Computer Science at Kent

Regular expression matching with input compression and next state prediction.

Gerald Tripp

Technical Report No. 3-08
October 2008

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

Regular expression matching with input compression and next state prediction.

Gerald Tripp[*]

Computing Laboratory Technical Report 3-08, October 2008.
University of Kent, Canterbury, Kent. CT2 7NF, UK.

Abstract

Automata based regular expression matching can often require large amounts of memory for its state transition tables, particularly when matching multiple complex regular expressions with the same automata. For systems with limited memory resources it is common to try to compress the state transition tables. One technique called *row displacement with state marking* does this by identifying default values for the next state and then packing the remaining information into a one dimensional array. Although this compression technique works well when matching multiple strings, it is not as effective when matching multiple complex regular expressions.

This paper describes a technique called next state prediction. This performs lossy compression of the current state and input values and uses these to select a likely next state from a prediction table. This is used in conjunction with a standard row displacement with state marking algorithm and leads to an overall reduction in the memory required for the various tables.

The algorithms have been tested with a number of different design parameters, and compared with a 'baseline version' where this technique is not used. When testing this system with a set of regular expressions from the Snort intrusion detection system, the memory required was around 46% of that required for the baseline version. The design has been modelled in VHDL for use within an FPGA and tested via simulation and operates at a search rate of 2.0 Gbps irrespective of the regular expressions being searched for or the input data being scanned.

1. Introduction

Previous work by the author [1] has looked at the implementation of regular expression matching within Field Programmable Gate Arrays (FPGAs). This used standard techniques [2] for taking a Regular Expression (RE) and creating a Deterministic Finite Automaton (DFA) for its implementation. In addition to this, a further stage was added to group together sets of characters that cause the same transitions within the automaton and to compress the input data so as to form a set identifier. The automaton is then modified so as to label edges with one or more of these set identifiers.

The current paper continues this previous work, implementing some of the areas left for further study in the previous paper. In performing this further work, it was discovered that

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

some automata designed to match multiple regular expressions required larger than expected amounts of memory. The automata were implemented as before using *row displacement with state marking* [3], but this was found not to give the expected amount of compression. A further stage has been taken in this current work that looks at the use of *next state prediction* as a way of improving memory efficiency.

The next section gives a brief overview of previous work, section three gives details of the baseline design for matching multiple regular expressions and outlines how the use of Kleene-star operators causes a major increase in resource requirements. The following section looks at an algorithm for creating a next state prediction to use in conjunction with the existing design. Section five gives details of a hardware design to implement the next state prediction system and the results of the resource requirements for a sample set of regular expressions for both the old and new designs. The final section gives conclusions and ideas for further work.

2. Background

The aim of this work is to produce designs for regular expression matching systems that can be implemented efficiently in hardware, for example within a Field Programmable Gate Array (FPGA). These will typically aim to operate at a deterministic search rate, for example consuming input data at the rate of one input word per hardware clock cycle.

2.1 Overview of implementation techniques

Two main approaches are taken in the literature: to convert the RE into a Non deterministic Finite Automata (NFA) and to implement that in hardware, or to take this a stage further whereby the NFA is further converted into a DFA and to implement that in hardware. In terms of processing the REs, this is typically done using standard techniques [2] that have been used for many years.

Hardware base NFA implementation

A popular technique in the literature is to use a direct implementation of a NFA in hardware. This has the advantage of not becoming overly complex when used for RE matching, however we do have the problem that we can typically have more than one state active at once. This makes a software implementation quite complex, as each cycle we need to determine which states *(plural)* will become active next. A technique for implementing this in hardware was described by Sidhu and Prasanna [4], whereby each 'state' is implemented as a 1-bit flip flop and logic is then used to implement the 'edges' between states. This type of system will operate in parallel and can run quite fast, although the clock speed may need to be reduced for some complex automata. A lot of work has built on and improved on the original design, including techniques to optimise the character decoding [5] and schemes to extend this to operate with multi-byte input data [6] [7]. One disadvantage of this technique is however that it produces a hardware design that is specific to the set of REs being matched and thus requires it to be implemented in field programmable devices if the matching requirements for a system can change and will require at least partial reconfiguration at run time if we wish to change the rules 'on the fly'.

Hardware based DFA Implementation

A benefit of DFA implementation is that this has only one active state, and this can, for example, be held as a state variable. A disadvantage however is that the DFAs generated for REs can be quite complex. DFAs are commonly used within hardware designs, with small

examples often being synthesized into a collection of flip-flops and logic gates. Larger DFAs can be implemented as a traditional state variable and a state transition table, as we might in software. A problem however, is that the tables can be quite large and may not be practical to implement within the relatively small amounts of memory found with FPGAs.

With table based DFA implementation, quite a lot of work has gone into ways of compressing the state transition tables – as these are generally quite redundant. One method of compressing the state transition tables is to use 'row displacement with state marking' [3] - which is described below. This technique was used by Sugawara et.al for a high speed hardware based string matching system [8] which implemented Aho-Corasick multi string matching [9]. Another compression method by Brodie et.al used run length encoding as the way in which the state transition tables were compressed [10].

The hardware implementations of DFAs are generally not as efficient in resource usage as the implementations of NFAs, but they do however have the advantage that they can be table based and thus can enable simple run-time reconfiguration without changes to the hardware design.

Row Displacement with state marking

This is one of a number of techniques that was used originally to compress the size of tables used for parsers or lexical analysers within compilers. It operates on the basis that state transition tables typically have a lot of redundancy – with one or more main routes through the tables to match various strings or patterns and then large numbers of error cases that take us to the idle state or to a state that represents a match of a suffix of the data already matched. As a trivial example, we can look at an automata that matches the string “abcba”, the state transition table for which is shown as the left most array in Table 1.

Next State (state transition table)

Current state	Input a	b	c	*z*†
0 - IDLE	1	0	0	0
1 (a)	1	2	0	0
2 (ab)	1	0	3	0
3 (abc)	1	4	0	0
4 (abcb)	5	0	0	0
5 (abcba)	1	2	0	0

Next State (default)

Input a	b	c	*z*
1	0	0	0

Next State (difference)

Current state	Input a	b	c	*z*
0 - IDLE				
1 (a)		2		
2 (ab)			3	
3 (abc)		4		
4 (abcb)	5			
5 (abcba)		2		

Table 1: Original, Default and Difference Arrays

From this table, we can see that the most common (default) value for each column is the value at the top – i.e. the next state for the current input in the idle state. We can split this state transition table into two parts: a *default* table and a *difference* table, as also shown in Table 1.

The default table is a single dimensional table and will typically be quite small. The difference table is still of the same size as before, but is now sparse, and we can take

† Note: the character *z* is used in this paper to represent any character not in the 'alphabet' of characters being search for in the string or regular expression.

advantage of this when its being stored. The difference table can be divided into its separate rows and these are then packed into a one dimensional table, in a way such that there are no collisions between entries – each entry is then *tagged* with the current state that the entry belongs to. The creation of this *packed array* is shown in Table 2.

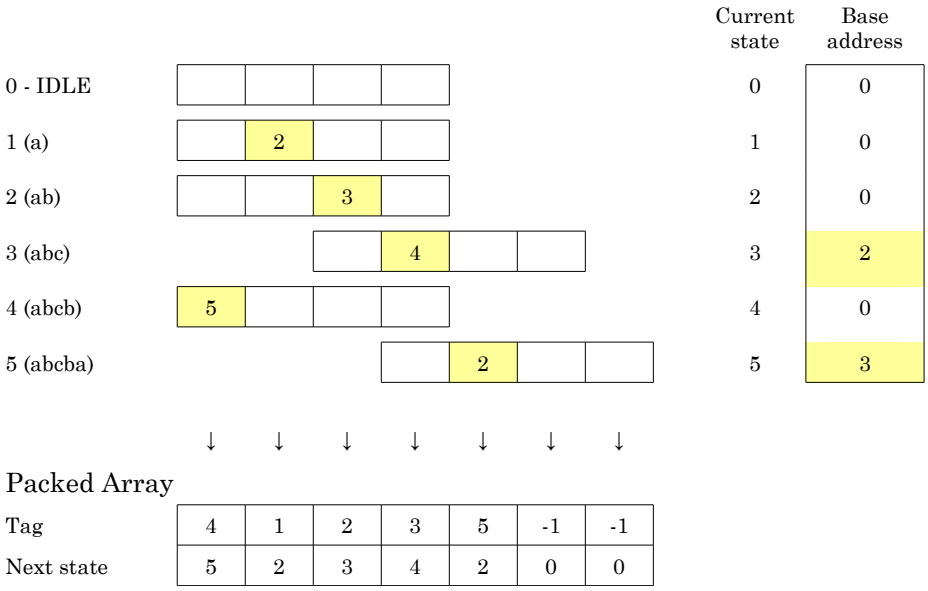

Table 2: Building the packed array

To find the next state, we look in the packed array starting at the base address for the current state plus an offset for the current input value. If the 'tag' value we find is equal to the current state, then we have found a valid entry and we can take the next state value from the table. If there is no valid entry in the packed array, then we take the entry from the default array instead. In practice, we don't need a separate table to look up the base address for the current state as this can be pre-calculated and incorporated into the default and packed arrays along with the next state values.

2.1 Previous work

Previous work by the author [1] looked at how we could build a table based DFA implementation and used 'row displacement with state marking' as the way in which to reduce the amount of memory required for its implementation. An input compression system was also used to reduce input redundancy and hence the overall automata size. This compressed all input values that have the same effect on the automata into the same 'token' value for input into a modified automata. The initial version as published only matched a single RE per automata (or *engine*), although it was noted that this could be extended to match multiple REs. When extending this system later to match multiple REs, it was noticed that the memory requirements increased far more than expected. This current paper looks at some of the reasons why this happens and a mechanism that helps to address this.

3. A baseline multiple-RE matching engine

As an initial stage, a modified version of the previous design was produced that was capable of matching multiple REs. In terms of hardware, this was a trivial change that just added extra output 'match' bits. This modified version is shown in a simplified form in Figure 1 - this is

based on the design mentioned above, which itself was developed from the design described by Sugawara et.al in [8]. Some minor changes needed to be made to the software to support having multiple REs within the same matching engine which included retaining separate terminal states for each RE and ensuring that these do not get merged during state minimization.

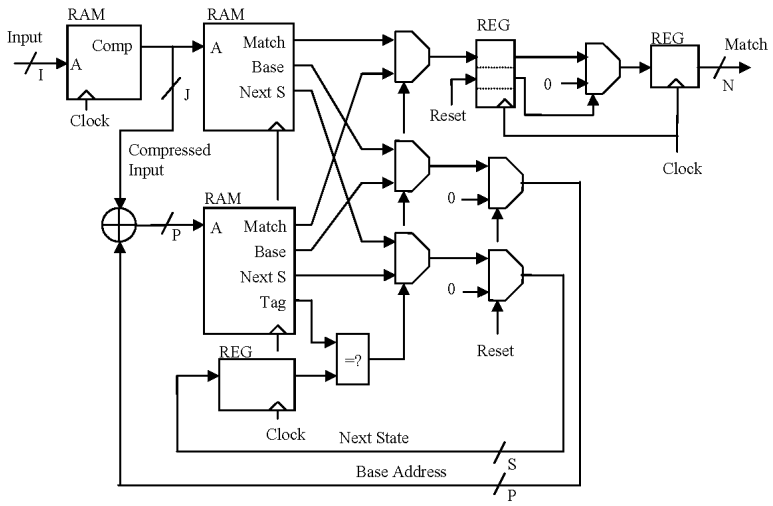

Figure 1: Baseline Multi-RE matching engine

For this baseline design, each engine is capable of matching up to N regular expressions and is implemented using a S bit state variable, a packed array with a P bit address input, and an I bit data input that is compressed in range to be of width J.

The operation of each of the three tables in this design are as follows:

- The input compression table compresses the input data in a way that input values that have the same effect on the matching automata are compressed to the same output value.
- The default array takes the compressed input and gives a default value for the next state (and other values) of the automata based on the next state given by the current input in the automaton's IDLE state.
- The packed array holds information for all of the cases where the next state given by the default array is not correct for a particular combination of input and current state.

The packed array is usually relatively sparse and this is compacted using row displacement with state marking [3]. Data is retrieved from this array using a form of indexing using the compressed input as an offset† from the start of the information for the current state. The start address of the information for a particular state is calculated in advance and stored in the packed and default arrays along with the next state information. More information on the implementation of this algorithm are given in [1].

The baseline design needs three blocks of memory as follows:

† The indexing is actually performed using bitwise exclusive-or in place of add, for reasons of hardware performance.

input compression table, of size = $J.2^I$ bits;

default array, of size = $(S+P+N).2^J$ bits;

packed array, of size = $(2S+P+N).2^P$ bits.

Thus, the total amount of memory M required is:

$$M = J.2^I + (S+P+N).2^J + (2S+P+N).2^P$$

Actual implementations may however vary so as to optimize the use of individual memory components within an FPGA – as with the design previously reported, the input compression and default arrays can be combined into a single table of size: $(S+P+N+J).2^I$ which gives no increase in memory used if $J=I$.

As an example, Table 3 shows the state transition table for matching the following Perl Compatible Regular Expression (PCRE) [11]: "/[vV]a[rR]iable/". The first line forms the content for the default array, and the other 7 highlighted entries form the data that needs to be compressed into the packed array. As can be seen, there is very little data here that is required to perform this matching operation.

Next State

Current State	Input 0 {a}	1 {b}	2 {e}	3 {i}	4 {l}	5 {r, R}	6 {v, V}	7 {*z*}
0 [idle]	0	0	0	0	0	0	1	0
1	2	0	0	0	0	0	1	0
2	0	0	0	0	0	3	1	0
3	0	0	0	4	0	0	1	0
4	5	0	0	0	0	0	1	0
5	0	6	0	0	0	0	1	0
6	0	0	0	0	7	0	1	0
7	0	0	8	0	0	0	1	0
8	0	0	0	0	0	0	1	0

Table 3: State Transition Table for matching: "/[vV]a[rR]iable/".

3.1 Limitations

In testing this system with sample patterns from the Snort intrusion detection [12] community rule set, it was found that less REs could be matched within a single matching engine than was initially expected, and because of this the REs and their state transition tables were examined carefully to see why this was.

One property of the REs that appeared to require particularly large amounts of resources in this implementation was the use of the Kleene-star operators. These allow us to match multiple instances of a pattern and are often used in the snort rule REs for absorbing white space or matching alphanumeric strings. A problem we have in intrusion detection is that we need to match *any* instance of our regular expression, irrespective of its context – i.e. we don't want the matching system to be fooled by any previous data. A problem we have here though is the this 'wild-card' style matching may also match data that could be a prefix of the RE itself.

Take the following (more complex) PCRE as an example:

"/var=[a-z]+;/"

Here we are looking for a lower case text string being assigned to the variable 'var', with a semicolon used to define the end of the pattern. So the following input data, for example, would give us a match:

var=abx; → matches: "var=abx;"

The following, rather more complex, input data examples would however also give a match:

var=var=xyz; → matches: "var=xyz;"

var=xvar=var; → matches: "var=var;"

The first assignment in each of the above will not match, as they are not terminated by a semicolon, whereas the second ones will. A problem however, is that the automaton performing the matching needs to be looking for a second instance of the string 'var' whilst it is looking for a possible right hand side to the expression. The second '=' character is the deciding point that determines that the pattern isn't matching as expected and will cause the match to drop back to the suffix 'var='. The resulting automaton table for this match is shown in Table 4.

	Next State						
	Input						
Current State	0 { v }	1 { ; }	2 { r }	3 { = }	4 { z }	5 { a }	6 { b-q, s-u, w-z }
0 [idle]	1	0	0	0	0	0	0
1	1	0	0	0	0	2	0
2	1	0	3	0	0	0	0
3	1	0	0	4	0	0	0
4	5	0	6	0	0	6	6
5	5	8	6	0	0	7	6
6	5	8	6	0	0	6	6
7	5	8	9	0	0	6	6
8	1	0	0	0	0	0	0
9	5	8	6	4	0	6	6

Table 4: State Transition Table for matching: "/var=[a-z]+;/".

We can see in this case that the data in the state transition table is more complex than we had in the example shown in Table 3. With the first example, we had a single route through the automaton table, with various error paths to follow on mismatch, which in that example were not state dependent. With the RE automaton above, we can see that whilst trying to match the '[a-z]+' part of the expression (in states: 4, 5, 6, 7 & 9), the automaton behaves in a very different way. This leads to the state transition table having far less redundancy than the one used in our first example – and thus making it more complex to compress.

In the example shown in Table 4, the next state values that differ to the default next state for that input character in state 0 are highlighted. This affects 28 out of the possible 70 table entries. In cases where multiple complex REs are being matched, then this effect becomes

even more pronounced as we are effectively trying to remember sequences of input data that can occur within multiple wild card matches in multiple REs. The existing design does of course operate correctly, but all of our highlighted entries in the state transition table end up needing to be represented within the packed array as differences from the default values.

3.2 Identifying redundancy

Not withstanding the problems noted above with the standard row displacement with state marking approaches, we can see *visually* from Table 4 that there is still quite a lot of redundancy in the state transition table that we may be able to exploit. We can see this easier if we rearrange the order of rows and columns in this table as shown in Table 5.

	Next State						
	Input						
Current State	0 { v }	5 { a }	2 { r }	6 { b-q, s-u, w-z }	1 { ; }	3 { = }	4 { z }
0 [idle]	1	0	0	0	0	0	0
1	1	2	0	0	0	0	0
2	1	0	3	0	0	0	0
3	1	0	0	0	0	4	0
8	1	0	0	0	0	0	0
4	5	6	6	6	0	0	0
5	5	7	6	6	8	0	0
6	5	6	6	6	8	0	0
7	5	6	9	6	8	0	0
9	5	6	6	6	8	4	0

Table 5: Re-ordered State Transition Table for matching: "/var=[a-z]+;/".

We can see in Table 5 that the next state behaviour can be split up into a number of rectangular *regions*, according to groups of both input and current state. Within each of these regions, the next state values are most commonly the same, and again the differences are highlighted for clarity. This time however, after comparing each entry with a default next state for its region, there are only 7 values (out of 70) that differ from the region default, as opposed to 28 out of 70 differing from the input default for the previous table. It should be remembered however that this is only a simple example and we need an algorithm that can automatically identify how to perform this optimisation in such a way that it can be implemented in a practical system.

3.3 A more intelligent 'default' table

In practice, we can have what ever content we wish in our default array – the common examples are to use: the next state for when that input occurs in the IDLE state; the most common next state for that input; or just have a default of the IDLE state and have no 'default table' at all. Any differences between the value we have in the default array and the correct next state will always be held in the packed array. Given this is the case, we can think about being more clever about what is actually held in any default array.

By definition, the content of the state transition table is dependent on both the current state and input. However, what we can do in our new version is to make a prediction about what

the next state will be. We were effectively doing this already, by using the input value, albeit rather ineffectively. It doesn't actually matter whether we are correct in our prediction, however the more accurate the prediction the less information we will need to store as a correction in our packed array and hence the less memory we will need to implement this.

Looking at the different states and input values, we can see that these regions of the table have a high probability of having the same outcome. This time we look at both input and current state, and use this to determine which region of the table we are in. We cannot use all information from both input and current state, as this would require a complete state transition table. What we can do instead is to compress both the current state and input values in a way so as to select a particular 2D table region and then to specify the default for each region.

4. Next state prediction

We can replace the original default array by three tables that allow us to generate a next state 'prediction'. As shown in Figure 2, compression tables are used for both the current state and input values so as to give indices into a 2D prediction table that gives the prediction for that region.

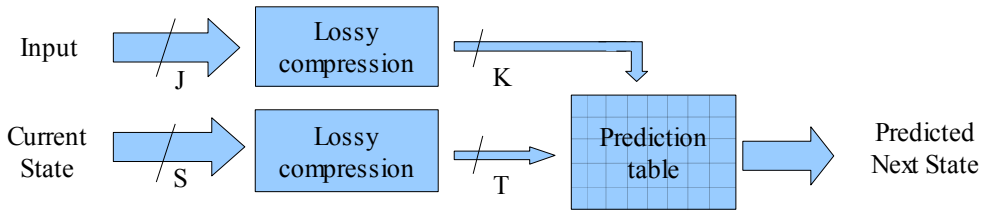

Figure 2: Predicting the next state.

The compression we use here for the current state and input are both lossy, and our next state prediction can therefore not be accurate because of this. So long as we do not lose too much information, then this doesn't matter as we are using the packed array to patch up any differences between our prediction and reality. The difficult problem is working out how to compress the input and current state so as to create a good prediction. The more we compress the input and current state values, the smaller we can make the prediction table, but the less accurate the prediction will be – and hence the larger the memory requirements for the packed array that holds the corrections.

4.1 Calculating Distance

We can look at the next state in one of the two dimensions of the state transition table and see how this index can be compressed. Taking the current state first, we have an input of width S bits and we wish to compress this to a width of T bits. We therefore have a maximum of 2^S state vectors (a list of next states for each input value) which we wish to compress into a maximum of 2^T *predictor* state vectors. We first define the distance between two state vectors: this being the number of elements in one state vector that are different to the corresponding element in that position in the other state vector. It does not matter here what the values are or how different the values are. We start now by defining a maximum distance d between state vectors and work our way through the state vectors forming sets of state vectors that are all within distance d of each other. The number of sets created S_d will

depend on the value of d and we can iteratively increase the value of d until we satisfy: $S_d \leq 2^T$.

From Table 4 we can see that with value of: $d = 2$ we have the following state vector sets:

{0, 1, 2, 3, 8}, {4, 5, 6, 7, 9}

For each of the sets, we generate a new predicted state vector that has elements consisting of the most common element values for that position from each of that set's state vectors. We now have a set of predictor state vectors. For each real state vector we create a mapping between that state vector and the predictor state vector that has the least distance. This information is then used to create the compression table that takes the S bit current state and outputs the T bit compressed current state. For our previous example, with $T = 1$, we have the state compression table and predicted state vectors as shown in Table 6.

Current State	Compressed Current State
0	0
1	0
2	0
3	0
4	1
5	1
6	1
7	1
8	0
9	1

	Predicted Next State						
	Input						
Compressed Current State	0 { v }	1 { ; }	2 { r }	3 { = }	4 { z }	5 { a }	6 { b-q, s-u, w-z }
0	1	0	0	0	0	0	0
1	5	8	6	0	0	6	6

Table 6: State compression and state predictor vectors for matching: "/var=[a-z]+;/".

The same procedure is then repeated on the table of predicted state vectors, this time in the other dimension so as to create shorter predicted state vectors by merging together next state values on the basis of distance given the different input values. From this we create the required input compression table and the prediction table. We use our previous example again, this time using $K = 2$ for the input compression, giving us the state and input compression tables and next state prediction table as shown in Table 7.

The replacement of the default table by the three new tables is likely to increase the overall amount of memory, with these tables requiring the following amount of memory:

$$T.2^S + K.2^J + (S + P + N).2^{(T+K)} \text{ bits}$$

The closer the values of T and K to S and J respectively, the more accurate will be the prediction, and hence the smaller the size of the packed array – and unfortunately, the larger the size of the predictor and compression tables. Empirical studies show that the values of $T = K = 4$ appear to be quite effective for the case where $I = J = S = P = 8$.

Current State	Compressed Current State
0	0
1	0
2	0
3	0
4	1
5	1
6	1
7	1
8	0
9	1

Input	Compressed Input
0 { v }	0 { v }
1 { ; }	1 { ; }
2 { r }	2 { a-u, w-z }
3 { = }	3 { z , = }
4 { z }	3 { z , = }
5 { a }	2 { a-u, w-z }
6 { b-q, s-u, w-z }	2 { a-u, w-z }

Predicted Next State				
	Compressed Input			
Compressed state	0 { v }	1 { ; }	2 { a-u, w-z }	3 { z, = }
0	1	0	0	0
1	5	8	6	0

Table 7: State and Input Compression and predictor tables for matching: "/var=[a-z]+;/".

5. Hardware design

The hardware design ends up being slightly different to the theoretical architecture, because of the various look-ahead functions that we need to keep the automata cycle down to one hardware clock cycle. In particular, the generation of the compressed state value can be done in parallel with the creation of the next state, in the same way that we generate the base address value. We will know these values at rule processing time and do not need to regenerate them again at run time. The hardware design we end up with is shown in Figure 3. Note that for clarity, this schematic is simplified to avoid showing the data paths and signals used for boot loading.

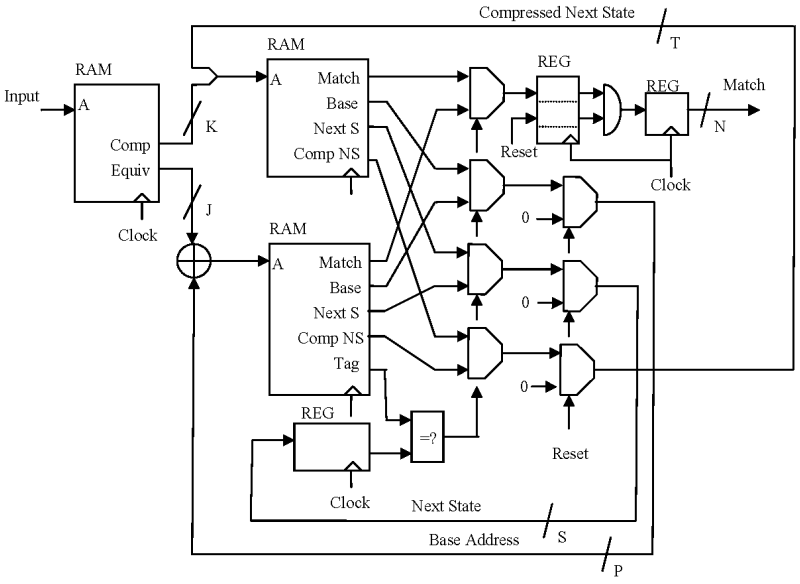

Figure 3: Schematic of regular expression matching system with next state prediction.

The overall memory requirements M' for this hardware design, taking into account the extra memory required for the look ahead operations is as follows:

$$M' = (J+K).2^{I} + (S+P+N+T).2^{K+T} + (2\mathrm{S}+P+N+T).2^{P}$$

The input is compressed twice: first the disjoint input set compression is used to give 'Equiv' as the input to the packed array address calculation, then this value is compressed again to give 'Comp', the lossy compressed value as one of the inputs to the predictor table. In practice, both compressed values can be calculated at compile time for all possible input values and stored in a single input compression table with two outputs. The two dimensional indexing into the predictor table is performed by simply concatenating the compressed next state and compressed input buses.

5.1 Determining the number of engines

To see how effective this scheme is, we take a set of 319 regular expressions from the Snort intrusion detection [12] *community rule set* and see how many matching engines are required in each case for their implementation. The software used for this takes an iterative approach with increasing the number of REs allocated to each engine to see how many will fit and each time checking whether the design could be implemented in accordance with the various maximum bus width parameters.

Original published version

Here we are able to implement only 304 of the 319 REs, with the other 15 being too large to fit. This version uses parameters: $I = J = S = P = 8$ & $N = 1$ and would require a total of 304 engines. The raw memory requirement per engine here is 12.5 K bits – giving us of course 12.5 K bits/RE.

Baseline Multi-RE Engine

Here we are still able to implement only 304 REs, with the other 15 being too large to fit in the given design size. The design uses parameters: $I = J = S = P = 8$ & $N = 7$ and requires a total of 194 engines for the REs implemented. This requires more memory, for the multiple match outputs – and has a raw memory requirement of 15.5 K bits/engine – giving us 9.9 K bits/RE.

Next State Prediction Engine

Here we can implement all 319 REs. The design uses parameters: $I = J = S = P = 8$, $K = T = 4$ & $N = 9$ and requires a total of 75 engines. These engines are larger as they include the input and state compression tables and have a raw size of 19.5 K bits/engine – thus giving us 4.6 K bits/RE. Each of these engines would require 2 BRAMs (albeit not completely used) in a Xilinx Virtex4 FPGA [13], giving a total of 152 BRAMs – or 0.47 BRAM/RE.

5.2 Hardware design results

A VHDL model has been produced and tested by simulation with a number of regular expressions and some artificial input data. This design has been synthesized and built for a Xilinx XC4VLX25-12sf363 [13] FPGA.

This gives the following resource requirements per engine:

LUTs: 75

BRAMs: 2

The design was constrained with a cycle time specified as 4.0 ns and this successfully builds with a clock to set-up time just below this. This gives an overall search rate of 2.0 Gbps, independent of the regular expressions being searched for and the data being searched. The engine produced can search for a maximum of 9 regular expressions, although this will depend on the complexity of the Regular Expressions. In the tests in the previous section, the average number of REs per engine was 4.3, with quite a wide variation between engines.

6. Conclusions

This paper looks at problems of state transition table size when implementing Regular Expression matching deterministic finite automata within Field Programmable Gate Arrays. *Row displacement with state marking* works very well as a method to reduce DFA memory use for implementing systems using Aho-Corasick multi string matching, but is less successful when applied to systems that implement multiple Regular Expression matching. This appears to be related particularly with Kleene Star operations that potentially can be used to absorb unlimited amounts of data within part of a match. This causes the automata to change what might be referred to as its *normal* default behaviour.

The technique used here is to group different states and input values together by the use of lossy compression and to use these to select a prediction of what the next state is likely to be. This prediction is more effective than just using the input value to select a default next state and results in sparser difference arrays and hence less memory for the packed array. For the example designs and test data, the prediction system used around 46% of the memory requirements per Regular Expression as that required for our baseline version where this was not used.

6.1 Further Work

The next stage planned for this work is to see how variations in the design parameters affect the overall memory utilisation. A particular area of interest is the trade off between the amount of memory used for the prediction tables as compared with that for the packed array. Taking this work a step further, it will also be interesting to see how this scheme scales to larger word sizes, and whether this introduces any new issues that need to be addressed.

References

[1] G. Tripp, Regular expression matching with input compression: a hardware design for use within network intrusion detection systems. Journal in Computer Virology, 3(2):125-134, June 2007.

[2] J.E. Hopcroft, R. Motwani, J.D. Ullman. Introduction to automata theory, languages and computation, 2nd ed. Addison-Wesley, Reading. 2001.

[3] D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Langendoen, Modern Compiler Design, Wiley, Chichester, 2000.

[4] R. Sidhu, V.K. Prasanna, 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.

[5] C. Clark, D. Schimmel, 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] P. Sutton, 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.

[7] C. Clark, D. Schimmel, Scalable multi-pattern matching on high-speed networks. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04). Napa, California. 2004.

[8] Y. Sugawara, M. Inaba, K. Hiraki, 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.

[9] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18(6) 1975, pp.333-340.

[10] B. Brodie, R. Cytron, D. Taylor, 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.

[11] P. Hazel, PCRE - Perl-compatible regular expressions, 2006. Retrieved 11 January 2007 from http://www.pcre.org/pcre.txt.

[12] M. Roesch, Snort - Lightweight Intrusion Detection for Networks. In proceedings of LISA'99: 13th Systems Administration Conference (pp. 229-238). Seattle, WA : USENIX, 1999.

[13] 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