Computer Science at Kent

Regular expression matching using associative memory.

Gerald Tripp

Technical Report No. 4-10
October 2010

Published by the School of Computing,
University of Kent, Canterbury, Kent. CT2 7NF, UK.

Regular expression matching using associative memory.

Gerald Tripp[*]

School of Computing Technical Report No. 4-10, October 2010
University of Kent, Canterbury, Kent. CT2 7NF, UK.

Abstract.

This paper describes a method for the implementation of regular expression matching based on the use of a form of associative (or content addressable) memory. The regular expression matching is performed by converting the regular expression into a Deterministic Finite Automata, but then using associative memory to hold the state transition information. Rather than try to simplify the resulting automata, this approach starts from the point of view of each next state in the automata being arrived at from some particular area of state-input space. We implement our automata by defining a number of orthogonal regions of state-input space, each of which is compared against our current state and input data. The highest priority region that produces a match will identify a corresponding next state for the automata. This work differs from previous work in that the associative memory used performs a full set membership test, and is hence more selective than systems based on Ternary Content Addressable Memory. This report describes work in progress, which to date has produced the rule processing software and outline designs for the hardware.

1 Introduction

Network Intrusion Detection Systems monitor network traffic for various types of attack. These can be implemented on individual computers (host based) or as separate systems that monitor an entire computer network (network based). Whereas a host based implementation has only to deal with inspecting its own traffic, a network based system may have very high traffic levels to deal with, particularly when monitoring high speed sections of a network. The network based approach is however very useful as it can be used to protect machines on a network that don't (or can't) protect themselves.

Intrusion detection systems go further than a basic firewall, in that they are able to scan the entire contents of data packets rather than just looking at header fields. The approach used is typically to have a number of rules that relate to various attacks that are being searched for – each rule may first specify values in a packet header fields that are required for the rule to generate a match and, if this first stage is successful, one or more patterns of data to search for in the packet's body. A well known example of such an Intrusion Detection System is Snort [1]. Originally, the patterns searched for were just fixed strings – but

[*] Email: G.E.W.Tripp@kent.ac.uk

increasingly the patterns are defined as Regular Expressions. The use of regular expression provides for a far more powerful matching system that can allow for alternative strings or variations in the data, but unfortunately requires more processor power for the searching. We will typically have a large number of rules that we are scanning each packet with, and if we just have fixed strings to search for then we are able to use a multiple string matching algorithm such as Aho-Corasick [2] to do this. It is more difficult to match multiple regular expressions in a single search algorithm, particularly if the regular expressions perform 'wild card' matching, as we always need to remember any partial matches that we have for each of our regular expressions.

There are a number of intrusion detection systems, such as Snort, that are implemented in software. These can work well for host based intrusion detection systems, but it can be more difficult to use these for network based systems. The amount of processing required will depend of the amount of network traffic and the number of rules that each network packet needs to be tested against. For a system with a large number of rules, we may have problems keeping up on busy networks once the network data rate gets into gigabit speeds. One approach to help with this is to move some of the more computationally intensive parts of the scanning into hardware.

This paper looks at an algorithm for implementing regular expression matching in hardware, with associative memory being used to help reduce the overall amount of memory resources. The algorithms are targeted at the memory provided within Field Programmable Gate Arrays (FPGAs), although these same algorithms could also be implemented in custom Application Specific Integrated Circuits (ASICs).

The rest of this paper is structured as follows: section 2 explains the background to this work and examples of some of the other work in this research area; section 3 describes the approach used in this work, which partitions the state-input space in matching *regions*; section 4 gives the results of processing a set of regular expressions and details of the resource requirements; and the last section gives conclusions and a short discussion of the direction of future work in this area.

2 Background

In the past, Intrusion detection systems relied on searching for fixed strings, however the trend instead has been to move towards using regular expressions because of the added flexibility this gives. Regular expressions allow patterns to be created using a small number of operators, as follows:

- Concatenation: AB – means match A followed by B
- Alternate: A|B – means match A or B
- Kleene Star: A* - means match 0 or more occurrences of A

These operators can be combined together, along with the use of brackets where required, to build quite complex patterns or expressions. Intrusion detection systems such as Snort use a format referred to a 'Perl Compatible Regular Expression' (PCRE) which builds on the traditional regular expression to provide more concise formats for expressing patterns, for example:

- A+ is equal to AA*
- [A-D] is equal to (A|B|C|D)

To search for a regular expression, this will typically be first converted into an automata representation, with the automata potentially changing state on each input character and then using one or more 'terminal state' of the automata to indicate a match.

2.1 Non deterministic Finite Automata (NFA)

The first stage of processing a regular expression (RE) is to convert it into a Non-deterministic Finite Automata (NFA). This is normally done using Thompsons Algorithm [3] or some variant of this. The conversion to an NFA is straight forward, however NFAs can be relatively slow to execute in software as an NFA can typically have multiple active states and all of these and any transitions they cause need to be tracked by the software.

NFAs have however been implemented in hardware [4, 5]; this can give a very efficient implementation as each state can be modelled as a flip-flop with its Boolean value indicating whether or not it is active. Logic gates are used to implement the transitions between states and determine whether a state becomes active or not. A lot of work has been done in this research area, including a number of papers such as [6, 7] reporting designs that enable multiple bytes to be processed in the same clock cycle. One disadvantage however is that logic is synthesised to implement each NFA, and whilst this is very efficient in terms of hardware resources it is more difficult to change the REs at run time without rebuilding the logic for the design.

2.2 Deterministic Finite Automata (DFA)

As compared to the NFA, the Deterministic Finite Automata (DFA) is far easier to implement. The main reason being that the DFA has only one active state, whereas with the NFA many states can be active at once. However, in converting the NFA to a DFA, we need to have states in the DFA that represent all possible combinations of active states within the NFA, and this can sometimes lead to the DFA having a very large number of states. There are two stages to the creation of the DFA: subset construction and state minimisation. The subset construction generates the initial DFA from the NFA, and the state minimisation combines together any equivalent states that this may have created.

We can define our Deterministic Finite Automata (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. If we receive input value i in current state s then the next state is given by: $\delta(s,i)$. The output is only dependent on the current state, with a match being indicated by the automata being in a *terminal state*. The type of automata is therefore a *Moore Machine*, and has an output given by: $\lambda(s)$.

A simple implementation for a DFA can consist of a state register that holds the current state of the automata, a 2D lookup table to implement $\delta(s,i)$ and a 1D lookup table to implement $\lambda(s)$. An example is given in Figure 1. Alternative methods can be used to implement $\delta(s,i)$ and $\lambda(s)$ - such as by using logic gates, but these may not necessarily be updatable at run time in the same way that a lookup table can be.

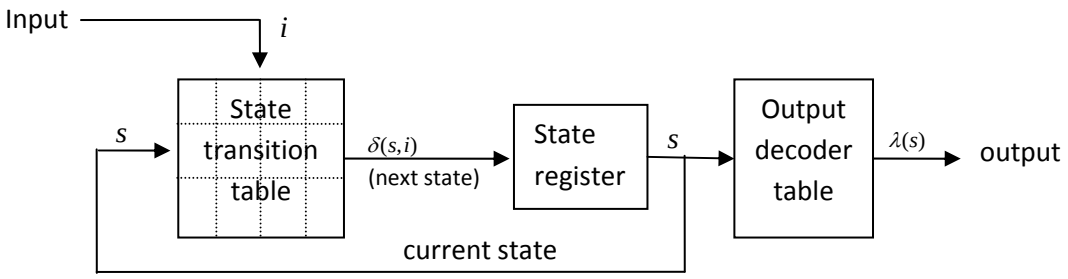

Figure 1 – Simple DFA implementation.

One of the concerns about matching REs using a DFA, is that we can sometimes end up with a very large number of states – this will however depend on the complexity of the RE, and is not always a problem in practice.

As an example, we take a set of 319 REs that were extracted from the Snort Intrusion detection System. This is a subset of REs in that it doesn't include the REs that use the PCRE "{}" operator – as this wasn't implemented in our RE processing software. Each of these REs were first converted to a NFA, and then to a DFA using standard techniques [8]. In each case the number of states required for the DFA was noted. The distribution of the number of states for this set of REs is shown in Figure 2.

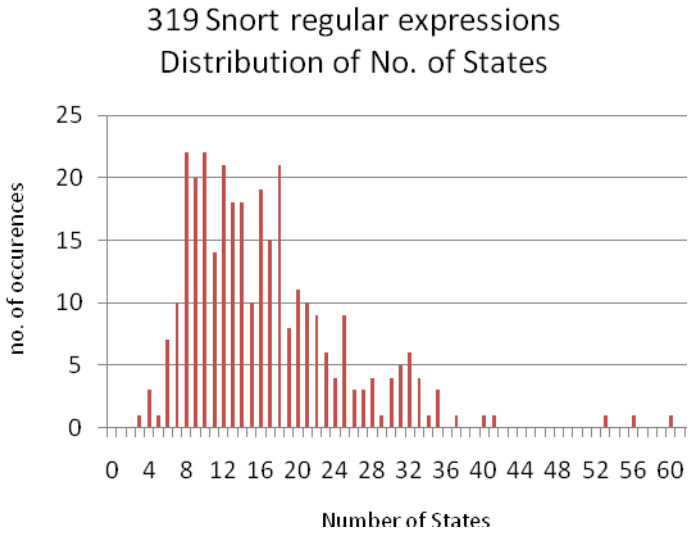

Figure 2 – Distribution of the number of DFA states for a set of 319 Snort REs.

We see from Figure 2 that we don't actually end up with very large numbers of states for these REs. We should note however, as mentioned above, that we don't currently process REs that use the PCRE {} operator. This can be used to count the number of instances of a pattern and will often lead to very large NFAs and DFAs – often maybe without the author of the Regular Expression realising this. An example, such as "ab{1,1000}c" will match between 1 and 1000 b's preceded by an 'a' and followed by a 'c'. This will produce an automata with over a 1000 states. This issue is not explored in the current work, and may be investigated later. A paper that specifically addresses this issue [9] implements an eXtended Finite Automata (XFA) by incorporating counters into the automata to hold the pattern count information and hence avoids the increase in state space.

A case that can cause problems however is when we combine multiple REs together in a single large RE particularly if one or more of the REs matches multiple wild card values. This works ok for NFAs, but for DFAs can easily cause a state space explosion. There's a strong argument in favour of having a separate DFA for each RE, particularly when these DFAs are implemented in hardware.

2.2.1 Memory requirements

In sizing our designs, we will typically work with maximum figures for $|I|$ and $|S|$ that are powers of 2, as the data will be represented as values on a bus of a fixed width. As an example, we may take the maximum value of $|I|$ as 256 to represent all possible input byte values. We will take the maximum size for $|S|$ here as 32, as this will cover over 95% of the regular expressions being considered. In practice it is not necessarily wise to structure our design to deal with the maximum possible automata size, as any outliers can be dealt with by a small number of larger automata and thus give us a lower memory use overall.

Taking these parameters, the memory requirements are as follows:

State transition table size = $\lceil \log_2(|S|) \rceil \cdot |S| \cdot |I| = 5 \times 32 \times 256 = 40 \text{ Kbits}$

Output decoder table size = $1 \times |S| = 1 \times 32 = 32 \text{ bits}$

Total = 40.03 Kbits

This leaves us with quite a large amount of memory to hold the tables for these relatively simple automata.

2.3 Input Compression

The state transition tables in the previous sub-section are very sparse as the DFA will typically only be matching a small subset of the 256 possible input characters. In some cases, the edges on our DFA may be labelled with multiple values – such as when we decide that we are not interested in the 'case' of characters, or where we match a range of values, such as all letters. We can take advantage of this by compressing the input data – this we do by categorising the input data into one of a number of 'equivalence classes' – where each member of a particular equivalence class will have an identical effect on the DFA. The output from the input compression is sometimes referred to as an *equivalence class identifier* [10]. A schematic of a DFA with input compression is shown in Figure 3.

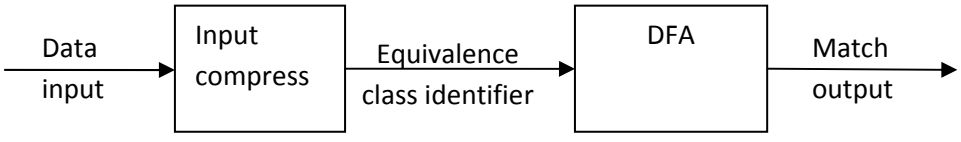

Figure 3 – DFA with input compression.

The group of REs referred to in section 2.2 were processed again, and this time the input characters were grouped into equivalence classes as described above. In each case the number of equivalence classes was recorded, and the distribution of this data is shown in Figure 4.

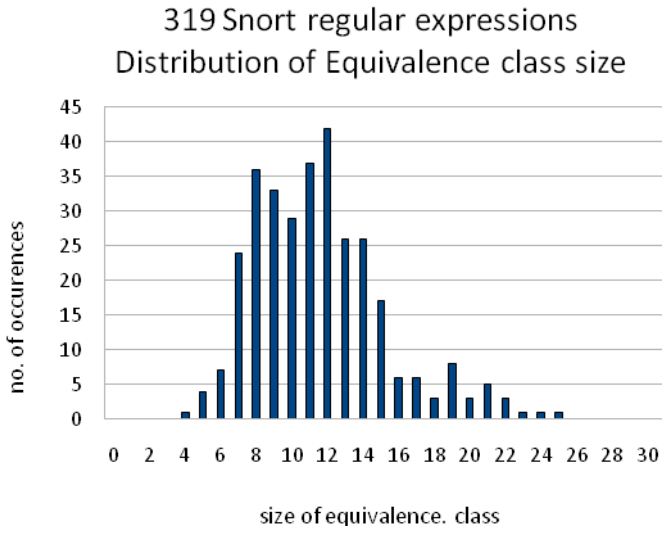

Figure 4 – Distribution of the Equivalence Class size for a set of 319 Snort REs.

2.3.1 Memory requirements

We show the compressed input set as: I'. We can take the maximum value of $|I'|$ as 32, as this will cover the number of equivalence classes for all of these automata.

Taking this new design, the memory requirements are now as follows:

Input compression table = $|I| \cdot \lceil \log_2(|I'|) \rceil = 256 \times 5 = 1280$ bits

State transition table size = $\lceil \log_2(|S|) \rceil \cdot |S| \cdot |I'| = 5 \times 32 \times 32 = 5$ Kbits

Output decoder table size = $1 \times |S| = 1 \times 32 = 32$ bits

Total = 6.28 Kbits

This is 15.7% of the memory used in the baseline version shown in section 2.2.1.

2.4 Reducing the size of the state transition table

The memory sizes quoted in sections 2.2.1 & 2.3.1 look quite small from a perspective of a standard PC, but this is not quite the same when building these systems directly in hardware. As with the PC, we can use standard computer memory modules if we wish, but these will normally have quite a high latency between requesting a piece of data and receiving it. On the other hand: memory within an FPGA is distributed throughout the chip; it is closely coupled with the logic and will have low access latency. The amount of memory available within an FPGA is however several orders of magnitude less than that available in computer memory modules. We therefore have a situation that the memory resources are very critical and can be a limiting factor in designs that rely on the use of memory.

Because of the limited amount of memory, a number of researchers have focused on either reducing the size of the DFA or reducing the amount of memory that is required for its implementation. One of the methods that has traditionally been used to compress state transition tables for compilers is to pack the two dimensional array into a one dimensional structure - an example being 'row displacement with state marking' which is described by

Grune et al. in [11]. These ideas have been used more recently within hardware based intrusion detection systems, such as the one described by Sugawara et al. in [12]. Further gains in compression for row displacement systems, based on 'next state prediction' are described by the current author in [13]. An alternative compression mechanism using run length encoding is described by Brodie et al. in [10].

Other authors have concentrated on reducing the complexity of the DFA itself. As with the STT compression this takes advantage of there often being a number of common transitions within a DFA – such as there being a high probability of going to the idle state if the match fails, or going to the first state of the automata when the first character of a pattern is received. A notable paper in this area is the one by Kumar et al. [14] on D^2FA – or Delayed input DFA. Here, the idea is that each node has a default transition that is taken if none of the other transitions succeed. When taking the default transition, we do not consume any data input but re-present that character again in the new state – this may happen more than once. An advantage is that we don't need to concern ourselves with all of the fallback or mismatch cases, as we can refer the input to another state to deal with – a disadvantage is that the DFA is no longer guaranteed to be able to process data every clock cycle and some form of input buffer will probably be needed to hold data until it is processed.

An alternative approach is taken by Becchi et al. [15] which merges similar states together. Their method then tags incoming transitions to indicate which of the original states they relate to and then uses these tags to enable outgoing transitions. This reduces the overall size of the DFA, but at the expense of greater complexity when deciding on state transitions.

2.5 Holding the state transition table in associative memory

An alternative method of implementing the state transition table is to store the contents in 'associative memory'. A simple implementation could be to use standard Content Addressable Memory (CAM) – this type of memory operates by providing the CAM with a search key, and the CAM searches the contents of its memory and returns the address of a location that contains that item of data. In the case of there being multiple matches, some form or priority mechanism is used, such as returning the lowest address that gives a match.

Instead of having a standard state transition table where we use the current state and input as indices, we store all of the valid pairs of current state and input (that take us to a non-IDLE state) in the CAM and then each cycle of the automata we present the current state and input as a search key, and the CAM will return the address that contains this pair – or indicate that there is no match. We can use the address to select the next state from a separate table, or use the no-match to select a default next state of the IDLE state. Some forms of CAM allow us to store an associated result that is returned instead of the memory address – thus saving us having a separate table for this.

This type of implementation can work very well if the state transition table is sparse – such as in some cases of string matching. Some regular expressions however may have quite complex state transition tables, particularly when the regular expression is performing wild card matching or matching a large range of possible characters.

As well as the standard (or binary) CAM, we can also use Ternary CAM (TCAM). This allows us to use wild card (or don't care) values as part of a match – typically on a bit or byte basis. We can optimise the amount of memory used in the TCAM by using the wildcard values to

match multiple data items, and hence reduce the numbers of lines of TCAM that are required. This approach is used by Alicherry in [16].

To give an example of how TCAM could be used for holding state transition information, a standard two dimensional state transition table for matching the string "abcde" is shown in Table 1 along with a TCAM implementation of this table.

	Next state						Match values		Result
Current state	Input						Current state	Input	Next State
	a	b	c	d	e				
0	1	0	0	0	0		1	b	2
1	1	2	0	0	0		2	c	3
2	1	0	3	0	0		3	d	4
3	1	0	0	4	0		4	e	5
4	1	0	0	0	5		*	a	1
5	1	0	0	0	0		*	*	0
	(Standard state transition table)						(TCAM state transition table)		

←Decreasing Priority←

Table 1 – A 'TCAM' state transition table for the string: "abcde".

It's not always easy to make use of the wild card values, and we need to be careful how we order our rows of TCAM to give the correct match priority, as multiple entries in the TCAM may give a match.

A recent paper by Meiners et al. [17] uses a TCAM based implementation, compresses the input data and provides an algorithm for choosing the state identifiers to enable TCAM entries to match multiple associated states. A deferment tree is used for deciding on the ordering of entries with the TCAM.

The next section presents an alternative approach for addressing this issue, deciding on the order of data within the Content Addressable Memory and uses a different type of CAM to give a way of reducing the number of CAM words that are required.

3 Methodology

The approach here looks at the use of associative memory to hold the state transition table and a way in which we can structure our data so as to reduce the number of entries in the associative memory.

For each next state *n*, we can specify a set N_n of the ordered pairs of state and input that take us to that next state:

$$N_n = \{(s,i) \mid s \in S, i \in I, \delta(s,i) = n\}$$

We can check if a particular pair of state s and input i values take us to next state n by testing whether $(s,i) \in N_n$ is true – however this would require a Boolean 2D table of dimensions $|S| \text{ by } |I|$ or some compressed version of this. This may require quite a lot of memory, so doesn't gain us much by itself.

3.1 Regions

As an approximation, we can define an orthogonal ***region*** that is bounded by the sets of individual inputs I_n and states S_n that appear in N_n :

$$I_n = \{i \in I \mid \exists (s,i) \in N_n, s \in S, i \in I\}$$
$$S_n = \{s \in S \mid \exists (s,i) \in N_n, s \in S, i \in I\}$$

The region R_n is then defined as the cross product of these input and state sets:

$$R_n = S_n \times I_n = \{(s,i) \mid s \in S_n, i \in I_n\}$$

We can test if particular pairs of *i* and *s* fall in a region *n*, by testing whether: $(i \in I_n) \wedge (s \in S_n)$ is true – this can be implemented as a pair of Boolean lookup tables of sizes $|I| \text{ and } |S|$ respectively, and a single AND operation.

The regions are however only approximations to the corresponding next state as: $N_n \subseteq R_n$. In some cases, such as when $(|S_n| = 1) \vee (|I_n| = 1)$ for example, we will have $N_n = R_n$, and hence the approximation is always correct; these are referred to here as being *perfect* regions. In cases where $(|S_n| > 1) \wedge (|I_n| > 1)$ then the region may or may not be perfect.

The most obvious examples of these regions can be represented by enclosed rectangular areas in a 2D table, such as in Table 2.

Current State \ Current Input	0	1	2	3	4	5
0	0	0	1	1	1	0
1	0	2	2	0	0	0
2	0	2	2	0	3	0
3	0	0	0	0	0	0
4	0	4	0	5	0	5
5	0	0	0	0	5	0

Table 2

$R_0 = \{0,1,2,3,4,5\} \times \{0,1,2,3,4,5\}$ – *imperfect*
$R_1 = \{0\} \times \{2,3,4\}$
$R_2 = \{1,2\} \times \{1,2\}$
$R_3 = \{2\} \times \{4\}$
$R_4 = \{4\} \times \{1\}$
$R_5 = \{4,5\} \times \{3,4,5\}$ – *imperfect*

Some regions however are a bit more difficult to show in a graphical form, although they still might be perfect regions. In Table 3, all regions other than R_0 are perfect. And some regions may overlap with or being contained within another, such as in Table 4.

Current State \ Current Input	0	1	2	3	4	5
0	2	2	5	0	2	2
1	2	2	0	4	2	2
2	0	1	5	0	0	1
3	0	0	3	3	3	0
4	2	2	0	0	2	2
5	2	2	5	0	2	2

Table 3

$R_0 = \{0,1,2,3,4,5\} \times \{0,1,2,3,4,5\}$ – *imperfect*
$R_1 = \{2\} \times \{1,5\}$
$R_2 = \{0,1,4,5\} \times \{0,1,4,5\}$
$R_3 = \{3\} \times \{2,34\}$
$R_4 = \{1\} \times \{3\}$
$R_5 = \{0,2,5\} \times \{2\}$

Current State \ Current Input	0	1	2	3	4	5
0	0	0	1	1	1	0
1	0	2	2	0	0	0
2	0	2	2	3	0	3
3	0	0	0	0	0	0
4	0	0	0	5	4	5
5	0	0	0	3	5	3

Table 4

$R_0 = \{0,1,2,3,4,5\} \times \{0,1,2,3,4,5\}$ – *imperfect*

$R_1 = \{0\} \times \{2,3,4\}$

$R_2 = \{1,2\} \times \{1,2\}$

$R_3 = \{2,5\} \times \{3,5\}$

$R_4 = \{4\} \times \{4\}$

$R_5 = \{4,5\} \times \{3,4,5\}$ – *imperfect*

3.1.1 Building a prioritized region list

The perfect regions are by definition disjoint, and hence if a state input pair falls into any of these regions then that must indicate the corresponding correct next state. All of the state input pairs relating to the perfect regions can be *marked* as being dealt with. We can now work iteratively through the other regions, looking for a region where that region gives a correct next state prediction when *the marked pairs are ignored* – recording the order of selection of regions and marking any new pairs that each region covers.

For Table 4, our prioritized list of regions is as shown in Table 5.

Region	Priority	Perfect?
R_1	equal 1st	yes
R_2		
R_3		
R_4		
R_5	2nd	no
R_0	3rd	no

Table 5 – Prioritized regions

In many cases we can work through all of the regions in this way, until we have an ordered list of regions to check our current s and i values against in order to determine the next state. In this example, regions R_1, R_2, R_3 and R_4 are perfect and can be tested in parallel or in any order, as at most one of these regions could contain a particular input state pair.

If we check our current s and i values against these regions in the following order: R_1, R_2, R_3, R_4, R_5, R_0 – then the first of these to give a match will identify the corresponding next state.

3.1.2 Sub-regions and partitioning

Sometimes we may find as we iterate through the regions, that there are no more regions that, when ignoring marked pairs, give a correct prediction of the next state. If this happens, we can partition one of the remaining regions into two or more ***sub-regions*** that do give a correct prediction after ignoring marked pairs. For example in Table 6, the table on the left has a region outlined, with the relevant next state values shaded; this is partitioned into two disjoint sub-regions as shown on the right, which in this simple example are both *perfect* regions – although this doesn't need to be the case.

0	0	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	1	1	0
0	0	0	1	1	0
0	0	0	0	0	0

→

0	0	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	1	1	0
0	0	0	1	1	0
0	0	0	0	0	0

Table 6 – Imperfect region split into two sub-regions

There may however be several different regions to choose from when performing a partition, and there may be several different ways in which to partition our chosen region – with some partitions being better than others, this may require us to search for what appears to be the most optimal partition. After performing a partition we add the set of sub-regions to our ordered list of regions and mark the new pairs that we have covered. We can carry on iterating over the remaining regions and performing further partitions where required.

We can treat the lowest priority region as a special case. This typically takes us to the idle state, and because it's the lowest priority will only indicate the next state when all other region matches fail. This can then act as a *default* next state, and given that it is the lowest priority we don't actually need to test for it being true.

3.2 So, why is this interesting?

Although this may look like rather a complex method to determine the next state, it does map very well into hardware. A region (or sub-region) membership test can be implemented as a single 'row' of associative memory, as shown in Figure 5.

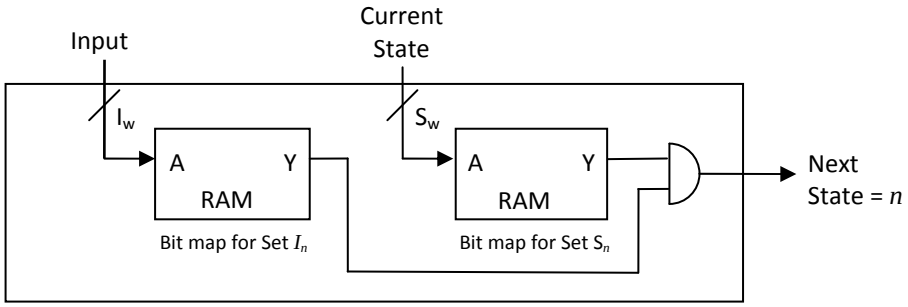

Figure 5 – Associative Memory Row

The two blocks of memory are used to hold bit-maps indicating whether $i \in I_n$ & $s \in S_n$ and will be of size: $2^{S_w} \times 1\text{-bit}$ and $2^{I_w} \times 1\text{-bit}$ respectively, where: $I_w = \lceil \log_2(|I|) \rceil$ and $S_w = \lceil \log_2(|S|) \rceil$.

We can then use these to build a multi-line associative memory DFA as shown in Figure 6.

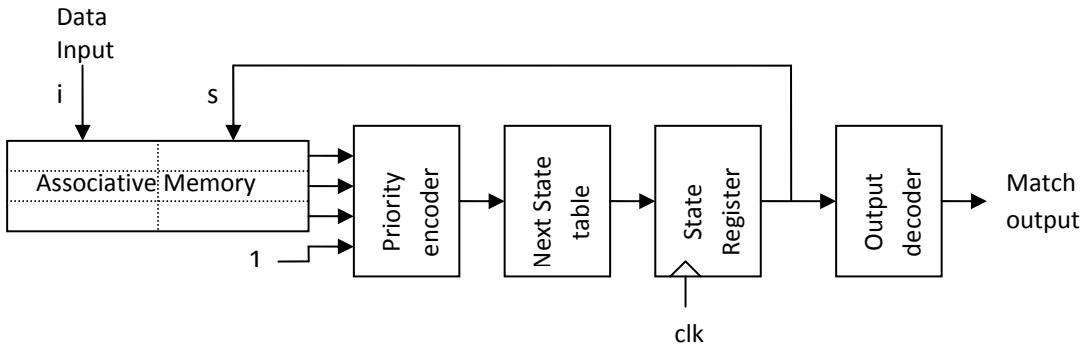

Figure 6 – Basic Associative memory Automata

Here, the priority encoder determines the lowest associative memory row number that generates a match and the next state table maps this into the corresponding next state for that line. The final '1' fed into the priority encode relates to a default match if all others fail (typically the region that takes us to the idle state) and means that we are able to save one line of associative memory.

Standard types of Ternary Content Addressable Memory (TCAM) allow for matching exact data values and also *wild card* matching, typically on an individual bit (or byte) basis – which can then be 0, 1 or don't care. The style of associative memory shown here is more flexible than TCAM, as it's testing set membership – this style of implementation is sometimes used for implementing TCAM, by setting either one bit to true to indicate an exact match or all bits to true for a wildcard value. It does however have a resource requirement that is proportional to the size of the set being tested, so our state and input sets need to be limited in size to allow practical implementations to be created.

4 Results

The snort regular expressions used in section 2.2 were processed again and this time the state transition tables were converted into a set of associative memory entries for a design based on Figure 6. The method used was as described in section 3.1, using an iterative approach of allocating regions in a priority order, and with partitioning of regions into sub-regions where necessary. For each regular expression the total number of regions and sub-regions was recorded, and a distribution of these requirements is shown in Figure 7.

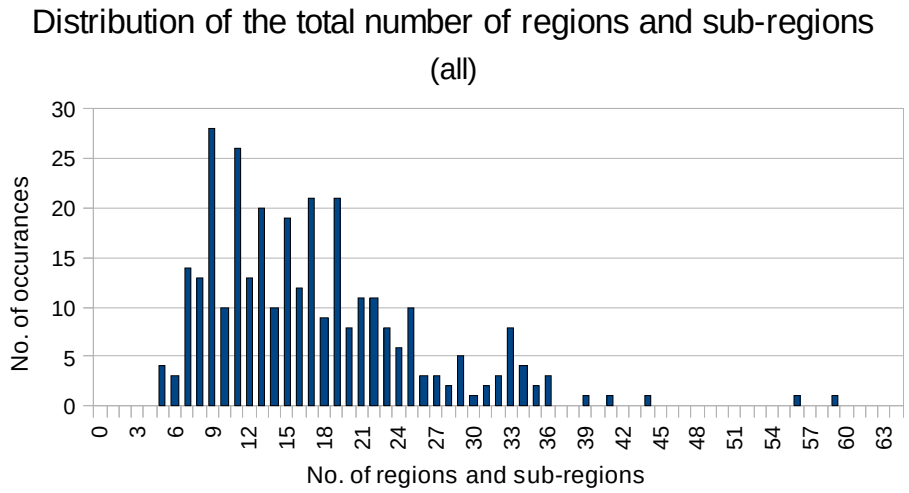

Figure 7 – Distribution of the total number of regions and sub-regions for a set of 319 Snort REs.

Allowing for a design that covers 95% of the rules, we would require a system with up to 33 regions or sub-regions – which would require 32 rows of associative memory (plus the default region). Including the input compression, and with the input and state count used previously, we would require the following amount of memory:

Lookup table	Memory required
Input compression	1.25 Kbits
Associative memory	2 Kbits
Next State table	160 bits
Output decoder	32 bits
Total =	3.4 Kbits

Table 7 - Memory requirements for basic associative automata

This is 8.6% of the requirements for the baseline figure in section 2.2.1.

In practice, there are a lot of *singleton* regions, where: $|R_n| = 1$. These could be matched using something far less complex than the associative memory system. Hence we modify the matching engine to try to match as many of these singletons as we can separately and keep the associative memory for matching the rest of the regions.

For each current state, we identify a corresponding next state and an input test value on which that transition is true. This additional test will not necessarily cover all of these singleton regions, as a current state may appear in more than one singleton region, but it works well in practice. This ends up with a more complex design, as shown in Figure 8, but one that requires significantly less associative memory rows and which appears to be more efficient overall – despite the need for the extra tables. If the input test value for the current state matches the current input then we take the predefined next state – if not we use the next state defined by the associative memory.

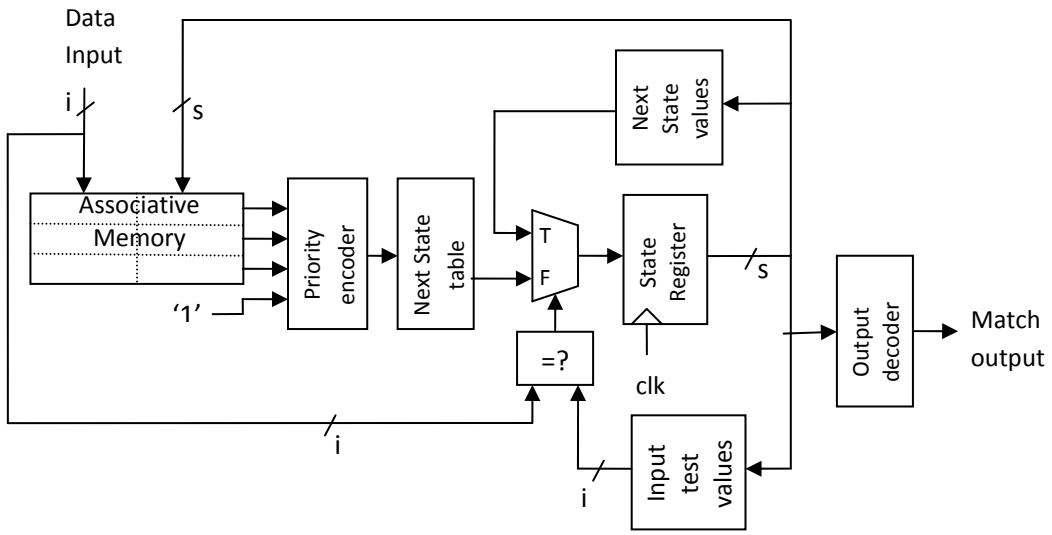

Figure 8 – Associative Automata with next state testing.

Again, we re-run our regular expression processing, this time with identifying singleton regions that can be matched with the additional test system, and recording the number of regions and sub-regions that still need to be matched using the associative memory - the distribution of this data is shown in Figure 9. As a cross check, the software checks that tables for the associative memory and the new input test system give the same results as would be given by the state transition table for all combinations of *i* and *s*.

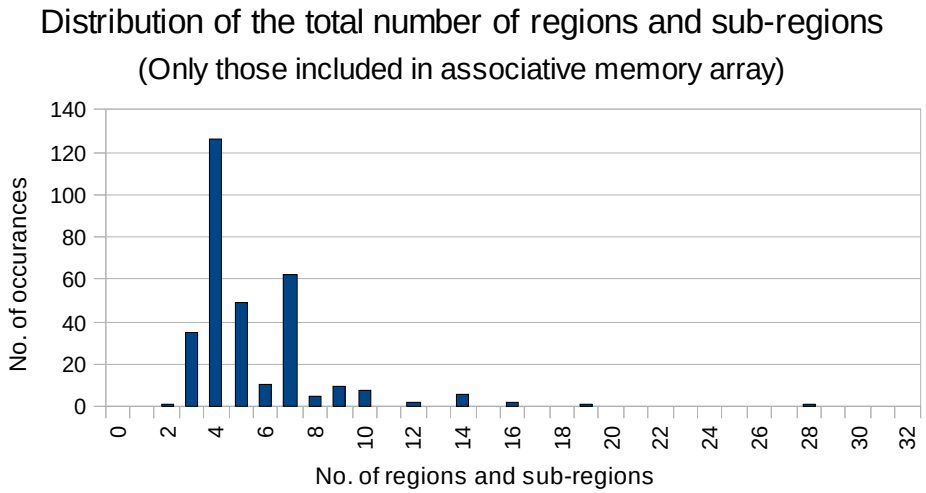

Figure 9 - Distribution of the number of regions and sub-regions requiring associative memory for a set of 319 Snort REs

This time we require far less rows of associative memory, and the distribution is also far more compact. As before we chose parameters that enable 95% of the rules to be implemented – this gives us a requirement of needing the associative memory to match 10 regions or sub-regions – which will require 9 rows of associative memory (plus the default region). The new memory requirements are now as shown in Table 8.

Lookup table	Memory required
Input compression	1.25 Kbits
Associative memory	576 bits
Next State Table	80 bits
Singleton test	320 bits
Output decoder	32 bits
Total =	2.2 Kbits

Table 8 – Memory requirements for optimized associative automata

This is 5.6% of the memory requirements for the baseline version described in section 2.2.1, and 36% of the memory requirements of the version with input compression as shown in section 2.3.1. We should also note that more than half of the memory is for the input compression table – there may be improvements that can be made here, such as sharing compression tables between multiple matching engines.

4.1 Target technology

In practice, we need to align the memory resources that we use with those available in our target technology. The technology looked at here is the Xilinx Virtex 6 series FPGAs. The memory available in these devices consists of: 36 Kbit Block RAM primitives that we can use for our input compression and Look Up Tables (LUTs) that are primarily used to implement logic, but (in the case of LUTs residing in SliceM slices) can also be used to implement small tables or for our associative memory.

We can configure the BRAM as 512 x 64-bit and use this to implement 12 of our 256x5-bit input compression tables. We can configure the LUTs as 32x1-bit tables – the particular FPGA LUT primitive chosen here is the SRLC32E shift register, which can enable the configuration data to be loaded via a serial bus at boot time and allow any of the stored bits to be selected for reading via an address bus. The memory resource requirements are shown in Table 9.

Memory table	Size	BRAMs	LUTs (SliceM)
Input compression	1.25 Kbits	1/12	
Associative memory	576 bits		18
Next state table	160 bits		5
Singleton test tables	320 bits		10
Output decoder	32 bits		1
Total primitives/engine:		0.083	34
Total number of primitives in a Xilinx XC6VLX760 FPGA		720	132480

Table 9 – FPGA memory resource requirements.

The 'SliceM' LUTs consist of about 25% of all LUTs that are available on the Virtex-6 FPGAs – so even if we use a large proportion of these LUTs as memory, there will still be plenty of

LUTs left that can be used for implementing logic. The design is heavily based on tables and the maximum number of 'engines' per FPGA is likely to be limited by the number of SliceM LUTs. A maximum of 3500 regular expression matching engines in a large FPGA looks like being a feasible prospect.

5 Conclusions

This paper describes a system that enables Regular Expressions to be matched using a DFA that is implemented using a form of associative memory to store the state transition tables. The state transition information is held in a form whereby we are searching for orthogonal regions of the state-input space and using a highest priority match to select the required next state. The system is optimised by using a more conventional matching system to look for one transition from each current state that is enabled by a single input value, and then using the associative memory to deal with the remaining, and typically more complex, transitions.

When using a set of regular expressions taken from the Snort Intrusion detection system, we find that a large majority (95%) can be implemented using a system with only 9 lines of associative memory. The design is dominated by the use of lookup tables, and a number of these are relatively small – these can be implemented using the distributed memory provided by Xilinx Virtex-6 look up tables (LUTs). Initial indications suggest that we may be able to implement around 3500 of these regular expression matching automata in a large Xilinx Virtex-6 FPGA.

5.1 Further Work

The next stage is to implement this design in VHDL and test the design using simulation. It will be interesting to see what resources are required to implement this design in practice and the overall rate that we can expect to be able to clock this design at, thus giving the per automata scan rate. This further work is currently in progress.

References

[1] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *LISA '99: 13th Systems Administration Conference*, Seattle, WA, 1999, pp. 229-238.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM,* vol. 18, pp. 333-340, 1975.

[3] K. Thompson, "Regular Expression Search Algorithm," *Communications of the ACM,* vol. 11, pp. 419-422, June, 1968 1968.

[4] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *9th International IEEE symposium on FPGAs for Custom Computing Machines, FCCM'01*, Rohnert Park, California, USA, 2001.

[5] R. Franklin*, et al.*, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *IEEE Symposium on Field-Programmable Custom Computing Machines FCCM '02*, Napa, California, USA, 2002, pp. 111-120.

[6] C. R. Clark and D. E. Schimmel, "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns," in *Field Programmable Logic and Applications, 13th International Conference, FPL2003*, 2003, pp. 956-959.

[7] P. Sutton, "Partial Character Decoding for Improved Regular Expression Matching in FPGAs," in *ICFPT*, 2004.

[8] J. E. Hopcroft*, et al.*, *Introduction to automata theory, languages and computation*, 2nd ed. Reading Addision-Wesley, 2001.

[9] R. Smith*, et al.*, "Xfa: Faster signature matching with extended automata," in *IEEE Symposium on Security and Privacy*, 2008.

[10] B. Brodie*, et al.*, "A Scaleable Architecture For High Throughput Regular Expression Pattern Matching," in *ACM ISCA'06*, 2006.

[11] D. Grune*, et al.*, *Modern Compiler Design*. Chichester: Wiley, 2000.

[12] Y. Sugawara*, et al.*, "Over 10 Gbps String Matching Mechanism for Multi-stream Packet Scanning Systems," in *Field Programmable Logic and Applications, 14th International Conference, FPL 2004*, 2004, pp. 484-493.

[13] G. Tripp, "Regular expression matching with input compression and next state prediction, Computing Laboratory Technical Report No. 3-08," University of Kent,October 2008 2008.

[14] S. Kumar*, et al.*, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *SIGCOMM*, 2006.

[15] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *INFOCOM*, 2007.

[16] M. Alicherry*, et al.*, "High Speed Pattern Matching for Network IDS/IPS," in *ICNP06*, 2006.

[17] C. R. Meiners*, et al.*, "Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems," in *19th USENIX Security Symposium (USENIX Security)*, Washington DC, 2010.