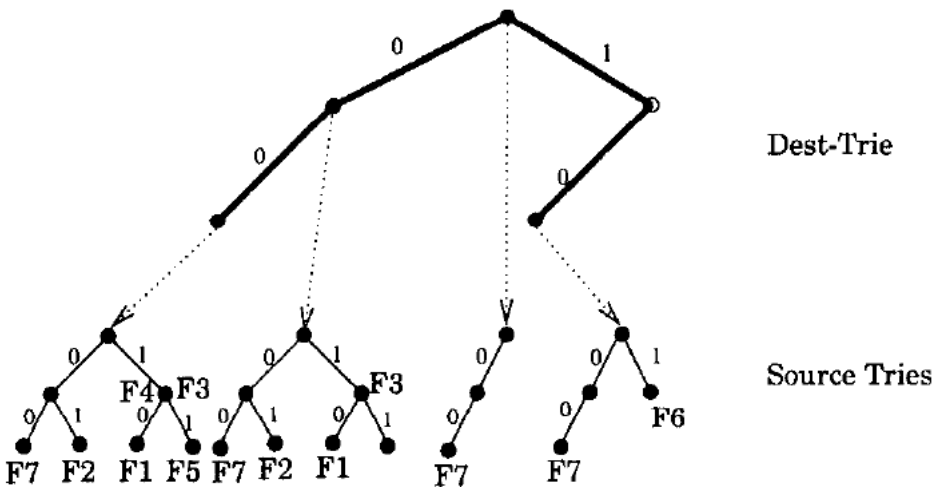

Figure 6: B) The general trie structure showing the pointers from dest-trie to relevant source-trie filters. Note that source address filters are repeated when one dest-trie filter is a prefix of another dest-trie filter. [19]

point of failure [19].

Tries however only work well for 2-dimensional filters [13, 20, 19]. The structure of tries also means that they are suitable for IP address masks but are not suitable for arbitrary port ranges without changing the ranges specifications into prefixes, a task which can significantly increase the number of tries required to adequately describe the desired filters [13, 20, 19]. As will be discussed later, the cost of using a trie structure to perform port filtering can be made acceptable provided other filter optimisation techniques are employed and tries have indeed been used in conjunction with other techniques such as cross-producting (to be discussed in the next section) in [19] and TCAM's in [17].

Cross-producting

Srinivasan et al introduced another technique in [19] as well as the grid-of-tries approach to packet classification: cross-producting. Cross-producting operates by performing 'best matching prefix' searches on each field in the filter, possibly in parallel [13] given that these searches can be done independently [20], and then combing the results of these independent searches into one more search to provide a final overall classification. While searching each field independently for a best matching prefix is relatively simple an efficient method is required to combine the search results into a final usable result [19]. The scheme proposed by Srinivasan et al proposes dividing filters into columns which group the different field values by field type. Then a cross-product is formed of these sets to create a list of all possible combinations of field values that can be made from the field values in the original filter set. An example to illustrate this process is in figure 7.

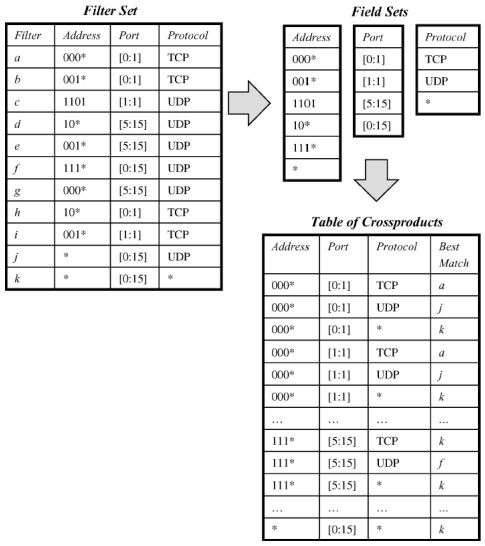

Figure 7: The process of forming cross-products from [20].

Cross-producting requires a prioritisation of fields to enable the final cross-products to be more closely aligned with one filter than another filter as once the cross-product of filters has been performed each cross-product is labelled with a filter that best matches the cross-product. When a packet is received and the closest field prefix for each field has been located independently then this combination of individual field filter prefixes is used as a hash to locate the closest matching filter from the cross-product table.

Cross-producting, while providing a high throughput, has exponential memory requirements, O(Nd) [13, 20]. Srinivasan et al therefore proposed a hybrid approach to packet classification using their grid-of-tries technique to classify based on destination and then source matching and cross-producting to match ports and flags [13, 20, 19].

Bit Vector

The bit vector classification technique views the packet classification problem as a geometric problem with each D-field filter specifying a region in D-dimensional space and the classification equates to locating which named region a packet can be said to be placed in [13, 20], an illustration of this view is given in figure 8 below.

A brief, simplified, algorithm is adapted from [13] and [20] and described below. Note that there are N filters and D types of fields in each filter.

1. Prioritise the filter rules (if priority is irrelevant then priorities can be arbitrary)

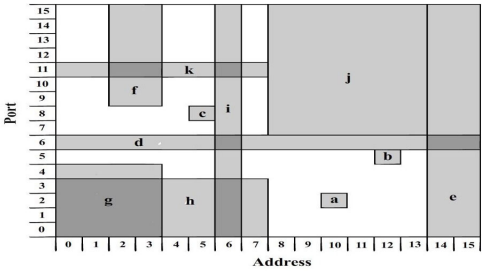

Figure 8: A geometric view of the packet classification problem. Filtering on 2 field (Port number and address) results in a 2 dimensional filter. The shaded regions with letters represent filters and filters may overlap (darker regions). [13]

2. For each axis
   (a) Divide the axis into regions bounded by the points where boundaries of regions in D-space that are orthogonal to the current axis may intersect the current axis.
   (b) For each region along the current axis
      i. Assign an N-bit bit vector equal to zero
      ii. If region x in D-space, described by rule x, lies on the current region
         A. Then set bit x of the current region's bit vector to 1

A graphical representation of a geometric view of the classification problem can be seen in figure 8 and a graphical representation of the result of the algorithm can be seen in figure 9 above.

When an incoming packet is received its fields are decomposed and each one is processed independently of the others to locate which filter pertains to the packet with the given field value. Alternatively one can describe the process as locating which region along each of the D axes the packet might belong to. The D bit vectors (1 bit vector result from each field or dimension) are then AND'ed together to give a final bit vector result with each 1 bit representing a rule that pertains to the packet fields as a group e.g. a packet with port number 8 and address 5 will return bit vectors 001 0100 0110 and 001 1000 1001 as seen in figure 9. When these bit vectors are AND'ed the result will be 001 0000 0000 indicating that filter c classifies the packet. If the rules in each dimension are ordered according to some priority scheme then the significance of the 1 bits in the final bit vector will match the priority of the rules in the rule set. Li et al {lucentbitvect} describe a bit vector implementation that uses trie structures instead of taking the multidimensional geometric view described above.

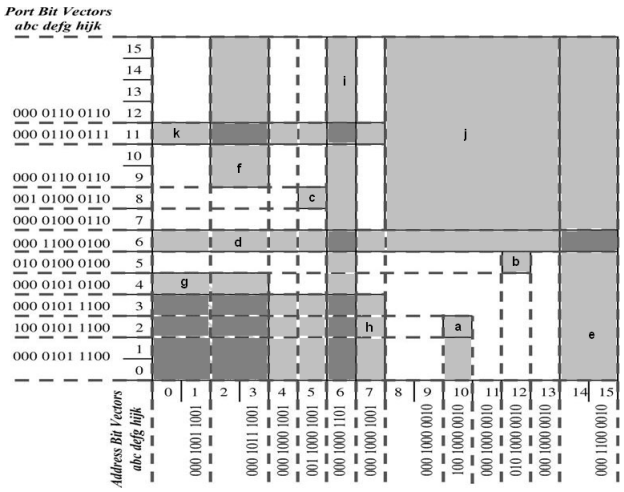

Figure 9: Regions in D-space showing filters and bit vectors for regions along each dimension (N = 11; D = 2) (Figure adapted from [13])

Tuple Space

Sirinivasan et al {tuplespace} introduced the tuple space classification algorithm and Taylor {taylor} provides a less formal description of the tuple space algorithm. The tuple space technique makes the observation that while there may be many filters in a filter set, the number of different lengths of prefixes in the fields of the filter is typically significantly less than the number of filters {taylor, tuplespace}. For example, while a 32-bit IP address could conceivable provide 100's of different IP ranges and specific IP addresses to be used in a filter, there can only be 32 different lengths of IP address masks or prefixes in the filter set. Using this observation, a description of a filter in a tuple space technique indicates the number of bits in the prefix describing each field in the filter and the concatenation of all these indicating numbers is called a tuple {taylor, tuplespace}. An example from {tuplespace} supposes that if there are two 2-dimensional filters, F1 = (01*, 111*) and F2 = (11*, 010*) then a tuple which could describe the filters is [2, 3].

Port numbers in a filter set are frequently specified using ranges which make prefix representations inconvenient {tuplespace} hence Sirinivasan et al introduced an encoding scheme whereby port ranges are divided up into Nesting Levels with non-overlapping ranges and each range within a level is given a locally unique Range ID {taylor}. Port ranges are then specified within the tuple using the nesting level that the range is located in. An example of the complete tuple creating process is given in figure 9 taken from {taylor}. The description of the protocol field within the tuple was simply a '1' if the protocol was specified exactly or '0' if the protocol field filter contained a ternary character '*' {taylor}.

(Address fields are 4-bits and port ranges cover 4-bit portnumbers.)

Filter	SA	DA	SP	DP	Prot	Tuple
a	0∗	001∗	2 : 2	0 : 15	TCP	[1, 3, 2, 0, 1]
b	01∗	0∗	0 : 15	0 : 4	UDP	[2, 1, 0, 1, 1]
c	0110	0011	0 : 4	5 : 15	TCP	[4, 4, 1, 1, 1]
d	1100	∗	5 : 15	2 : 2	UDP	[4, 0, 1, 2, 1]
e	1∗	110∗	2 : 2	0 : 15	UDP	[1, 3, 2, 0, 1]
f	10∗	1∗	0 : 15	0 : 4	TCP	[2, 1, 0, 1, 1]
g	1001	1100	0 : 4	5 : 15	UDP	[4, 4, 1, 1, 1]
h	0011	∗	5 : 15	2 : 2	TCP	[4, 0, 1, 2, 1]
i	0∗	110∗	2 : 2	0 : 15	UDP	[1, 3, 2, 0, 1]
j	10∗	0∗	2 : 2	2 : 2	TCP	[2, 1, 2, 2, 1]
k	0110	1100	0 : 15	0 : 15	ICMP	[4, 4, 0, 0, 1]
l	1110	∗	2 : 2	0 : 15	∗	[4, 0, 2, 0, 0]

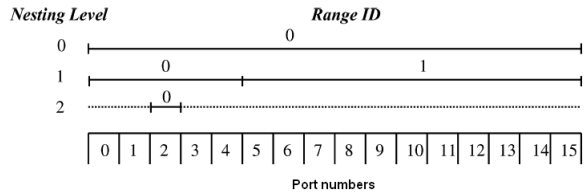

Figure 10: A table of tuples derived from a filter set and a diagram of the associated encoding of port ranges.[20]

A variation of the tuple space searching method, called the Pruned Tuple Space Search described in {tuplespace, taylor} creates trees of the source and destination prefixes with nodes containing tuples that may match a packet that maps to that node in the tree. By mapping the source and destination address of a candidate packet to these trees two list were obtained of tuples and the intersection of these lists provided the tuples that needed to be searched for filters that may match the candidate packet.

The BV-TCAM Architecture (A hybrid technique)

The authors in [17] decided their system would be required to report all rules which were matched by a packet instead of just the rule with the greatest priority that was matched by the packet. The authors also used a hybrid algorithm in their approach to solving the packet classification problem and their approach made use of Ternary TCAM and a grid-of-tries. Bit vector outputs from these two schemes were then combined thus they labelled their architecture the BV-TCAM architecture [17].

TCAM technology may be fast but it is however expensive in terms of the space required hence it is very beneficial to limit the number of entries required in TCAM memory to be able to perform effectively by careful structuring of the entries [17]. Song and Lockwood achieve this by observing that many rules in a rule set typically share common IP addresses and protocols but differ when specifying ports or port ranges. This led the authors to use TCAM technology to check matches in IP address and protocol fields. Port specifications were checked using a bit vector algorithm as TCAM is not very well suited to checking the

arbitrary ranges that one finds for port fields in filters as mention earlier in the discussion focussing on TCAM technology.

The output of the IP addresses and protocol matching TCAM mechanism was encoded in a separate bit vector which had a bit length equal to the number of rules being checked. If a packet matched a certain rule then the corresponding bit in the bit vector was set to 1 otherwise it remained set to 0. To match port numbers the authors created a trie structure, The port numbers were mapped down the trie with branch selection governed by the presence of a 1 or 0 at the currently inspected bit of the port number binary value, this is the origin of the bit vector description in the algorithm. Branches in the trie structure extend until either they represent the exact port address to be matched or, in the case of port ranges, the prefix of the port number represented up to that point is sufficient to describe the range of port numbers. Each leaf node of the trie then contained a bit vector indicating which rules had been matched in an identical method to that reported from the TCAM segment of the classification unit. The combination of these two segments, the TCAM and the BV method, then produced a final result indicating which rules a given packet had matched.

The BV-TCAM architecture developed by Song and Lockwood achieved a throughput capable of operating at a rate of 2.488 Gbps, or OC48 [17]. It is therefore also a good demonstration of how separate technologies and algorithms can be combined to form a hybridized system.

Deep Packet Inspection

As mentioned earlier, deep packet inspection forms a key part of packet processing for the aim of defence [12]. This often requires scanning the payload of packets, and the packet headers in some instances, to locate malicious data embedded within the packet by attempting to locate specific search strings within packets. [5, 6, 3, 12, 18] demonstrate different algorithms for such tasks which will be discussed below.

Cho et al [6], like a few of the other sources that discuss deep packet inspection [5, 18], make reference to Snort IDS rule sets. Briefly, "Snort is an open source network intrusion prevention and detection system" [10] that is able to detect packets on network interfaces and perform various actions based upon different properties of the packets detected [16]. This is done by applying rules contained in a rule file that form a rule set to each packet that Snort detects [16]. Relevant to deep packet inspection is the ability of Snort to locate strings specified in Snort rules within network packets [16].

Simple N parallel rule checks

A simple approach described in [5] is to have N comparators to match N strings for N rules. Each rule can then be treated independently and matching for all rules can occur in parallel. An illustration of the system is shown in figure 11 [5] where one can see that some initial processing on the packet headers is done

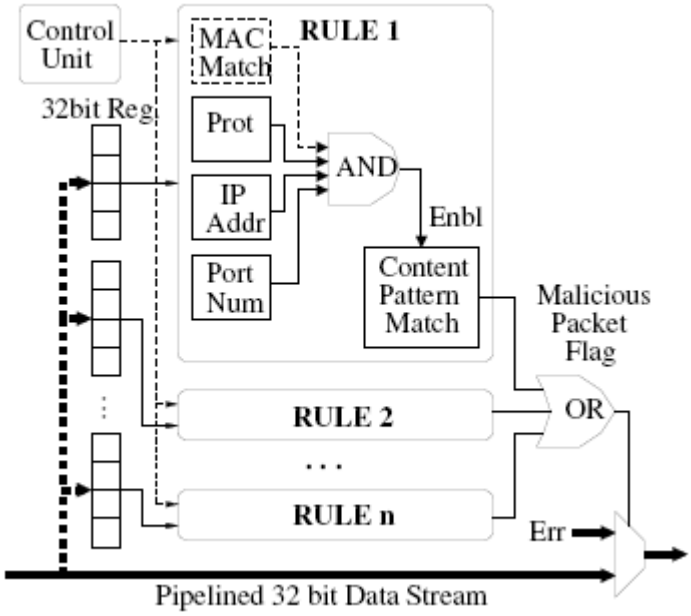

Figure 11: The structure of the system described by Cho and Magione-Smith [5]

first to check which rules for packet content matching need to be checked. The authors of [5] go on to address the problem of determining which rule has been matched once any match has been detected.

Figure 12 shows how the content matching portion of the design operates by matching a four byte portion of the packet against a sample string with four different offsets.. If one assumes that a packet will only match one rule then the index of the rule matched is encoded in a binary number whose bits are used to navigate a path down a binary tree to the matched rule.

If more than one pattern is matched in a packet however then it becomes impossible to determine which patterns have been matched by looking at the index encoder output. Cho and Magione-Smith [5] then propose dividing the rules into multiple sets with none overlapping patterns, so that no more than one rule in a set can be fired by a packet at time, and assigning to each set an index decoder as described above. This however has obvious increasing space requirements as the number of required index encoders increases with the number of rule sets.

Sourdis and Pnevmatikatos demonstrate in [18] a system similar to that of Cho and Magione-Smith described above but make use of a fan-out structure to essentially copy the packet to many comparator networks that check for the occurrence for patterns in the packet as explained earlier in [5]. Other noticeable differences between the approaches of Cho and Magione-Smith compared to that

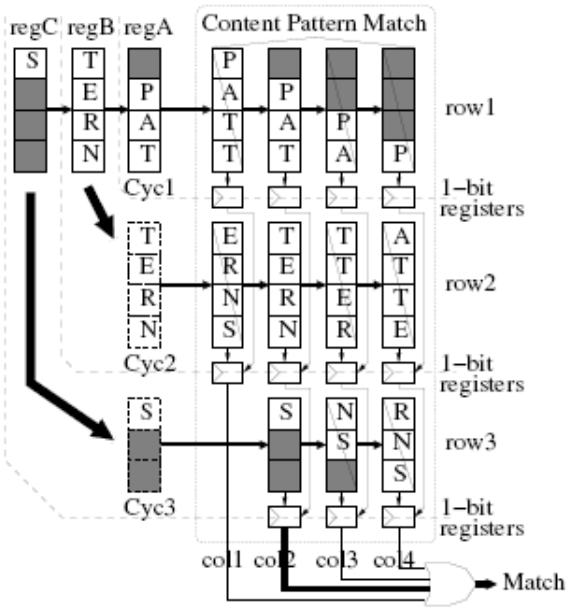

Figure 12: The structure of the content pattern matcher developed by Cho and Magione-Smith [5]

of Sourdis and Pnevmatikatos are:

- the assumption by Sourdis and Pnevmatikatos that only one pattern will be matched in their system at a time, provided that different pattern suffixes are used [18], enabling the use of a simple index encoder similar to that proposed in [5],
- and careful design of the pipelined comparator so as to conveniently make use of the one 4-input LUT and single flip-flop to identify half bytes of characters and then AND-ing the outputs of the LUT's to form a complete pattern match output.

The technique of matching the pattern to be identified at different one byte offsets within packet used in [5] is also repeated in [18].

Deterministic Finite State Automata (DFA's) in FPGA's

An approach to pattern matching within a sample space is to use deterministic finite state automata and these have been successfully implemented on FPGA's as the work of Moscola et al demonstrates in [12].

In their work, Moscola et al first demonstrate briefly how a pattern composed of characters from a finite alphabet can be described using regular expressions, an example is given below: "Vi(R|r)u(S|s)" would represent the patterns Virus,

ViRus, ViruS and ViRuS. Symbols such as '*' can be used to represent multiple occurrences of characters or character sets within a pattern [12].

Moscola et al then state the benefits of DFA's over Non-deterministic Finite Automata (NFA's), namely the less space required for a completed DFA than for a completed NFA without consideration for the space required during construction of either automaton [12]. Moscola et al parse the pattern for each rule through an application called JLex which constructs a DFA that can identify the regular expression for the desired pattern and the output from JLex is then used to generate VHDL source code which is then used to program the FPGA in the final implementation (VHDL is a hardware design language used to specify the functionality of FPGA's).

The action of the system presented in [12] is described as follows:

1. various protocol wrappers remove the different protocols that may surround the payload of a packet,

2. the packet is then routed through multiple content scanning modules which use the DFA's programmed onto the FPGA to search for the patterns to be identified,

3. once the packet payload has passed through all content scanning modules the appropriate actions are taken with regard to the patterns identified,

4. appropriate actions may include dropping the malicious packet, sending an alert message to a specified network address and outputting the packet from the content scanning module after wrapping the packet in the protocol headers stripped from the packet prior to searching.

Figure 13 shows a block diagram of the content scanning module presented by Moscola et al as it appears in [12].

Moscola et al state that their system presents data to the content scanning module at 32 bits per clock cycle and note that their system passes data to their regular expression DFA's - (REn DFA) in the diagram above – at a rate of 8 bits per clock cycle. Since this would produce a back log in the system Moscola et al designed their system to present packets to one of four content scanning modules selected in a round-robin method thereby achieving a 32 bit per clock cycle processing rate to match the 32 bit per cycle incoming and outgoing rates of the system.

An alternate approach describe by Baker and Prasanna in [2] which 'uses a modified version of the KMP (Knuth, Morris, Pratt) algorithm' [2]. Briefly, the KMP algorithm is able to detect possible occurrences of a pattern within a string and upon a mismatch does not start the next search from very next offset in the string but rather from the next possible start of the pattern in the string [15]. Suppose the pattern is 123 and the string is 121231 the first search for the patter will start at offset 0 but when a mismatch occurs at the third character then the algorithm starts the next possible match at the third character rather than the next offset (which would be at the second character) because the next position in the string at which the patter may be matched

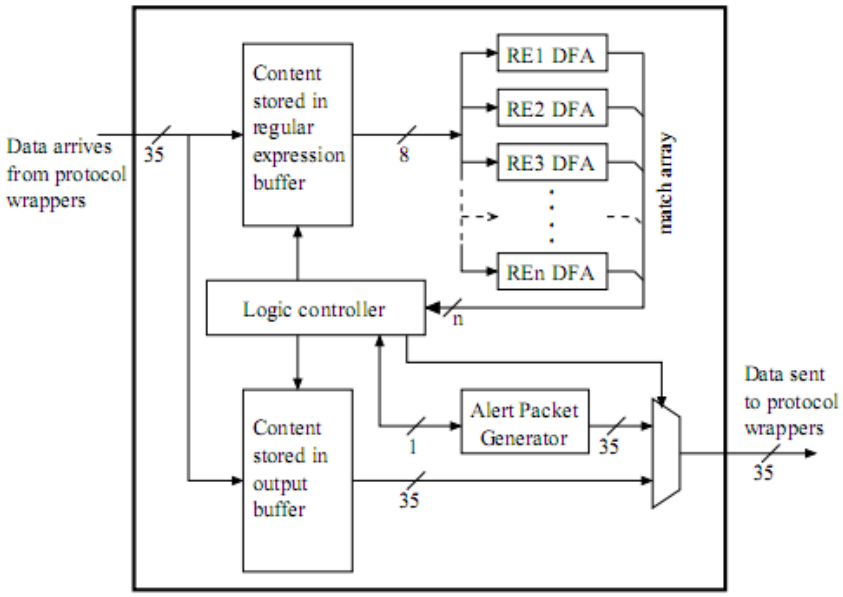

Figure 13: The DFA content scanning module presented in [12].

is the third character. The KMP algorithm has worst case behaviour O(k + n) [2]. Baker and Prasanna state that even though other algorithms may have better average case performances than the KMP algorithm no other algorithm has a more efficient worst case performance. Baker and Prassana then describe this property of the KMP algorithm as important because an attacker may try to flood an intrusion detection system in an attempt to force through some malicious packets past the IDS; obviously the best way for an attacker to do this would be to construct packages of the worst case for the IDS. The result of using the KMP algorithm is that this worst case then has O(k + n) behaviour [2].

The contribution of Baker and Prasanna is their demonstration that by adding a second comparator and an input buffer (the minimum size of which Baker and Prasanna derive) the KMP algorithm can be used and the resulting system can always accept at least one input character per cycle of operation [2]. This is an obvious improvement over the traditional approach of using a single comparator which results in the system being able to accept at most one input character per cycle of operation [2], a condition that can be seen to lead to the type of attack mentioned earlier where worst-case packets are injected into the input stream in an attempt to flood the IDS.

Conclusion

As can be seen from this review of literature surrounding packet processing using FPGA's, the use of FPGA's for wire-speed packet processing is well known in the area of packet classification and packet field processing with research dating back to 1998 being presented in this review. However, it can be anticipated that the more difficult task of packet payload inspection might still have much to offer computer scientists who wish to study this area of FPGA utilization.

References

[1] Altera, 2010. Altera Corporation webpage.

[2] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on fpgas. In *International Symposium on Field Programmable Gate Arrays*, pages 223 – 232, 2004.

[3] Peter Bellows, Jaroslav Flidr, Tom Lehman, Brian Schott, and Keith D. Underwood. Grip: A reconfigurable architecture for host-based gigabit-rate packet processing. In *In: Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 121–130. IEEE Computer Society Press, 2002.

[4] Stephen Brown and Jonathan Rose. Architecture of fpgas and cplds: A tutorial. *IEEE Design and Test of Computers*, 1996.

[5] Young H. Cho and William H Mangione-Smith. Deep network packet filter design for reconfigurable devices. In *ACM Transactions on Embedded Computing Systems*, 2008.

[6] Young H. Cho, Shiva Navab, and William H Mangione-Smith. Specialised hardware for deep network packet filtering. In *International Conference on Field Programmable Logic and Applications*. The University of California, 2002.

[7] Pawel Chodowiec and Kris Gaj. Very compact fpga implementation of the aes algorithm. In *5th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 2779, pages 319–333, 2003.

[8] Gregory Ray Goslin. A guide to using field programmable gate arrays (fpgas) for application-specific digital signal processing performance, 1995. Xilinx Incorporated.

[9] Scott Hauck. The roles of fpgas in reprogrammable systems. In *Proceedings of the IEEE*, volume 86, pages 615–639, April 1998.

[10] Sourcefire Inc. webpage.

[11] David Moore, Colleen Shannon, Geoffery M. Voelker, and Stefan Savage. Network telescopes: Technical report. Technical report, San Diego Supercomputer Center, University of California and Computer Science and Engineering Department, University of California, 2004.

[12] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a content-scanning module for an internet firewall. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 2003.

[13] Alastair Nottingham and Barry Irwin. Gpu packet classification using opencl: A consideration of viable classification methods. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 160 – 169, 2009.

[14] Kaushik Ravindran, Nadathur Satish, Yujia Jin, and Kurt Keutzer. An fpga-based soft multiprocessor system for ipv4 packet forwarding. In *In Proc. 15th International Conference on Field Programmable Logic and Applications (FPL-05*, page 487492, 2005.

[15] Mireille Regnier. *Lecture Notes in Computer Science*, chapter Knuth-Morris-Pratt algorithm: An analysis, pages 431 – 444. Springer Berlin / Heidelberg, 1989.

[16] Martin Roesch and Chris Green. *SNORT Users Manual 2.8.6*. The Snort Project.

[17] Haoyu Song and John Lockwood. Efficient packet classification for network intrusion detection using fpga. In *International Symposium on Field-Programmable Gate Arrays*, 2005.

[18] Ioannis Sourdis and Dionisios Pnevmatikatos. *Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System*. Springer Berlin / Heidelberg, 2003.

[19] V. Srinivasan, G. Vargheset, S. Suri, and M. Waldvogelg. Fast and scalable layer 4 switching. In *ACM SIGCOMM Computer Communication Review*, 1998.

[20] David E. Taylor. Survey and taxonomy of packet classification techniques. ACM Computing Surveys, 2005.

[21] Jean-Pierre van Riel and Barry Irwin. Identifying and investigating intrusive scanning patterns by visualizing network telescope traffic in a 3-d scatter plot. In *Proceedings of 6th Annual Information Security Conference*, 2006.

[22] Xilinx. *The Programmable Logic Databook 2000*.

[23] Xilinx. *Xilinx Content-Addressable Memory v6.1 Product Specification*, 2008.

[24] Cho H. Young, Shiva Navab, and William H. Mangione-Smith. Specialized hardware for deep network packet filtering. In *Proceedings of 12th International Conference on Field Programmable Logic and Applications*, 2002.

[25] Jihan Zhu and Peter Sutton. Fpga implementations of neural networks - a survey of a decade of progress. In *Proceedings of the 13th Annual Conference on Field Programmable Logic and Applications*, pages 1062–1066, 2003.