J Comput Virol (2007) 3:125–134
DOI 10.1007/s11416-007-0047-z

EICAR 2007 BEST ACADEMIC PAPERS

Regular expression matching with input compression: a hardware design for use within network intrusion detection systems

Gerald Tripp

Received: 12 January 2007 / Accepted: 19 March 2007 / Published online: 11 April 2007
© Springer-Verlag France 2007

Abstract This paper describes an optimised finite state automata based hardware design for implementing high speed regular expression matching. Automata based implementations of regular expression matching can become quite complex and if table driven can use large amounts of memory—this can be a problem for hardware based implementations, as the amount of memory available within standard Field Programmable Gate Array (FPGA) components can be quite small as compared with the amount of resources we expect to find within a software environment. This work uses an existing 'packed array' style of table based automata implementation, but then adds a form of input compression to group together characters that are treated identically by the automata. A hardware design for such a system has been created for use within a Xilinx Field Programmable Gate Array and tested by simulation. The design operates at a fixed scan rate of 2.0 Gbps independent of the regular expression used or the input data being scanned. The regular expression rules are first compiled by software and then loaded into the design at run time and may be updated dynamically without modification to the design.

1 Introduction

Network intrusion detection consists of monitoring computer networks for various forms of attack. This may be on an individual computer system (host based) that just monitors the traffic arriving at that machine or it could be looking at all of the traffic on a central part of the network (network based) for attacks that target any networked computer. The network based intrusion detection systems are more difficult to build because of the high data rates these may have to deal with, but are very useful as they are able to be used to help protect any machine on the local network, even if it is not capable of running such software itself—such as a networked peripheral like a printer.

The first line of defence is usually provided by one or more firewalls; these examine the headers of network packets and can allow traffic to proceed (or be dropped) on the basis of the IP addresses, port addresses and other header fields. This filtering is usually dynamic, and may be updated on the basis of outgoing TCP connections, for example.

Intrusion detection systems go further than a basic firewall, in that they look inside the contents of the packets as well as the header fields. This is more complex, as we may now not have any particular location within the packet to inspect and we may have to scan the packet's entire contents for the data items we are looking for. The most well known intrusion detection system is probably Snort [14]. This operates by using a set of intrusion detection rules—the first part will check for packets on the basis of the header fields and then we may look inside selected packets for various 'content' strings.

The content strings are a fixed set of bytes that we will need to search the packet for; the only variation in the string that is allowed is that we can chose to ignore the case of letters if we wish. A problem here is that there may be minor variations in input data that will cause the string not to match, such as the addition of a single space character. This could lead to us needing to look for multiple content strings to match all possible variants of the input data. To deal with this, the

Gerald Tripp is a Lecturer in Computer Science at the University of Kent.

G. Tripp (✉)
The Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, UK
e-mail: G.E.W.Tripp@kent.ac.uk

Snort system also allows the use of regular expressions to perform matching. These are useful as we can use a single regular expression to match multiple variants of the pattern that we may be looking for. Regular expression matching can, however, be more complex to implement—particularly if we need to build a hardware implementation rather than software. The requirement for needing regular expressions has increased over the last few years and a large number of Snort rules now use these for matching, often as well as using content strings.

A lot of intrusion detection systems are implemented in software. This can be fine for host based systems (so long as we don't choose to have too many rules!), but with network based intrusion detection systems this can be a problem under high network loads, particularly at central points within large high speed networks. A solution to this problem can be to use hardware based intrusion detection systems, or at least provide some form of hardware support. A lot of existing hardware based systems have targeted the problem of fixed string matching as this is less complex. Other systems have implemented regular expression matching, but they usually fix the rule set at the time the hardware design is 'synthesised'.

This paper looks at how we can implement the regular expression matching part of intrusion detection and describes how a regular expression matching system can be designed for implementation within a Field Programmable Gate Array (FPGA). The method used is to build a table based automata implementation but to use a form of input compression that groups together input characters that are treated in the same way by the automata. The table based approach allows the system to be dynamically updated at run time to allow for changes in the regular expressions being matched; the input compression helps to make significant reductions in the automata memory requirements.

The next section looks at some of the background and related work both in the hardware based regular expression and string matching fields. Section 3 describes the automata implementation mechanism and introduces the compression scheme. The results section gives details of the memory requirements for standard Snort regular expressions and describes a hardware design for a regular expression matching engine that is targeted at an FPGA. The final section gives conclusions and ideas for further work.

2 Background and related work

The most well known software intrusion detection system is probably *Snort* [14]. Many improvements have been made to this over the years, particularly introducing schemes to optimise the order in which we check data. A paper by Kruegel and Toth [12] implements a modified Snort rule engine, which uses decision trees to reduce the number of comparisons made against incoming network data.

Abbes et al. [1] use decision trees along with protocol analysis; this allows comparisons to be made against particular fields within packets at different protocol layers, which reduces the numbers of false positives as compared to a system that blindly searches an entire network data packet or uses simple offset and depths constraints.

2.1 String matching systems

A number of pattern matching systems just allow us to search for a string. This is important, as matching a string of bytes is usually easier than matching a complex pattern (or regular expression) of a similar length, and this has implications on the number of patterns that may be searched for. One of the most efficient systems is the use of Bloom filters—these allow a very large number of strings to be searched for in parallel, but do suffer from false positives. Work by Attig and Lockwood [2] show that Bloom filters can be a very efficient front end system that will remove the large majority of innocent network traffic—this is then followed by a back end system that identifies the packets that are actually threats. This system has the advantage that the back end filtering system is not presented with a high load so may be implemented as a conventional software based intrusion detection system.

Baker and Prasanna [3] use a pipelined approach whereby they use a set of comparators to identify the presence of data bytes that are of interest. The outputs of the comparators are each fed into a chain of flip-flops, which form a pipeline. A string can be identified as being present if all of its bytes are identified as being present in the correct order in the comparator output pipelines. They show that this scheme can be extended to operate with n-byte input data, by having a set of comparators and pipelines for each input byte and then looking for all n byte alignments of the string across these n sets of pipelines.

Sugawara et al. [16] and Tripp [19] use finite state automata approaches to string matching. These first compress multi-byte input data into small tokens that represent a group of characters and then feed these into finite state automata that record how much of the string or strings have been matched.

2.2 Pattern matching

Most true pattern matching systems use a search pattern that is defined as a regular expression. The regular expression (RE) allows fixed values and also allows various amounts of choice and repetition. The basic operations are as follows:

– Concatenation: abc means “a” then “b” then “c”.
– Alternatives: abc|def means “abc” or “def”.
– Kleene star: a* means either: “”, “a”, “aa”, “aaa”, “aaaa” etc.

Brackets can also be used to group parts of expressions together when required. In practice many other operators are used, particularly to give ways of defining multiple values for a single byte and for various numbers of repeats. Snort uses a regular expression standard called the Perl Compatible Regular Expression (PCRE) [9]. A few of the other operators that will be referred to later are shown below:

– $[abc] = a|b|c$.
– $[d-g] = d|e|f|g$.
– `[^abc]` = any character other than a, b or c.
– {x,y} = repeat previous expression a minimum of x and a maximum of y times; where x and y are integers and either x or y (but not both) are optional.

To implement a RE matching system as an automata, we may go through many stages of processing. The first stage is to convert the RE into a Non-deterministic Finite Automata (NFA) usually using Thompson’s algorithm [18]. The normal procedure here is to break the RE down into single operations and then to convert each of these into a small NFA fragment. The pieces of NFA are built up on a stack and operators applied to the top elements of the stack to collect pieces and join these together. The NFA is unusual in that it may have more than one node active at any time, and there can be multiple edges leaving a node that are enabled on the same input data item. As well as this we can have ε-transitions, which are edges taking us from one node to another, but which do not consume any input data.

As an example, an NFA representation of the regular expression “$a(b|c)d$” is shown in Fig. 1, where node 0 is the initial node and node 7 is the terminal node that indicates a match.

2.3 Non deterministic finite automata implementation

The NFA is not that efficient to implement in software, as the current state of the NFA will be the set of nodes that are currently active. Processing input data then consists of looking at each of these nodes to see what edges are enabled and hence which set of nodes will next be active. Sidhu and Prasanna [15] have shown that an NFA can be implemented quite efficiently in hardware—as each node can be implemented as a flip-flop and the edges can be implemented as comparators, logic and routing resources to link the nodes together. This mechanism was used by Franklin et al. [7] to implement the Snort rule set, which was first converted into a single large RE. Several papers have followed on from this, making significant improvements in resource utilisation and performance. This earlier work used a comparator per NFA edge, and this required routing resources to take the input data to lots of distributed comparators—there typically being large numbers of comparisons against the same input byte values. Clark and Schimmel [5] improved on this system by replacing the distributed comparators with a global 8–256 decoder that generates 256 logic signals that identifies the presence or absence of each data byte value on the input. The appropriate logic values (rather than the data) are then distributed to the edge logic. Further advances to incorporate multi-byte matching have been made by Sutton [17] and by Clark and Schimmel [6].

The approaches above are very efficient in terms of logic resources, but have the disadvantage of a certain amount of inflexibility. The NFAs are implemented as interconnected pieces of logic and flip-flops—changes to the set of REs will therefore require changes to the logic inside the FPGA. In the field this would be likely to be made by rebuilding the logic we need and then performing a partial (or full) reconfiguration of the FPGA.

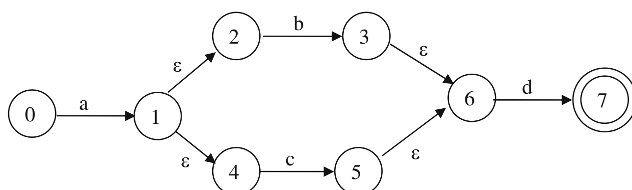

Fig. 1 Example NFA for “$a(b|c)d$”

2.4 Deterministic finite automata implementation

We can convert our NFA into a Deterministic Finite Automata (DFA) using a standard set of operations [11] which are outlined in the remainder of this paragraph. The DFA can only have one node active at a time, and can only one edge leaving any particular node on a given input character value. The DFA also does not have any ε-transitions. The DFA is simpler to implement as its state consists of which single node is currently active. The DFA can be implemented as a piece of sequential logic in various ways or can be based on one or more lookup tables. There are two stages in converting an NFA into a DFA. The first is referred to as ‘subset construction’ and consists of creating nodes for the DFA that each represents a set of nodes that may be active in the NFA; edges are then added to the DFA to correspond to the transitions made within the NFA. The resulting DFA may have large numbers of equivalent nodes, and a ‘DFA minimisation’ stage is needed to combine these together.

As an example, Fig. 2 shows an example DFA implementation of the regular expression “$a(b|c)d$”. This has been modified to allow matching to restart part way through a

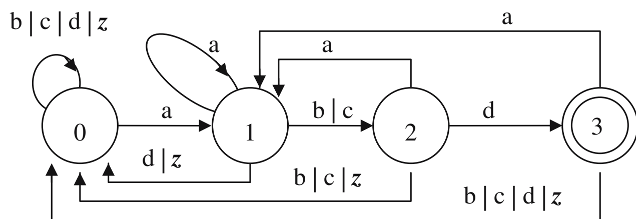

Fig. 2 Example DFA for "$a(b|c)d$"

failed match and has been minimised to remove redundant nodes. The character z is used here to refer to all other characters (if any) other than those covered by the regular expression. All edges are shown including those leading to the initial node. A DFA approach to hardware based RE matching was used by Moscola et al. [13] who converted Spam Assassin rules into DFAs and then output these in a VHDL finite state automata format, which would then be synthesised and built into an FPGA design.

3 Regular expression implementation

Much of the existing work on hardware based regular expression implementation has converted the RE into a piece of logic for implementation within an FPGA. A problem with this approach is that changes to the REs require changes to the FPGA design. An alternative approach is to use a table based implementation; with careful design we can build systems that can be configured at run time by simply writing new data to the table. However, simple table based implementations can require quite a lot of memory resources. One problem is caused by the REs themselves that often create automata with more nodes (and edges) than might be expected; a second problem is that the tables required for implementing automata can have a high level of redundancy.

A basic table based implementation would require a two dimensional array (the 'state transition table'), with the current state (representing the currently active node) as one index and the current input value as the other. We use these to index into the table to give us a value for the next state and any output data. There may be redundancy in the table as we will typically only be interested in a subset of possible input values in any state, the others probably taking us to the initial state or the 'first character matched' state of the automata.

Assuming we implement this as a piece of memory, with each index used as part of the address bus input, then the amount of memory M in bits for a DFA with s states, i input bits and o output bits will be as shown in Eq. 1.

$$M = \left(\lceil \log_2 s \rceil + o\right) . 2^{i+\lceil \log_2 s \rceil} \tag{1}$$

The simple version based on a two dimensional state transition table will be referred to as the 'baseline' design, and any new implementation should aim to improve on this.

3.1 Packed array DFA implementation

From the work described by Sugawara et al. [16] we can see that efficient implementation of DFAs for fixed string matching can be performed by using a hardware based 'packed array' implementation. This uses a technique known as 'row displacement' which is commonly used in compiler lexical analyzers—a good review of these techniques is given by Grune et al. in [8]. The particular style used by Sugawara is referred to as row displacement with state marking, and this is explained below.

With this type of design we first generate a default array, which contains the default next state for each input value—this is typically equivalent to the next state when in state 0. We then create a difference array which contains the differences between the state transition table and the default array. We can use the default array and the difference array together to find the next state: we first look in the difference array for the next state; if there is no entry in the difference array then we use the value from the default array for that input character. A simple example of a state transition table and its associated difference and default arrays is shown in Table 1—as with the example in Sect. 2.4, the character z represents any characters other than those covered by the RE.

The difference array is typically very sparse and we can take advantage of this by using a more compact method of storage. We can treat the difference array as a set of 'state vectors' and pack the state vectors together (overlapping) into a one dimensional 'packed array'—this is done carefully so as to avoid any collisions between entries in different state vectors. This is performed as a search operation; we will try increasing values of the start position of each state vector until we find one that does not cause collisions with any of the existing data. Finding the best way of packing the state vectors into the table is an NP-complete problem and heuristics are often used as a simple way to find solutions that are 'sub-optimal' [8].

The entries in the packed array are tagged with state that they belong to, with (−1) being used to show that the table

Table 1 State transition table, default and difference arrays for RE "$a(b|c)d$"

State transition table

Current state \ Input	z	a	b	c	d
0	0	1	0	0	0
1	0	1	2	2	0
2	0	1	0	0	3
3	0	1	0	0	0

Difference array

Current state \ Input	z	a	b	c	d
0					
1			2	2	
2					3
3					

Default array

z	a	b	c	d
0	1	0	0	0

entry is unused. We can find an entry by first finding the start of the state vector for the current state and then indexing from this position with the input value. If the entry we fetch has the same tag value as the current state, then we have found a valid entry for that combination of current state and input, if not then there is no valid entry in the packed array and we fetch the next state for the input value from the default array.

The indexing system used by Sugawara et al. was a simple addition operation; this was modified by Tripp [20] who substituted the 'add-based' indexing for a faster bitwise exclusive or operation.

In practice we also have a base address (BA) entry in each of the default and packed arrays that act as a look-ahead operation to define the start address of the state vector for the next state (NS) value—thus avoiding the need to have a separate table that stores the BA for each state. A very simple example is shown in Table 2 for the regular expression "$a(b|c)d$"; in practice this particular example has all state vectors at offset 0. More details of the original implementation are described by Sugawara et al. in [16]; the bitwise exclusive or based indexing variation is described by Tripp in [20]. We would hope that this packed array implementation would give a reduction in the memory requirements, as compared with the baseline model. A difference however is that the automata for REs are often far more complex than those for fixed string matching.

3.2 Input compression

The fixed string matching algorithms by Sugawara et al. [16] and Tripp [19] use input compression to reduce the redundancy in the input data from multi-byte input words, so as to generate a smaller input word size to the finite state automata. If we build a system for matching regular expressions, we potentially have a very different type of automata design to consider. Here we may have multiple characters that take us from one state to another, we can think of this as being multiple edges between states—or as an edge that is enabled on a set of different input values. A problem now is that our difference array may no longer be sparse, as there may be many different entries for each current state that take us to the same next state on different input values. We should be able to reduce the number of different input values using compression—however, simple schemes such as compressing each input character of interest into a different value will often give little or no improvement because expressions such as "a[^a]a" will force all characters into the 'alphabet' of characters we are interested in.

Table 2 Default and Packed arrays for RE "$a(b|c)d$"

Default array						Packed array					
	Input						Input				
	z	a	b	c	d		z	a	b	c	d
NS	0	1	0	0	0	NS			2	2	3
BA	0	0	0	0	0	BA			0	0	0
						TAG	−1	−1	1	1	2

Instead, we compress input characters to the same value if they are treated in the same way by the automata. This is more complex than it sounds as there may be different overlapping groups of characters that are treated differently by different parts of the DFA. To be able to compress our input data, we need to form the smallest set of disjoint sets of input characters. We then compress the input value by identifying which set it belongs to.

Given a DFA $M = (S, I, O, \delta, \lambda)$, where: S is the set of states, I is the set of inputs, O is the set of outputs, δ is the state transition function and λ is the output function.

We obtain details of the sets of characters enabling edges of the DFA from the state transition function δ. If we receive input value i in current state s then the next state is given by $\delta(s, i)$.

We define E_{sn} as the set of characters enabling the edge or edges between current state s and next state n. This is shown in Eq. 2.

$$E_{sn} = \{i \in I | \delta(s, i) = n\}. \tag{2}$$

We then determine the complete set of edge sets P_a as shown in Eq. 3.

$$P_a = \{E_{sn} | s, n \in S\}. \tag{3}$$

We now have a set P_a that gives us the sets of characters that we are interested in for all DFA edges. These sets may however have overlaps, which will not enable us to define an input value as belonging to a single set. We now create a new set P_d, which is a set of disjoint sets of input characters, and such that each member of P_a can be created by either a single member of P_d or the union of two or more members of P_d.

We create the new set P_d as shown below, treating P_a as a list that we can modify at will:

Repeat the following sequence whilst $|P_a| > 0$:

- Take the head element h from set P_a.
- If there is no overlap between h and any other member of P_a, then move h to P_d

  Otherwise, we search through P_a for the first member S_0, such that $h \cap S_0 \neq \emptyset$, we then:
  - ∗ Remove h and S_0 from P_a.
  - ∗ Add sets: $h \cap S_0$, $h - S_0$ and $S_0 - h$ to the end of P_a.

We only add a new character set to P_d when we can see that the set has no overlap with any other set, so at the end P_d will contain a set of disjoint character sets.

From P_d we can create a simple compression table that maps any input character into a numeric value k that represents which member of P_d the character belongs to. The DFA can then be modified to replace each edge or edges between a pair of nodes with a new edge that is enabled on the set of values of k that represents the members of P_d that enable that edge.

3.3 Example

As a very simple example, we have the following regular expression:

```
``g[e-m][j-s][n-w]x''
```

This gives the sets shown in Eqs. 4 and 5.

$$P_a = \{\{z\}, \{g\}, \{e, f, g, h, i, j, k, l, m\}, \\ \{j, k, l, m, n, o, p, q, r, s\}, \\ \{n, o, p, q, r, s, t, u, v, w\}, \{x\}\} \quad (4)$$

$$P_d = \{\{z\}, \{e, f, h, i\}, \{g\}, \{j, k, l, m\}, \{n, o, p, q, r, s\}, \\ \{t, u, v, w\}, \{x\}\} \quad (5)$$

The characters contained in each set for this example along with the set numbers and values of each character (in hexadecimal) are shown in Table 3. Using the information in Table 3, we can generate a 256 element lookup table to perform the compression. The table is created by taking each of the possible characters as an index into the table and storing its set number at that position in the table. The compression table for this example is shown in Table 4.

Table 3 Disjoint set membership and character values

Set Number	Set Members: character with (value in hex).
0	z = all characters **not** listed below
1	e(0x65), f(0x66), h(0x68), i(0x69)
2	g(0x67)
3	j(0x6A), k(0x6B), l(0x6C), m(0x6D)
4	n(0x6E), o(0x6F), p(0x70), q(0x71), r(0x72), s(0x73)
5	t(0x74), u(0x75), v(0x76), w(0x77)
6	x(0x78)

Table 4 Input compression table

Most significant 4-bits of input (in hex)	Least significant 4-bits of input (in hex)															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	1	2	1	1	3	3	3	3	4	4
7	4	4	4	4	5	5	5	5	6	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3.4 Memory minimisation

In computer systems we have seen the amount of memory available rise by several orders of magnitude over the years; so for software the amount of memory used is not too much of an issue. What we have are large blocks of heavily pipelined memory that are accessed via one or more levels of processor cache. With custom hardware designs we often seek to obtain high throughput by having multiple instances of individual components that operate in parallel. If these each need access to memory then this could be provided via normal memory components, but this can lead to bottlenecks in the interface to that memory which may not be able to support the total aggregated bandwidth requirements. There is also an issue that the hardware component's performance may be affected by the latency in obtaining items from memory; this is even more of an issue if multiple components were contending for access to the same block of memory.

FPGAs normally have multiple blocks of internal memory. These blocks of memory are independent and can be closely coupled with individual components. Access to the memory within the FPGA is also very fast, with data typically being accessible within a single clock cycle. The problem, however, is that the amount of memory available within FPGAs is quite small when compared with the amounts we are used to in computer systems. For example: the Xilinx Virtex 4 FPGA [21] with the largest amount of memory (the XC4VFX140) has a total of 9,936 Kbits in 18 Kbit blocks and 987 Kbits in 16 bit blocks; over two orders of magnitude less than the memory available in a cheap desktop PC. Memory minimisation is therefore an important issue when designing systems targeted at FPGAs, both to maximise the amount

that can be implemented within a given FPGA and to enable the smallest (and cheapest) FPGA to be used for a particular commercial product. The next section shows the effect of the input compression scheme described in Section 3.1 on the memory requirements for a 'packed array' based regular expression matching system and describes a VHDL design targeted at an FPGA.

4 Results

Software was written to take a regular expression and to create a minimised DFA, using standard techniques as outlined in Sects. 2.2 and 2.4 above. The software was then modified to enable the creation of data for packed array automata as described in Sect. 3.1, with no input compression. As test data, regular expressions were extracted from the 22 Nov 2006 'CURRENT' version of the Snort community rule set. These rules were sorted to create a set of 340 unique regular expressions. From this basic set, 319 were chosen, which are all of the regular expressions apart from 21 that used the $\{x, y\}$ PCRE repeat operator that was not implemented as part of the regular expression processing software. Each regular expression was processed by the software to create the required sets of data for use within a finite automata implementation and the memory requirements noted in each case. The distribution of memory requirements for this first set of measurements is shown in Fig. 3b with the requirements for the baseline design shown in Fig. 3a for comparison. The graphs in Figs. 3 and 4 have a logarithmic x-axis and show the memory requirements in bands of increasing powers of 2. As an example, the value 8 on the x-axis refers to all memory sizes m in the range: 4 Kbits $< m \leq 8$ Kbits. The average memory use per RE for the first experiment is 80% of the memory required for the baseline version. This is as expected because of the complexity of some of the REs in the snort rule set; much of the gain of the packed array approach is offset by the greater implementation complexity. The software was then modified to use the input compression scheme described in Sect. 3.2 and then run again to measure the memory requirements when the input compression scheme was used. The results for this are shown in Fig. 4. This time we can see that the large majority of the REs now have a memory requirement that is 8 Kbit or less. The average memory requirements per RE is now 17% of the memory needed for the baseline version. The average memory requirements data is given in Table 5.

4.1 Hardware design

An example hardware design was written in VHDL for a single regular expression matching engine with a maximum of 255 states and a packed array with a total of 256 words. This is suitable for implementing a single rule from 95% of the 319 different regular expressions processed. The design is targeted at a Xilinx XC4VLX25 [21] FPGA and is implemented in a single 18 Kbit Block RAM primitive (out of which 12.5 Kbits are actually used)—the two memory ports provided in the Block RAM primitive allow this to be treated as two separate blocks of memory, one of which is used for the

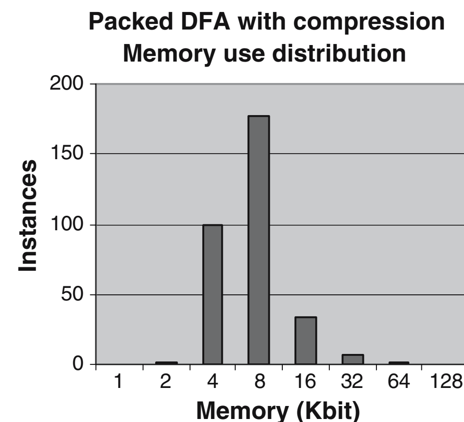

Fig. 4 Distribution of memory use per RE for packed array implementation with input compression

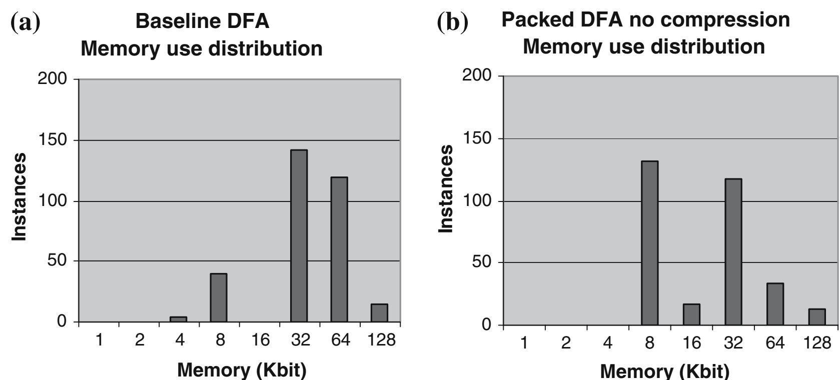

Fig. 3 Distribution of memory use per RE for **a** Baseline model, **b** Packed array implementation with no input compression

Table 5 Average memory use per regular expression

DFA Implementation	Average memory used (bytes)	Memory used as a proportion of that used for the baseline design (%)
Baseline design.	4,194	100
Packed array with no input compression	3,343	80
Packed array with input compression	702	17

packed array and the other of which is used as a single table that merges together both the input compression table and the default array. Pipeline registers are used in various places to provide the correct data alignment or to improve the clock speed. A boot port is provided to allow the table information to be written to memory, and this can be used whilst holding the reset input to the DFA active. There is only a single match output and this is also generated by these two blocks of memory as a look-ahead operation. A schematic of the design is shown in Fig. 5. The VHDL design was tested by selecting a number of different regular expressions to check different parts of the hardware and software implementation and then testing the design via simulation with large amounts of artificially generated input data to determine that the regular expression matching operates correctly. The design was also simulated 'post place and route' to check the timing of the resulting FPGA design. In addition, to look at a real life example, the design was simulated using a day's web logs from a departmental web server as input—a regular expression being used to identify the number of HTTP GET requests for one of the author's technical reports. The simulations performed correctly in all cases.

The resource requirements of the FPGA build are as follows:

Target device: XC4VLX25-12SF363

– 1 Block RAM component (out of a total of 72).
– 35 logic slices (out of a total of 10,752).

The logic resource utilisation is trivial, and the number of instances of the design that can be implemented in a single chip is limited by the amount of Block RAM resources available. The largest device (in terms of Block RAM) in the Xilinx Virtex4 series (XC4VFX140) has a total of 552 Block RAM primitives. We would expect to be able to use such a device to provide over 500 regular expression engines running in parallel, and still leave memory resources for other operations.

The regular expression matching engine could be created with a number of different memory sizes if required. If larger DFAs were used for the remaining 15 REs that wouldn't fit in the example implementation then we can calculate that 326 Block RAMs would be required in total for all 319 regular expressions that were used for these measurements.

The Xilinx tools were used to measure the timing of the design created, and these report a minimum clock period of 3.969 ns. The system is designed to operate at a deterministic input rate of one input word per clock cycle. At an input word size of 8-bits we therefore have a scan rate of 2.0 Gbps—this is independent of the regular expression used or the input data being scanned.

4.2 Related work

Work to implement regular expressions using the algorithm from Sugawara et al. has also been done by Hickman [10].

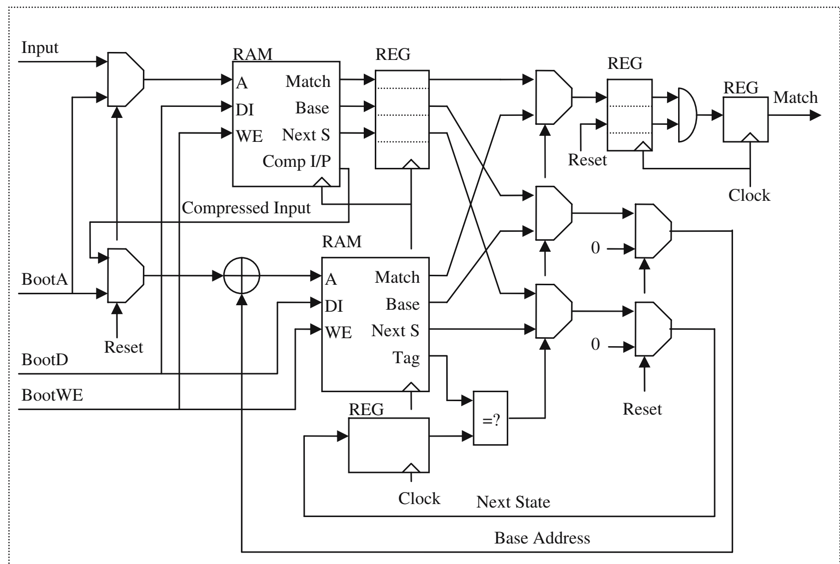

Fig. 5 Schematic of regular expression engine implementation–derived from [16]

Hickman's work takes a regular expression and creates first an NFA and then a non minimised DFA, the operation of which is simulated in Java.

The work described in the current paper goes further than the work by Hickman, as it: performs minimisation of the DFA; it uses the idea of using disjoint sets of characters to perform input compression; creates a VHDL design for implementation within an FPGA and tests this via simulation. As seen in Table 5, the addition of the compression stage make a major impact on resource utilisation, using 17% of the memory for the baseline design, as compared with 80% for an implementation just using the basic packed array automata implementation.

High speed regular expression matching has also been done by Brodie et al. [4]. They compress their state transition table by the use of run length encoding and have a fast system to decompress this during operation. Their system handles multiple bytes per FSM cycle by the use of an input compression system that creates an Equivalence Class Identifier (ECI)—this classifies the input data into one of a number of sets of input data patterns, where each member of a given set would have the same effect on the FSM. This has similarities to the multi-byte input classification systems used by Sugawara et al. [16] and Tripp [19] for fixed string matching, although the system used by Brodie et al. is more complex in that it deals with sets of different combinations of characters rather than the fixed character groups with leading or trailing wild cards characters that were considered by Sugawara et al. and by Tripp.

The system by Brodie et al. was implemented as a FPGA based system with a 32-bit input and operates at 4 Gbps with a memory use of 96 Kbits per engine. (This compares to 2.0 Gbps and a memory use of 18 Kbits per engine in this current paper.) They calculate that if their system was implemented in an Application Specific Integrated Circuit (ASIC), then they should be able to achieve speeds of 16 Gbps, with a 500 MHz clock rate.

5 Conclusion and future works

This regular expression matching system uses a DFA implementation based on the design by Sugawara et al. [16], but modifies this to use a different form of input compression which classifies input data depending on the way in which the characters enable edges of the automata. This compression is important, even for a system with a single byte wide input, because automata for implementing regular expressions commonly have edges that are enabled on multiple characters; as compared to fixed string matching for a single string where apart from failure paths there is generally a simple route from initial to terminal states. A set of regular expressions from the Snort community rule set have been processed, and the algorithm has an average memory resource utilisation of 17% of a simple baseline design as compared with 80% when the input compression is not used.

This work has produced a VHDL model of a high speed regular expression matching engine for implementation within an FPGA. The design has been tested by simulation, both at the behavioural level and 'post place and route". The design is capable of running at a scan rate of 2.0 Gbps independent of the regular expression being scanned for or the input data that is being scanned.

The current design could be expanded to enable the rule processing software to handle the $\{x, y\}$ repeat operator that is used in PCRE; we would then need to examine the resources required by the rules that use this operation. It may be that some rules would require large amounts of resources for their implementation because of the likely DFA complexity caused by using expressions such as `[^\n]{1000,}` which are just searching for long lines of text without line breaks.

A new challenge will be to look at the effect on resource utilisation as we increase the input word size. There may be problems when moving to large word sizes because of the potential problem of having a very large number of input symbols to deal with—however it will be interesting to find out at what point this becomes a problem. Another area to look at is how the system could be modified to operate with multiple REs per matching engine. The current system could be modified quite easily to allow multiple match outputs from each regular expression engine and it would be interesting to see what density of REs we could obtain per engine, possibly with careful grouping of REs.

References

1. Abbes, T., Bouhoula, A., Rusinowitch, M.: Protocol analysis in intrusion detection using decision tree. In: Proceedings of International Conference on Information Technology: Coding and Computing (ITCC'04), vol 1, pp 404–408. Las Vegas, Nevada (2004)
2. Attig, M., Lockwood, J.W.: SIFT: snort intrusion filter for TCP. In: Proceedings of IEEE Symposium on High Performance Interconnects (Hot Interconnects-13). Stanford, California (2005)
3. Baker, Z.K., Prasanna, V.K.: A methodology for synthesis of efficient intrusion detection systems on FPGAs. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04). Napa, California (2004)
4. Brodie, B., Cytron, R., Taylor, D.: A scaleable architecture for high-throughput regular-expression pattern matching. In: Proceedings of 33rd Annual International Symposium on Computer Architecture (ISCA 2006), Boston, pp 191–202 (2006)
5. Clark, C., Schimmel, D.: Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In: Proceedings of Field Programmable Logic and Applications, 13th International Conference (FPL 2003), Lecture Notes In Computer Science, LNCS 2778, pp 956–959. Springer, Heidelberg (2003)

6. Clark, C., Schimmel, D.: Scalable multi-pattern matching on high-speed networks. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04). Napa, California (2004)
7. Franklin, R., Carver, D., Hutchings, B.L.: Assisting network intrusion detection with reconfigurable hardware. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '02), pp 111–120. Napa, California (2002)
8. Grune, D., Bal, H.E., Jacobs, C.J.H., Langendoen, K.G.: Modern Compiler Design. Wiley, Chichester (2000)
9. Hazel, P.: PCRE - Perl-compatible regular expressions. Retrieved 11 January 2007 from http://www.pcre.org/pcre.txt (2006)
10. Hickman, A.: High Speed regular expression matching for intrusion detection. MSc Dissertation, University of Kent, Canterbury (2006)
11. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages and computation, 2nd edn. Addison-Wesley, Reading (2001)
12. Kruegel, C., Toth, T.: Using decision trees to improve signature-based intrusion detection. In: Proceedings of the 6th Symposium on Recent Advances in Intrusion Detection (RAID 2003), Lecture Notes in Computer Science, LNCS 2820, pp 173–191. Springer, Heidelberg (2003)
13. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03). Napa, California (2003)
14. Roesch, M.: Snort—lightweight intrusion detection for networks. In: Proceedings of LISA '99: 13th Systems Administration Conference, pp 229–238. USENIX, Seattle (1999)
15. Sidhu, R., Prasanna, V.K.: Fast regular expression matching using FPGAs. In: Proceedings of the 9th International IEEE symposium on FPGAs for Custom Computing Machines (FCCM'01). Rohnert Park, California (2001)
16. Sugawara, Y., Inaba, M., Hiraki, K.: Over 10 Gbps String Matching Mechanism for Multi-stream Packet Scanning Systems. In: Proceedings of Field Programmable Logic and Applications, 14th International Conference (FPL 2004), pp 484–493. Springer, Heidelberg (2004)
17. Sutton, P.: Partial character decoding for improved regular expression matching in FPGAs. In: Proceedings of 2004 IEEE International Conference on Field-Programmable Technology (FPT2004), pp 25–32 (2004)
18. Thompson, K.: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968)
19. Tripp, G.: A finite-state-machine based string matching system for intrusion detection on high-speed networks. In: Paul Turner, Vlasti Broucek, (eds) EICAR Conference Best Paper Proceedings, pp 26–40. Saint Julians, Malta (2005)
20. Tripp, G.: A parallel "String Matching Engine" for use in high speed network intrusion detection systems. J. Comput. Virol. **2**(1), 21–34 (2006)
21. Xilinx Virtex-4 Family Overview, DS112 v1.6, Preliminary Product Specification. (2006). Xilinx Inc. Retrieved 11 January 2007 from http://direct.xilinx.com/bvdocs/publications/ds112.pdf