Ultra-High Throughput String Matching for Deep Packet Inspection

Alan Kennedy, Xiaojun Wang, Zhen Liu
School of Electronic Engineering
Dublin City University
Dublin 9, Ireland
{alan.kennedy, liuzhen, wangx} @eeng.dcu.ie

Bin Liu
Department of Computer Science and Technology
Tsinghua University
Beijing, P.R.China
liub@tsinghua.edu.cn

Abstract—**Deep Packet Inspection (DPI) involves searching a packet's header and payload against thousands of rules to detect possible attacks. The increase in Internet usage and growing number of attacks which must be searched for has meant hardware acceleration has become essential in the prevention of DPI becoming a bottleneck to a network if used on an edge or core router. In this paper we present a new multi-pattern matching algorithm which can search for the fixed strings contained within these rules at a guaranteed rate of one character per cycle independent of the number of strings or their length. Our algorithm is based on the Aho-Corasick string matching algorithm with our modifications resulting in a memory reduction of over 98% on the strings tested from the Snort ruleset. This allows the search structures needed for matching thousands of strings to be small enough to fit in the on-chip memory of an FPGA. Combined with a simple architecture for hardware, this leads to high throughput and low power consumption. Our hardware implementation uses multiple string matching engines working in parallel to search through packets. It can achieve a throughput of over 40 Gbps (OC-768) when implemented on a Stratix 3 FPGA and over 10 Gbps (OC-192) when implemented on the lower power Cyclone 3 FPGA.**

I. INTRODUCTION

Network intrusion detection/prevention systems used for the deterrence of malicious attacks such as worms, viruses and Denial of Services depend heavily upon Deep Packet Inspection (DPI). Currently the use of DPI is limited to end-hosts. This is because edge and core routers do not have the processing power needed to inspect the entire content of a packet at wire speed. Typically edge routers only inspect the header of a packet carrying out multi-field packet classification while core routers only look at a packets destination address for forwarding. These edge routers usually operate at Gbps line speeds meaning that they only have a few ns to process each byte of a packet in order to inspect the entire packet content. This task becomes even more difficult for the routers at a network's core as line rates increase to 10-40 Gbps leaving only between 0.2-0.8 ns to process each byte of a packet.

The absence of an intrusion detection system at the edge or core of a network leaves it vulnerable to attacks due to the speed at which a virus or worm can spread. Slammer, the fastest spreading worm in history, infected over 75,000 hosts in only a 10-minute period [1] doubling in size every 8.5 seconds. The worm did not contain malicious content but was designed to overload a network, slowing down Internet speeds and even causing the loss of connection for some end-hosts. Another worm which caused mass damage by Denial of Service attacks was CodeRed, infecting 359,000 hosts in 14 hours [2]. With viruses/worms spreading at these speeds it would be unrealistic to expect the end-hosts of a network to update their systems to new threats due to the slow time it would take to react to the rapid attack. There is also the high cost in both the maintenance and lost work time due to updating the system.

The rules used for DPI in an intrusion detection system such as Snort [3] consist of two parts. The first part is a header rule which involves performing 5-tuple packet classification on a packet's header. The second part is a content rule where a specific string or strings must be searched for in a packet's payload at given locations. Research in [4] shows that, for Snort, the fraction of time that network intrusion detection spends finding these strings on real traces is between 40-70%, using 60-80% of the instructions executed. Based on these reasons, we decided to design a fixed string pattern matching hardware accelerator which would help the DPI needed for intrusion detection to be moved from the end-host to the edge or even core of a network improving network security. The hardware accelerator had to be designed with energy efficiency in mind, along with high throughput. This is because it is not simply good enough to throw more processing power at the problem of string matching without considering the power implications due to the already tight power budget on a router line card. Adding extra power usage to a router would mean the need for extra cooling and increased maintenance costs.

It is not possible to meet Gbps line speeds when implementing fixed string matching by increasing clock speeds alone. To reach speeds of 40 Gbps a hardware accelerator would need to run at 5 GHz assuming it could process each character from a packet in a single clock cycle. These speeds are not possible on current state-of-the-art FPGAs which typically run at speeds of around 500-600 MHz. Running a hardware accelerator at these speeds would also have massive power implications due to dynamic power consumption. To solve this problem we have designed a hardware accelerator which uses multiple string matching engines working in parallel on a single FPGA. These string matching engines use separate memory blocks to help overcome the memory bandwidth problem. For large rulesets containing many thousands of strings the search structures can be split across the memory of multiple engines with the engines working together to scan a packet. For rulesets containing fewer strings, the entire search structure can be placed on a single memory block, with the search engines working separately on individual packets, achieving maximum throughput.

For string matching algorithms using state machines such as Aho-Corasick [5] the majority of the memory is occupied by the pointers needed for making transitions between states. We reduce the number of pointers which needs to be stored for the Snort ruleset by up to 98.2% compared to the original Aho-Corasick algorithm. This is done by storing a small number of default pointers to the states which are most commonly pointed to in a small lookup table separate from main memory. These default pointers are used when a valid pointer stored in main memory is not found. Making a large decrease to the maximum number of stored pointers that a state may have also allows us to massively reduce the amount of logic needed for traversing the state machine, helping increase clock speeds. Unlike state machines which use fail pointers, we can guarantee a fixed throughput of 1 character per clock cycle for a packet being scanned. This prevents attacks being constructed which flood a system with packets it performs poorly on.

The rest of this paper is laid out as follows. In section II we review related work. The Aho-Corasick algorithm is explained in section III along with the modifications made to reduce memory consumption. We describe the memory organization in section IV along with the architecture of the complete hardware accelerator and string matching engine. Section V characterizes the strings used for testing and presents the performance in terms of memory consumption, throughput and power consumption. The paper is concluded in section VI.

II. Related Work

The area of fixed string matching is one of the best studied fields due to its many applications such as bibliographic search, word processing and use in Internet search engines. In recent times research has been concentrated on its use in the area of DPI for intrusion detection/prevention systems. Some of the first and best known algorithms in the area of fixed string matching include the Knuth-Morris-Pratt [6] and Boyer-Moore [7] methods, which work well for single string matching through the use of skip tables. Algorithms which work well for matching multiple strings include Aho-Corasick [5] and Commentz-Walter [8] using finite automata. There have also been many proposed algorithms seeking to improve on these approaches [9-12].

Two algorithms are presented in [13] based on the Aho-Corasick approach for string matching. They are designed with hardware acceleration in mind and reduce memory consumption through the use of bitmaps and path compression. Path compression combines together a series of successive states, each of which contains only a single pointer, in order to reduce the total number of states which need to be stored. Bitmaps are used to reduce the number of pointers at a state from its worst case of 256. Problems with the use of bitmaps are the large logic delay required to find a pointer, slowing down the performance of hardware implementation. Finding a pointer involves the checking and addition of the 256 bits contained within the bitmap. Both schemes also use fail pointers, meaning that they cannot guarantee the processing of a character on every clock cycle. Other modifications made to the Aho-Corasick algorithm that are targeted towards hardware implementation are presented in [14, 15].

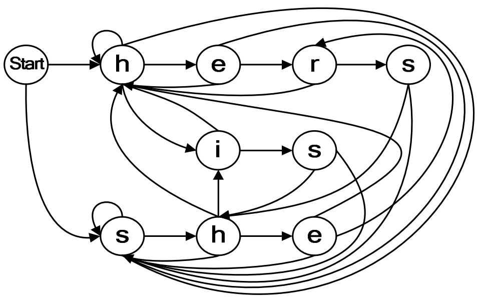

Figure 1. Aho-Corasick DFA

Another popular method for fixed string matching is through the use of TCAM [16-17]. A TCAM based multi-pattern matching scheme is presented in [18]. It can search for long strings by breaking them up before storing them in TCAM. This approach searches through the packet one byte at a time by looking at a set of strings equal to the TCAM width. It records all partial matches and their position to identify if a full match has taken place. They deal with issues such as optimum TCAM width and are able to search for correlated patterns and patterns with negations. The use of TCAM however means it is an expensive and power hungry approach.

III. Proposed Solution

A. *Aho-Corasick Algorithm*

Our algorithm is based on the Aho-Corasick approach for string matching. We therefore give an explanation of this algorithm so that our modifications can be better understood. The algorithm matches multiple strings using a Deterministic Finite state Automaton (DFA). The DFA has a start state which all strings to be matched are extended from. This state is the state where no strings have been partially matched. The strings to be matched extend from the start state 1 state per character. Strings sharing a common stem will also share a number of common states extending from the start state. To match a string against a text, the search begins at the start state and traverses from one state to another, based on transitions decided by the values of the input characters. A state's depth is the smallest number of transitions needed to reach it from the start state.

The original algorithm proposes two methods for calculating these transitions, with one solution using a failure function and the other a move function. The solution which uses the failure function requires the lowest amount of memory but cannot guarantee the processing of 1 input character per cycle. This is because in this solution each state stores only the transitions for characters whose next state is 1 level deeper than the current state. All other characters must follow a fail transition which will cause a wasted transition. Multiple fail transitions may have to be followed until the correct state is found, wasting many cycles. The approach which our work is based on uses the move function. In this approach each state stores the transitions for all states which could be transitioned to regardless of their depth. This means there is no need for a fail function and thus no wasted transitions, so that a new input character can be processed on each cycle. The disadvantage of this approach is that it needs to use larger amounts of memory to store all possible transitions.

Figure 1 shows a state machine constructed to find the strings (he, she, his, hers). In this state machine each state is represented by a circle with the value inside it indicating the character needed to traverse to it. The diagram shows all transitions stored at a state.

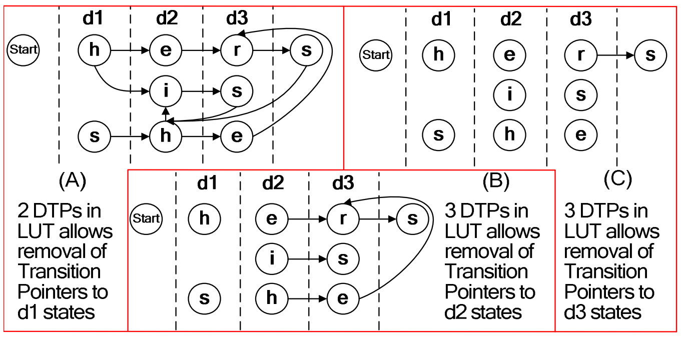
d1
d2
d3
Start
(A)
2 DTPs in LUT allows removal of Transition Pointers to d1 states
(B)
3 DTPs in LUT allows removal of Transition Pointers to d2 states
(C)
3 DTPs in LUT allows removal of Transition Pointers to d3 states

Figure 2. Reduced memory through use of Default Transition Pointers (DTP)

B. *Memory reduction*

The storage of transition pointers is the largest cause of memory usage when saving a state machine used for DPI. This is because each state has to store the 256 pointers needed to represent all possible character transitions unless some kind of memory compression scheme is used. Even only storing the pointers which point to a state other than the start state can lead to large memory usage. Our observation that allows for a reduction in memory usage is that the content varies widely between the strings used for DPI systems such as Snort. This means that the majority of transition pointers stored in states will point to only a few states near the start state. A large reduction in memory usage can be achieved by removing these same few replicated pointers from the states and placing them in a lookup table where they can be shared by all states. We reduce the number of transition pointers which needs to be stored at states by over 98% in the Snort ruleset used for testing by placing default transition pointers to the most commonly pointed to states at a depth of 1, 2 and 3 in the lookup table.

The maximum number of states which can occur at a given depth in the state machine is 256^d where d is the depth. This means that we can store a default transition pointer to all states at a depth of 1 in the state machine, as only 256 default transition pointers will need to be stored in the lookup table to cover all possible states at this depth. The default transition pointer for each character value will be a state number if a state with its value exists at this depth or the start state if it does not exist. Each state will then store 2 pieces of information for its transition pointers. The first is the state which must be transitioned to and the second is the value the input character must be to make this transition. An input character making a transition from one state to another will be compared to the values for all transitions stored at the current state. A match occurring at one of these values means transitioning to the state it points to. No match will mean transitioning to the state the default transition returned from the lookup table points to.

A large percentage of states will also contain transition pointers to states at a depth of 2 in the state machine because they are close to the start. Storing a default transition pointer to all possible states at this depth would not be memory efficient as 65,536 would need to be stored. We therefore store default transition pointers to only the 4 most commonly pointed to states for each character value at this depth. We found through testing of strings used in the Snort ruleset that 4 was the optimum value. Each default transition pointer to a state with a depth of 2 will need to store the state number it points to and the character value of the preceding state. States at a depth of 3 will be pointed to far less often than the states that precede it. However, through testing we found that significant memory savings can be made by saving 1 default transition pointer to the most commonly pointed to state for each character at this depth. This means storing the character value of the 2 states that precede it and its state number. Using depth 2 and 3 default transition pointers means a history of the previous 2 input characters will need to be recorded for comparison information.

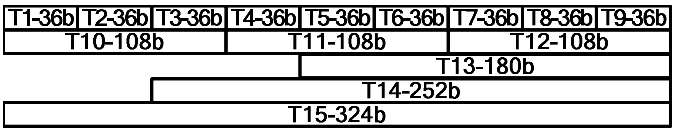

Figure 3. Position where each State Type (T) can be stored in a 324-bit wide memory word and its size in bits (b)

The process of transitioning between states now involves comparing the input character making a transition to the character values stored at the current state. As was the case before, a match at one of these values will mean transitioning to the state it points to. No match will mean looking at the information the input character returned from the lookup table. This will be information on the 6 possible default transitions. The previous 2 input characters recorded are compared to the preceding state character values for the depth 3 default transition. The previous input character is also compared to the 4 character values for the depth 2 default transition pointers. A match to the depth 3 character values will mean following its default transition pointer. No match will mean following one of the depth 2 default transition pointers if they contain a match. No match on either the depth 2 or 3 character values will mean following the depth 1 default transition which will be to the start state in the case where this state does not exist.

The diagrams in Figure 2 show how the use of default transitions can reduce the number of transitions which need to be stored at a state for the state machine shown in Figure 1. The average number of pointers which need to be stored at a state in Figure 1 is 2.5. Inserting default transitions to states with a depth of 1 reduces the average number of transitions which need to be stored at a state to 1.1 as shown in Figure 2(A). The insertion of default transitions to states at a depth of 2 reduces this number to 0.5 as shown in Figure 2(B). Finally inserting default transitions to states with a depth of 3 reduces the average number to 0.1 as shown in Figure 2(C).

IV. Memory Layout and Hardware Architecture

A. *Memory Layout*

The string matching engines have been designed to handle states with up to 13 transition pointers, which is adequate once the memory reduction techniques have been applied. To store this many pointers, 324-bit memory words are needed. A state contains on average less than 2 pointers, making it wasteful to store only 1 state in each memory word. For this reason we use 15 different state types. A state's type indicates how many pointers it has and its position in a memory word. State types 1-9 store states with 0-1 pointers, types 10-12 store states with 2-4 pointers, type 13 stores states with 5-7 pointers, type 14 stores states with 8-10 pointers and type 15 stores states with 11-13 pointers. Figure 3 shows where each state type can be stored in a memory word and its width. A state machine's states are carefully assigned a state type and memory word after it has been built to insure no gaps of unused memory. Each state contains 12 bits to indicate if it has any matching strings and if so the location of the string numbers in memory. The matching string numbers are stored in a memory block separate from the one used to store the state machine. This is so that we do not affect throughput when retrieving the matching string numbers. Each transition pointer stored at a state requires

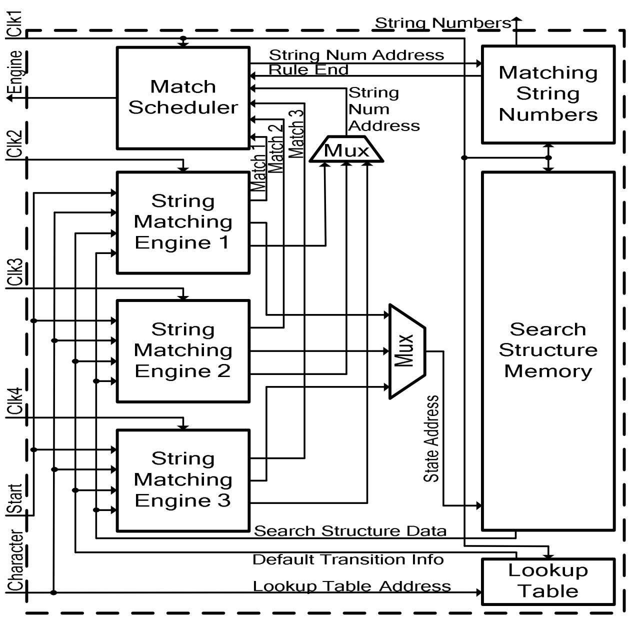

Figure 4. Architecture of String Matching Block

24 bits, with 8 bits used to store the character value needed to follow the pointer, 12 bits to store the memory address of the state being transitioned to and 4 bits to indicate its type.

B. *Architecture of String Matching Block*

In order to achieve a throughput in excess of 40 Gbps we use multiple string matching blocks on the same FPGA. For rulesets containing many thousands of rules these blocks can work in parallel on the same packets, with each block searching for a share of the strings. For smaller rulesets each block can work individually, searching for all strings in a packet to maximize throughput. Each block has 2,048 27-bit memory words to store the matching string numbers. Each of these memory words holds two 13-bit string numbers and 1 bit to indicate if all matching numbers have been outputted. A state with matching strings will point to the memory word where its matching string numbers are stored. These numbers will be outputted 2 at a time until a set bit indicates all numbers have been outputted. The lookup table which stores the default pointers uses 256 49-bit wide memory words. A default pointer does not need to store the address of the state it points to when used in the hardware accelerator. This is because each default pointer points to a fixed address in the memory used to store the state machine. Each default pointer to a state at a depth of 1 requires 1 bit to say if it points to a state at that depth or the start state. The default pointers to states with a depth of 2 require 8 bits to store the value of their preceding state, while the default pointers to states with a depth of 3 require 16 bits to store the value of their 2 preceding states.

In order to maximize throughput the memory used to store the matching string numbers, state machine and lookup table is true dual port. Each string matching block has 6 string matching engines with 3 engines sharing access to each port. Figure 4 shows half of 1 string matching block with the logic used to access the other port identical. Three string matching engines share each port as the maximum clock speed of each engine is slower than that of memory. The memory runs at a speed equal to 3 times that of a processing engine. Each engine sharing a port runs at the same clock speed, with the clock for each engine 120º out of phase with the previous. This allows for a simple memory interface with the read commands simply multiplexed together. A string matching blocks needs 6 packets

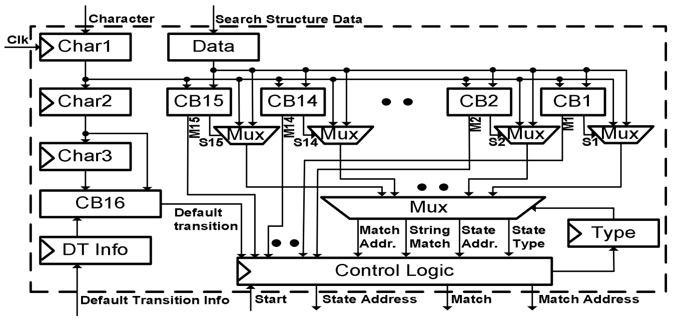

Figure 5. Architecture of String Matching Engine

to keep its engines busy. Each of the 6 string matching engines can process 1 byte from a packet per clock cycle. These engines run at one third the speed of memory so a string matching blocks throughput in bps will be 16*f_{max} where f_{max} is the maximum clock speed of its memory.

The characters from 3 packets being searched are multiplexed together and inputted through the same input port. The process of searching for matching strings in a packet works a follows. The first character or byte being searched is inputted into the string matching block with a start signal being set to indicate that it is the first character. This character will then look up its default transition information from the lookup table. This default transition information and the character will then be registered by the matching engine searching the packet. The state transitioned to will be determined by the default transition information because it is the first character, meaning that it can only transition to a state with a depth of 1 or the start state. This state information will be requested from the search structure memory. On the next clock cycle the string matching engine will register the next character from the packet, along with the default transition information this character will have returned from the lookup table. It will also register the state information which will have been requested from the search structure memory on the previous cycle. From this information it will then decide whether to make a transition stored at the state registered or a default transition. This process will continue until the end of the packet is reached.

A matching string will have been found when a state returned to a string matching engine contains a match bit which is set. This set bit and the accompanying memory location for the matching string number will be sent to the logic block labeled the match scheduler. This block will record the address and the engine number which recorded a match in a buffer. This buffer records matches for 3 engines. The address of the matching string numbers will be sent to the memory which saves these numbers when it gets to the front of the buffer. The match scheduler will output the number of the matching engine while the memory used for storing the numbers will use the address to output the matching string numbers. The match scheduler will keep incrementing the address until the memory storing the matching string numbers returns a set bit to indicate all matching string numbers stored at this state have been outputted. The match scheduler can then begin working on the next stored matching address which may be in its buffer.

C. *Architecture of String Matching Engine*

The architecture of a string matching engine can be seen in Figure 5. It consists of registers used to store the input character, previous 2 input characters, state information returned from search structure memory and the default transition information from the lookup table. There is also a register used to store the state type which will be analyzed.

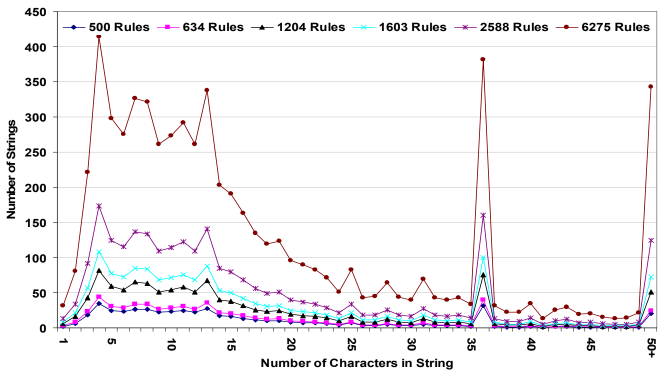

Figure 6. Distribution of string lengths for unique strings found

The state information and input character registered are fed into 15 comparator blocks with 1 comparator block for each state type. These comparator blocks are used to find out if a pointer from the state being analyzed should be taken. There is also a comparator block to decided which default pointer should be taken if the state being analyzed has no valid pointer.

V. Performance Analysis

A. *Strings used*

In order to test our method for memory reduction we used strings from the Snort ruleset. The Snort ruleset is generated from hand by skilled experts. They build rules based on known worms, viruses or other harmful activities. These rules are built by extracting unusual content from the payload and header information. As the number of known attacks increase so does the Snort ruleset. We chose to use the Snort ruleset as it is one of the most widely used in industry. We also chose it as it is one of the most difficult rulesets to implement fixed string matching on due to the many thousands of strings which need to be searched for. The Snort ruleset which we used contained 6,275 unique strings which need to be search for. The character distribution for these strings can be seen in Figure 6. It can be seen that the peak in the character distribution is between 4 and 13 bytes. In order to test our hardware accelerator on different sized rulesets we created 5 extra rulesets containing less strings. To do this we created a program which reduced the number of strings by randomly extracting strings while keeping the same character distribution. The character distribution for these strings can also be seen in Figure 6.

B. *FPGA implementation*

The hardware accelerator has been implemented in VHDL and targeted at Altera Cyclone EP3C120F484C7 and Stratix EP3SE260H780C2 FPGAs. Both FPGAs are built on Taiwan Semiconductor Manufacturing Company 65-nm process technology with the Cyclone 3 running at 1.2 Volts and the Stratix 3 at 1.1 Volts. The Stratix 3 implementation has been implemented with 6 string matching blocks, each using 3,584 memory words to store its state machine. Memory limitations has meant limiting the Cyclone 3 implementation to 4 string matching blocks, with each allowing 2,560 memory words to store its state machine. The architectures were synthesized using Altera Quartus II design software to obtain maximum clock speeds and resource utilization. Table I shows the memory and logic usage for the hardware accelerators along with the maximum clock speed of their memory.

C. *Transition pointer reduction*

The results in Table II show how our memory reduction techniques reduce the number of transition pointers which need

TABLE I. Resource Utilization

Device	Logic elements usage	Memory blocks	f_{max}
Cyclone 3	35,511/119,088	M9Ks 404/432	233.15 MHz
Stratix 3	69,585/254,400	M9Ks 822/864	460.19 MHz

TABLE II. Reduction in Transition Pointers

Strings	634	1603	2588	6275	500	1204	2588
Original Aho-Corasick							
States	11,796	29,155	46,301	109,467	9,329	22,026	46,301
Avg.Pointers	68.29	81.07	85.00	87.01	67.28	77.07	85.00
Our Method	Stratix 3				Cyclone 3		
Blocks	1	2	3	6	1	2	4
States	11,796	29,226	46,599	109,638	9,329	22,049	46,570
d1	68	97	108	110	67	83	125
Avg.Pointers	8.16	6.77	5.33	4.16	7.17	5.70	5.28
d1+d2	262	493	662	1,131	246	415	723
Avg.Pointers	3.43	2.68	2.09	1.92	2.87	2.21	2.20
d1+d2+d3	323	622	850	1,509	306	531	955
Avg.Pointers	2.39	2.01	1.9	1.54	2.09	1.88	1.18
Reduction	96.5%	97.5%	97.8%	98.2%	96.9%	97.6%	98.6%
Mem.(bytes)	148,259	296,967	445,641	838,298	105,599	214,141	429,656
Speed(Gbps)	44.2	22.1	14.7	7.4	14.9	7.5	3.7

to be stored at a state, and thus the memory consumption for the rulesets shown in Figure 6. It also shows the maximum throughput for a given number of rules using the Snort ruleset. By taking the ruleset with 634 strings as an example it can be seen that the original Aho-Corasick algorithm creates a state machine which stores an average of 68.29 transition pointers per state. This ruleset contains strings with 68 unique starting characters. This means that there will be 68 states at a depth of 1 in the state machine. Inserting default transitions to these states in our lookup table reduces the average number of transition pointers which need to be stored in a state to 8.16. We then add 194 default transition pointers to states at a depth of 2. This increases the total number of default pointers in the lookup table to 262 and reduces the average number of transition pointers which need to be stored at a state to 3.43. Finally, adding in 61 default transition pointers to states at a depth of 3 reduces the average number of transition pointers which need to be stored at a state to 2.39. This is a reduction of 96.5% compared to the unmodified algorithm.

The total memory consumption for the state machine, string numbers and lookup table is 148,259 bytes. This means that the hardware can achieve its maximum throughput of 44.2 Gbps, as each string matching block can fit the full state machine in its memory, enabling each block to search packets on its own. The ruleset with 1,603 strings, for example, will need to be split in 2 with a state machine for each group saved to a separate string matching block. This will mean a maximum throughput of 22.1 Gbps as 2 string matching blocks are required to search a packet. It can be seen that the memory consumption scales very well, as the number of strings grow when using our hardware accelerator. The number of bits needed to store each string actually decreases as the number of strings increase. This is because our hardware accelerator allows the strings to be broken up into multiple groups with the state machine for each group placed in a separate block.

D. *Power Consumption*

Post place and route simulations were carried out using the Quartus II PowerPlay Power Analyzer Tool with VCD files generated by ModelSim in order to measure the power consumed by the hardware accelerator. Figure 7 shows the power consumed by the hardware accelerator when

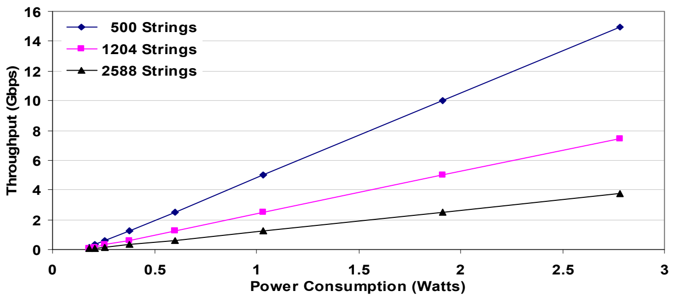

Figure 7. Power consumed by Cyclone 3 implementation

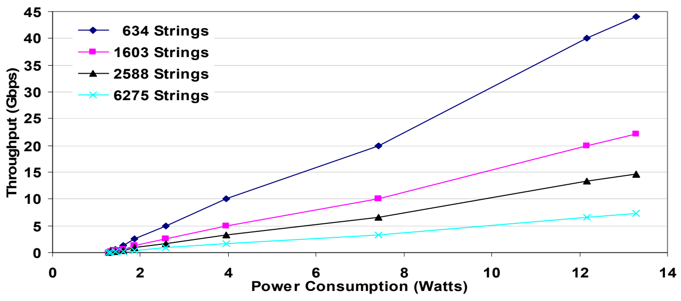

Figure 8. Power consumed by Startix 3 implementation

implemented on the Cyclone 3 FPGA. This graph was created by adjusting the clock speed of the hardware accelerator in order to measure the power consumption against throughput for the different sized rulesets used in testing. It can be seen that the hardware accelerator has a maximum power consumption of 2.78 Watts when searching through packets at its top speed.

The power consumption for the hardware accelerator implemented on a Startix 3 is shown in Figure 8. It has a maximum power consumption is 13.28 Watts when running at its top speed. It is worth noting that our hardware implementation only used the M9K block RAM on the FPGA and none of the M144K block RAM. This means that it is possible to double the memory available to the string matching blocks. This would allow the number of strings which could be searched to grow, as larger state machines could be saved. It would also allow throughputs to increase, as large sets of strings need to be split into fewer groups, meaning less string matching blocks would be needed to search each packet.

E. Performance Comparison

Table III shows how our hardware accelerator compares to the 2 algorithms presented in [13]. Their methods were tested using a set of strings from the Snort ruleset which contained 19,124 characters. For fair comparison we reduced the 6,275 strings from the Snort ruleset we used until it had 19,124 characters, while keeping the original character distribution using the method we described previously. It can be seen that our method shows a reduction of 20 times less memory needed to save the total data structure when compared to their bitmap compression technique. Our method also shows a reduction in the memory needed by a factor of 8 when compared to their path compression technique. A throughput increase is also obtained due to the fact that our method does not use fail pointers, and our simplified hardware architecture.

VI. CONCLUSION

In this paper we have shown that it is possible to implement the computationally heavy task of string matching at the line speed of a backbone network, with low power consumption. In our scheme, the Aho-Corasick algorithm with eliminated fail functions is employed to guarantee worst case performance. By introducing a number of default transitions, the memory consumption is greatly reduced so that the search structure can be easily packed into the on-chip memory of an FPGA. Furthermore, multiple searching engines are employed to search many packets in parallel improving throughput. Our approach also shows large improvements in memory consumption and throughput when compared to other hardware based approaches. In our future work, we will extend our architecture to more types of platforms such as ASIC.

TABLE III. PERFORMANCE COMPARISON

Approach	Device	Memory (bytes)	Throughput (Gbps)
Our method	Cyclone 3	138,470	7.5
Our method	Stratix 3	138,470	22.1
Bitmap[13]	ASIC	2,800,000	7.8
Path compression [13]	ASIC	1,100,000	7.8

ACKNOWLEGMENT

This work was funded by Enterprise Ireland and the Irish Research Council for Science, Engineering and Technology.

REFERENCES

[1] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. "Inside the slammer worm," in IEEE Security and Privacy, vol. 1, no. 4, Jul-Aug 2003, pp. 33-39.

[2] D. Moore, C. Shannon, and J. Brown, "Code-Red: A Case Study on The Spread and Victims of an Internet Worm," Proc. of the 2nd ACM Internet Measurement Workshop, ACM Press, 2002, pp. 273–284.

[3] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," Proc. of the 13th USENIX conference on System administration, Nov. 07-12, 1999, Seattle, Washington.

[4] S. Antonatos, K. G. Anagnostakis and E. P. Markatos. "Generating realistic workloads for network intrusion detection systems," SIGSOFT Softw. Eng. Notes 29, 1 (Jan. 2004), 207-215.

[5] A. V. Aho and M. J. Corasick, " Efficient string matching: an aid to bibliographic search," Commun. ACM 18, 6 Jun. 1975, 333-340.

[6] D.E. Knuth, J.H. Morris and V.R. Pratt, "Fast pattern matching in strings," SIAM Journal on Computing, vol. 6, no. 2, pp. 323-350, 1977.

[7] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Commun. ACM 20, 10 (Oct. 1977), 762-772.

[8] B. Commentz-Walter, "A string matching algorithm fast on the average," Proc. 6th International Colloquium on Automata, Languages, and Programming, pp. 118 132. (1979),

[9] J. J. Fan and K. Y. Su, "An efficient algorithm for matching multiple patterns," IEEE Trans. on Knowledge and Data Engineering, vol. 5, no. 2, pp. 339-351, 1993.

[10] U. Manber and S. Wu, "A fast algorithm for multi-pattern searching," in Tech.Report TR-94-17, CS Dept., University of Arizona, 1994.

[11] M. Fish and G. Verghese, "Fast content-based packet handling for intrusion detection," in UCSD Technical Report CS2001-0670, 2001.

[12] M. Crochemore and D. Perrin, "Two-way string-matching," J. ACM 38, 3 (Jul. 1991), 650-674.

[13] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. "Deterministic memory-efficient string matching algorithms for intrusion detection." In IEEE Infocom, Hong Kong, China, Mar. 2004.

[14] L. Tan and T. Sherwood 2005. "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," Proc. of the 32nd Annual international Symposium on Computer Architecture (June 2005), 112-122, Washington, DC

[15] S. Dharmapurikar and J. Lockwood, " Fast and scalable pattern matching for content filtering," Proc. of the 2005 ACM Symposium on Architecture For Networking and Communications Systems (October 2005), 183-192, New York, NY

[16] J. Sung, S. Kang, Y. Lee, T. Kwon, and B. Kim, "A Multi-gigabit Rate Deep Packet Inspection Algorithm using TCAM," IEEE Globecom, Nov. 2005, 453-457

[17] M. Alicherry, M. Muthuprasanna, V. Kumar, "High speed matching for network IDS/IPS", IEEE ICNP, pp. 187-196, 2006.

[18] F. Yu, R. Katz, and T. V. Lakshman. "Gigabit rate packet pattern-matching using TCAM." In IEEE International Conference on Network Protocols, Berlin, Germany, Oct. 2004.