A Scalable Architecture
For High-Throughput Regular-Expression Pattern Matching

Benjamin Brodie, Ron Cytron, David Taylor
Exegy Inc.

Abstract

We present and evaluate an architecture for high-throughput pattern matching of regular expressions. Our approach matches multiple patterns concurrently, responds rapidly to changes in the pattern set, and is well suited for synthesis in an ASIC or FPGA. Our approach is based on a new and easily pipelined state-machine representation that uses encoding and compression techniques to improve density. We have written a compiler that translates a set of regular expressions and optimizes their deployment in the structures used by our architecture. We analyze our approach in terms of its throughput, density, and efficiency. We present experimental results from an implementation in a commodity FPGA, showing better throughput and density than the best known approaches.

1. Introduction

Advances in computer networks and storage subsystems continue to push the rate at which data streams must be processed between and within computer systems. Meanwhile, the content of such data streams is subjected to ever increasing scrutiny, as components at all levels mine the streams for patterns that can trigger time-sensitive action. Patterns of interest include not only strings (*e.g.,* “dog” and “cat”) but also specifications that denote credit card numbers, currency values, or telephone numbers. The most general and widely used pattern-specification language is *regular expressions*. There is a well developed theory [11] for regular expressions and their implementation via **Finite-State Machines** (FSMs). However, the use of regular expressions for high-performance pattern matching is an area of ongoing research.

For some applications, such as packet-header filtering, the location of a given pattern may be *anchored*, in the sense that a match occurs only if the pattern begins or ends at a set of prescribed locations within the data stream. More commonly, in unstructured data streams and in packet payloads, a pattern can begin or end anywhere. Some applications require the concurrent imposition of thousands of patterns at every byte of a data stream. Examples include network intrusion detection and prevention systems, which operate using a rule base in excess of 10 thousand patterns [15];[1] email monitoring systems, which scan outgoing mail for inappropriate or illegal content; spam filters, which impose user-specific patterns to filter incoming mail; virus scanners [19], which scan files for signatures of programs known to be harmful; and copyright enforcement programs, which scan media files or socket streams for pirated content. In all of these cases, the set of deployed patterns undergoes frequent modification and can grow quickly.

Today’s high-end workstations cannot keep pace with pattern-matching applications, given the speed of data streams originating from high-speed networks and high-throughput storage subsystems. Like many before us, we turn to architectural innovation to address this performance gap; we discuss related work in Section 6.

Our architecture (with its concomitant software support) is the first design that accommodates regular expressions while providing the efficiency—in terms of both throughput and density—of systems that perform the simpler task of string matching. Our paper is organized as follows. Section 2 details the contributions of our paper. Section 3 presents an overview of pattern matching using regular expressions. Section 4 presents our pattern-matching architecture and the optimizations that result in both high throughput and density. While our design does not rely on reconfiguration, we have implemented our architecture on an **Field-Programmable Gate Array** (FPGA) to verify its operation and to quantify its performance. Results from that implementation are presented in Section 7. Even at the modest clock rate of an FPGA, our design outperforms all known approaches for regular-expression pattern-matching, and performs as well as approaches limited to string matching. We provide conservative density results that show our approach to be competitive in chip area as well. Section 8 offers some conclusions and describes future work.

[1] Extant work (Section 6) on high-speed recognition of “snort” patterns is limited to string matching; over 3 thousand of snort’s patterns are actually regular expressions and not just strings.

2. Contributions

We present a novel architecture for high-throughput pattern matching, using regular expressions to specify the patterns of interest. In detail, our contributions are as follows:

- We describe a novel pipelining strategy for computing state transitions that defers all state-dependent (iterative) operations to the final stage. As a result, we can achieve single-cycle state transition decisions, enable the use of more sophisticated and effective compression and encoding techniques, and maximize the throughput and scalability of the architecture.

- We describe a new *high throughput* FSM representation and its construction algorithm. The new FSM is designed to facilitate scaling of our architecture in the number of bytes processed per cycle. State transitions are triggered by a sequence of m symbols, rather than by a single symbol.

- We present a number of optimizations to improve the density of our architecture: character and sequence encoding to reduce a transition table's column count, run-length coding to compress the transition table, compact storage of the transition table using a layer of indirection, state permutation to improve the efficiency of the run-length coding, and storage layout to reduce the number of memory accesses required per transition.

- We describe a novel circuit for performing state transitions (selections) using the optimized structures described above. This state transition circuit processes fixed-length column segments (memory words) and allows run-length encoding columns to span multiple memory words.

- The patterns applied during a search can be rapidly changed, without altering the logic of the architecture itself. Our approach is thus suitable not only for FPGAs but also for **Application-Specific Integrated Circuits** (ASICs).

- We have written a regular-expression compiler that translates regular-expression specifications into the optimized structures used by our high-throughput architecture.

- We present results from prototyping our architecture on a commodity FPGA part clocked at 133 MHz, obtaining throughput of 4 Gbps on regular expression, pattern-matching benchmarks.

- We analyze our design for a plausible ASIC implementation. Based on conservative area and clock-speed assumptions, we show that our design achieves the highest throughput (in excess of 16 Gbps) and greatest density (in excess of 400 pattern-characters per mm^2) of all other known techniques, including those restricted to string matching.

$$\backslash\$[0-9]+(\backslash.[0-9]\{2\})\{0,1\}$$

Figure 1. Regular expression denoting US currency values.

3. Background

The area of regular expressions and the languages they denote is well studied [11] and need not be reprised in detail here. Figure 1 shows a (*perl*) regular expression that denotes (US) currency values:

- A match must begin with '$'.
- The match continues only by the appearance of one or more decimal digits.
- Optionally, the match may continue with a decimal point followed by exactly two decimal digits.

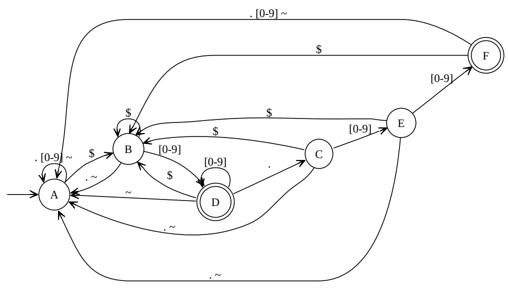
In practice, a pattern for such matches typically specifies that the match is surrounded by some delimiter, such as white space, so that the string "$347.12" yields one match instead of four ("$3", "$34", "$347", "$347.12").

The simplest and most practical mechanism for recognizing patterns specified using regular expressions is the FSM, which is formally described as the 5-tuple $(Q, \Sigma, q_0, \delta, A)$ whose components should be clear from the following example. A FSM operates as follows. It begins in state q_0. If the FSM is in some state $q \in Q$, then the next input symbol $a \in \Sigma$ causes a transition determined by $\delta(q, a)$. If the FSM effects a transition to a state $q \in A$, then the string processed up to that point is *accepted* and is in the language recognized by the FSM.

The regular expression in Figure 1 can be translated into the FSM shown in Figure 2(a) using a sequence of well-known steps [11], including a step that makes the starting position for a match arbitrary (unanchored). An FSM is typically implemented interpretively by realizing its transitions as a table: each row corresponds to a state of the FSM and each column corresponds to an input symbol. The transition table for the FSM in Figure 2(a) is shown in Figure 2(b). For example, if the FSM's current state is B and the next input symbol is 2, then the table calls for a transition to state D. In this paper, we label states by letters to avoid confusion with symbol encodings; in an implementation, a state is implemented typically by an integer index into the table.

4. Our Approach

The overall structure of our approach is shown in Figure 3. The architecture components are articulated in the



(a)

	Symbol			
State	$0\cdots9$	.	\$	$\sim\cdots$
A	A···A	A	B	A···
B	D···D	A	B	A···
C	E···E	A	B	A···
D	D···D	C	B	A···
E	F···F	A	B	A···
F	A···A	A	B	A···

(b)

Figure 2. (a) FSM to allow arbitrary pattern matches for the pattern specified in Figure 1; to avoid clutter, "$[0-9]$" denotes the set $\{0,1,2,3,4,5,6,7,8,9\}$ and the symbol "$\sim$" denotes all symbols not in $\{0,1,2,3,4,5,6,7,8,9,\$,.\}$. (b) Transition table for the FSM; if Σ is the ASCII character set, then the table has 256 columns.

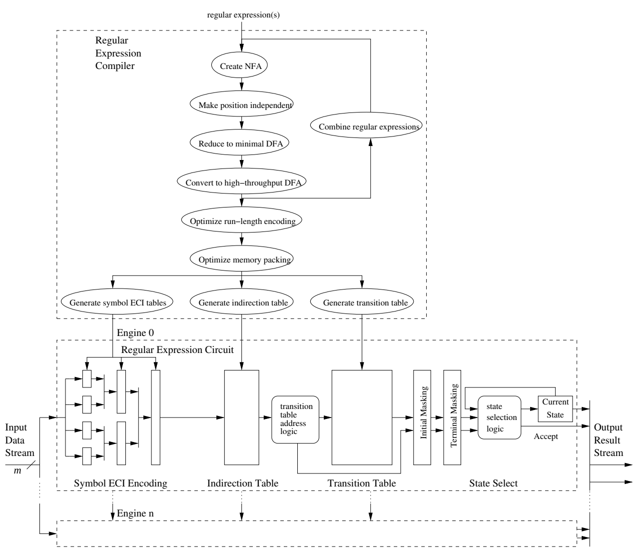


Figure 3. Overall structure of our approach.

Regular Expression Circuits (or engines), in the bottom dashed boxes. Our approach raises some challenging optimization problems, and these are realized in the Regular Expression Compiler, shown in the top dashed box.

With reference to Figure 3, our approach can be summarized as follows. A set of regular expressions is presented to the Regular Expression Compiler. The Regular Expression Circuits implement high-throughput FSMs, which are introduced in Section 4.1. Each FSM's resources are first reduced by the alphabet encoding presented in Section 4.2. High-throughput and encoding are combined in Section 4.3 to produce the Symbol ECI Encoding block shown at the left of the Regular Expression Circuit. The FSM's transition function is compressed as described in Section 4.4. The resulting transition information is deployed in the Indirection and Transition Tables. Those tables' construction and the state selection logic are presented in Section 4.5.

4.1 . A High-Throughput FSM

The FSM as presented thus far processes one input symbol (byte) at a time, performing a table lookup on each byte to determine the next state. However, modern interconnection interfaces transport multiple bytes per cycle, making the FSM the "bottleneck" in terms of achieving higher throughput.

We consider an extension of the FSM that allows it to perform a single transition based on a string of m symbols [6]. Formally, this adaptation yields a FSM based on the alphabet Σ^m; the corresponding transition table is of size $|Q||\Sigma|^m$. This apparently dramatic increase in resource requirements is mitigated by the compression techniques we describe in Section 4.2. For convenience, we let δ^m denote a transition function that operates on sequences of length m, with $\delta = \delta^1$.

As an example, consider doubling the effective throughput of the FSM shown in Figure 2(a) by processing two bytes at a time. Based on the table in Figure 2(b), if the current state is E, then the input sequence "2\$" would result in a transition to state B: $\delta^2(\mathrm{E}, 2\$) = \delta(\delta(\mathrm{E}, 2), \$) = \mathrm{B}$. By accounting for all such two-character sequences, a complete transition table can be computed for this higher-throughput FSM.

Figure 4 shows an algorithm that effectively doubles the throughput of a FSM. To obtain higher throughput FSMs, that algorithm can be invoked repeatedly, each time doubling the effective number of symbols processed per cycle.

function DOUBLESTRIDE(δ^k) : δ^{2k}
  foreach $q \in Q$ **do**
    foreach $a \in \Sigma^k$ **do**
      foreach $b \in \Sigma^k$ **do**
        $\delta^{2k}(q, ab) \leftarrow \delta^k(\delta^k(q, a), b)$
end

Figure 4. Algorithm for doubling the stride of a FSM.

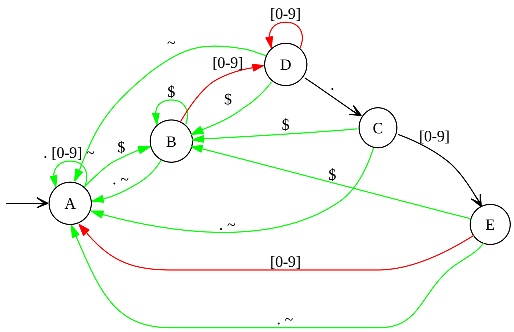


Figure 5. FSM for the regular expression in Figure 1. Transition on a red edge indicate a pattern match; transition on a green edge indicates the first symbol of a new potential match. All edges are green except as follows: (B,D), (D,D), and (E,A) on [0-9] are red; (D,C) on **.** is black; (C,E) on [0-9] is black.

Accepts and Restarts Because it performs multiple transitions per cycle, the higher throughput FSM can traverse an accepting state of the original FSM during a transition. As discussed in Section 1, many applications require matching at every position in the target data stream. Moreover, each “accept” may be required to report the origination point of the match in the target. To support recognition of accept and restart conditions, we augment the traditional transition function with two flags (bits): The first flag indicates a restart transition and the second flag indicates an accept transition. In our FSM diagrams, edges are colored green for restart and red for accept. Figure 5 shows such a diagram for an automaton that processes one symbol at a time and recognizes the language denoted by Figure 1. Figure 7 shows the associated transition table with the flags included. Because information about accepts is associated with edges (the δ function), a FSM with colored edges can have fewer states than canonical FSM.

The actions of an automaton with the colored edges are as follows. The automaton includes state variables b and e to record the beginning and end of a match; initially, $b = e = 0$, and the index of the first symbol of the target is 1. Transitions are then performed as follows:

function PARTITION($Q, \Sigma, \delta : Q \times \Sigma \mapsto Q$) : $(result, K, \kappa : \Sigma \mapsto K)$
  $WorkList \leftarrow \Sigma$
  $result \leftarrow \emptyset$
  $i \leftarrow -1$
  while $WorkList \neq \emptyset$ **do**
    $w \leftarrow$ some element of $WorkList$
    $equiv \leftarrow \{\, a \in \Sigma \mid \forall q \in Q \; \delta(q, a) = \delta(q, w) \,\}$
    $i \leftarrow i + 1$
    $K \leftarrow K \cup \{\, i \,\}$
    foreach $sym \in equiv$ **do** $\kappa(sym) \leftarrow i$
    $result \leftarrow result \cup equiv$
    $WorkList \leftarrow WorkList - equiv$
end

Figure 6. Finding equivalent columns of $\delta : Q \times \Sigma \mapsto Q$.

black: e is advanced by m—the stride of the automaton, which is the length of the input string that caused the transition. In Figure 5, $m = 1$. A match is in progress and the portion of the target string participating thus far begins at position b and ends at position e, inclusively.

red only: e is advanced by m and a match is declared. The target substring causing the match starts at position b and ends at position e.

green only: b is advanced by m and e is set to b. The automaton is restarting the match process.

red and green: The red action is performed before the green action.

4.2 . Alphabet Encoding

The size of a FSM’s transition table (δ) increases exponentially with the length of the the input sequence consumed in each cycle. In terms of a computing platform’s fastest storage resources, the size of that table places a practical limit on the length of an input sequence that can be accommodated. In this section, we present an algorithm that encodes the symbol alphabet so as to reduce the number of columns needed in a high-throughput transition table.

Given a transition table $\delta : Q \times \Sigma \mapsto Q$, an $O(|\Sigma|^2|Q|)$ algorithm for partitioning Σ into a (typically) reduced set of equivalence classes is shown in Figure 6. Using our example, the algorithm develops the equivalence classes that form the 4 columns shown in Figure 2(b). We call the index into the encoded table an ECI. In summary of the ideas presented thus far, Figure 5 shows a FSM that processes

Symbol	**Equivalence Class Identifier** (ECI)
$	0
.	1
0–9	2
all else	3

(a)

	ECI			
State	0	1	2	3
A	(B,1,0)	(A,1,0)	(A,1,0)	(A,1,0)
B	(B,1,0)	(A,1,0)	(D,0,1)	(A,1,0)
C	(B,1,0)	(A,1,0)	(E,0,0)	(A,1,0)
D	(B,1,0)	(C,0,0)	(D,0,1)	(A,1,0)
E	(B,1,0)	(A,1,0)	(A,0,1)	(A,1,0)

(b)

Symbols		ECI	Symbols		ECI
0	0	0	0	1	1
0	2	2	0	3	1
1	0	0	1	1	1
1	2	3	1	3	1
2	0	4	2	1	5
2	2	6	2	3	7
3	0	0	3	1	1
3	2	1	3	3	1

(c)

	ECI							
State	0	1	2	3	4	5	6	7
A	(B,1,0)	(A,1,0)	(D,1,1)	(A,1,0)	(B,1,0)	(A,1,0)	(A,1,0)	(A,1,0)
B	(B,1,0)	(A,1,0)	(D,1,1)	(A,1,0)	(B,1,1)	(C,0,1)	(D,0,1)	(A,1,1)
C	(B,1,0)	(A,1,0)	(D,1,1)	(A,1,0)	(B,1,0)	(A,1,0)	(A,0,1)	(A,1,0)
D	(B,1,0)	(A,1,0)	(D,1,1)	(E,0,0)	(B,1,1)	(C,0,1)	(D,0,1)	(A,1,1)
E	(B,1,0)	(A,1,0)	(D,1,1)	(A,1,0)	(B,1,1)	(A,1,1)	(A,1,1)	(A,1,1)

(d)

Figure 7. (a) Symbol encoding for 256 ASCII characters); (b) Transition table with 3-tuple entries: (*next state*, *restart flag*, *accept flag*)(b); (c) Symbol encoding for 2 symbols; (d) Transition table for stride 2 FSM.

a single symbol at a time, with red edges indicating "accept" and green edges indicating "restart". Analysis using our partitioning algorithm yields the symbol encoding shown in Figure 7(a) and the transition table shown in Figure 7(b). Applying the algorithm from Section 4.1 to double the stride and recomputing ECIs as described above, we obtain a FSM that accepts two symbols at a time. The resulting symbol encoding and transition table are shown in Figure 7(c–d). Of the 16 possible two-character input sequences, the transition table requires only 8 columns due to the ECI mapping.

4.3 . High-Throughput with Encoding

We now combine the ideas of Section 4.1 and Section 4.2 to develop an algorithm that consumes multiple bytes per cycle and encodes its input to save time (in constructing the tables) and space (in realizing the tables at runtime).

The set of transformations begins with a regular expression r and perform the following steps to obtain FSM^m:

1. The FSM d is constructed for r in the usual way [11]. For example, the regular expression in Figure 1 results in the FSM shown in Figure 2(a).
2. A set of transitions ρ is computed that would allow the automaton to accept based on starting at *any* position in the target. This is accomplished by simulating for each state a λ-transition to q_0. Specifically, ρ is computed as follows:

   $\rho \leftarrow \emptyset$
   foreach $p \in Q$ **do**
       foreach $a \in \Sigma$ **do**
           $\rho \leftarrow \rho \cup \{ (p, a) \mapsto \delta(q_0, a) \}$
3. From d and ρ, the base FSM[1] is constructed by the algorithm in Figure 8. The results of that algorithm are shown in Figure 5 for our running example.
4. State minimization is performed on a high-throughput FSM by the standard algorithm [11], except that states are initially split by incoming edge color (black, green, red), instead of by whether they are final or nonfinal states in a traditional FSM.
5. Given a FSM^k, a higher throughput FSM^{2k} with alphabet encoding is constructed by the stride-doubling algorithm discussed in Section 4.1.

The Symbol ECI Encoding Block To process an input stream at high throughput, our architecture shown in Figure 3 requires logic for transforming a sequence of m symbols into an ECI, suitable for presentation to a transition table. As shown in Figure 9, a straightforward method for performing this operation is to perform pairwise combinations using direct-addressed tables.

In theory, this technique may be used to assign an ECI to any number of input symbols. In practice, memory efficiency significantly degrades as the number of symbols covered by the final equivalence class increases.

4.4 . Transition Table Compression via Run-Length Coding

Run-length coding [13] is a simple technique that can reduce the storage requirements for a sequence of symbols that exhibits sufficient redundancy. The idea is to code the string a^n as the run-length n and the symbol a; we use the notation $n(a)$. Thus, the string aaaabbbcbbaaa is run-length

function $\text{BASEDFA}(Q, \Sigma, q_0, \delta, A, \rho)$: $FSM^1 = (Q', \Sigma', q'_0, K, \kappa, \delta')$

$K \leftarrow \{0\}$
foreach $a \in \Sigma$ **do** $\kappa(a) \leftarrow 0$
$WorkList \leftarrow \{q_0\}$
while $WorkList \neq \emptyset$ **do**
  $w \leftarrow$ some element of $WorkList$
  $WorkList \leftarrow WorkList - \{w\}$
  $Q' \leftarrow Q' \cup \{w\}$
  foreach $a \in \Sigma$ **do**
    $targs \leftarrow \{t \mid (\exists s \in w)\delta(s,a) = t\}$
    $targs \leftarrow \text{LAMBDACLOSURE}(targs)$
    $red \leftarrow targs \cap A \neq \emptyset$
    $green \leftarrow (\forall s \in w - \{q_0\})(\forall t \in Q - \{q_0\})$
      $\delta(s,a) \neq t$
    $\delta' \leftarrow \delta' \cup \{w \times a \mapsto (targs, green, red)\}$
    if $targs \notin Q'$
    then $WorkList \leftarrow WorkList \cup \{targs\}$
end

Figure 8. Construction of FSM^1 from the traditional FSM for a regular expression.

State	ECI 0	1	2	3
A	5(B,1,0)	3(A,1,0)	1(A,1,0)	5(A,1,0)
B			1(D,0,1)	
C			1(E,0,0)	
D		1(C,0,0)	1(D,0,1)	
E		1(A,1,0)	1(A,0,1)	

Figure 10. Run-length coded transition table

coded as 4(a)3(b)1(c)2(b)3(a). If each symbol and each run-length requires one byte of storage, then run-length coding reduces the storage requirements for this example by three bytes (from 13 bytes to 10 bytes).

Examining Figure 7(b), we see ample opportunity for run-length coding in the columns of the transition table. For the encoded symbols 0 and 3, the table specifies the same three-tuple for every previous state. Figure 10 contains a run-length coded version of the transition table in Figure 7.

While column compression can save storage, it appears to increase the cost of accessing the transition table to obtain a desired entry. In Section 4.5, we describe an organization for the columns that allows quick access to the column entries while retaining the advantages of compression. Based on a priority encoder, a circuit is presented that decompresses a sequence of column entries in a single cycle.

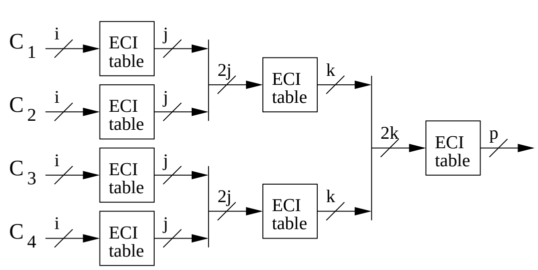


Figure 9. Implementation of the "Symbol ECI Encoding" block of Figure 3, using direct-addressed tables and pairwise combination.

4.5 . Supporting Variable-Length Columns in Memory

The compressed-column layout developed thus far must be mapped to a physical memory, in which the entire table will typically be too large to be fetched in one memory access. In the discussion that follows, we assume that x transition-table entries can be fetched per memory cycle. FPGAs and similar devices often support memory banks that can be configured in terms of their size and word-length. Moreover, the size of a given entry depends on the number of bits allocated for each field (run-length, next state, start flag, accept flag). For example, a physical memory that supports 5 access per cycle and holds 3 entries per word is accommodated in our model by setting $x = 5 \times 3 = 15$.

Some architectures offer more flexibility than others with respect to the possible choices for x. For example, the bits of an FPGA Block Ram can sometimes be configured in terms of the number of words and the length of each word.

The following considerations generally apply to the best choice for x:

- Memory accesses are generally reduced by driving x as high as possible.
- The logic in Figure 14 grows with the number of entries that must be processed at one time. The impact of that growth on the circuit's overall size depends on the target platform and implementation. It turns out that significant FPGA resources are required to realize the logic in Figure 14.

Transition Table Memory (TTM) The compressed columns should be placed in physical memory as compactly as possible. Figure 11(b) shows the columns packed into a memory with $x = 3$. Due to the varying length of each column, a given column may start at any entry within a memory word.

By introducing a layer of indirection in the transition table, we can leverage the memory efficiency provided by

ECI	pointer	index	count
0	0	0	1
1	0	1	3
2	1	1	5
3	3	0	1

Indirection Table

(a)

addr			
0	5 (B,1,0)	3 (A,1,0)	1 (C,0,0)
1	1 (A,1,0)	1 (A,1,0)	1 (D,0,1)
2	1 (E,0,0)	1 (D,0,1)	1 (A,0,1)
3	5 (A,1,0)		

Transition Table Memory

(b)

Figure 11. Indirection and Transition tables for the run-length coded table in Figure 10. The shaded shaded areas of the tables represent the second column (ECI 1) of Figure 10. Each row occupies one word of storage.

ECI	pointer	index	count
0	0	0	1
1	0	1	3
2	1	1	5
3	3	0	1

Indirection Table

(a)

addr			
0	5 (B,1,0)	3 (A,1,0)	4 (C,0,0)
1	5 (A,1,0)	1 (A,1,0)	2 (D,0,1)
2	3 (E,0,0)	4 (D,0,1)	5 (A,0,1)
3	5 (A,1,0)		

Transition Table Memory

(b)

Figure 12. Indirection and Transition tables with precomputed prefix sums (see Figure 11).

symbol	ECI	pointer	initial index	terminal index	word count
$	0	0	0	0	0
.	1	0	1	0	1
[0–9]	2	1	1	2	1
~	3	3	0	0	0

Indirection Table

addr			
0	5 (B,1,0)	3 (A,1,0)	4 (C,0,0)
1	5 (A,1,0)	1 (A,1,0)	2 (D,0,1)
2	3 (E,0,0)	4 (D,0,1)	5 (A,0,1)
3	5 (A,1,0)		

Transition Table Memory

Figure 13. Indirection Table and TTM organization as used by our State Select Block

run-length coding and compact deployment of entries in the TTM. As shown in Figure 11(a), the Indirection Table contains one entry for each ECI. Since ECIs may be assigned contiguously, the Indirection Table may be directly addressed using the ECI value for a given sequence of m symbols. Once the Indirection Table entry is retrieved using the input symbol ECI, the *pointer* is used to read the first memory word from the TTM. Recall x is the number of entries per memory word in the TTM. An entire column is accessed by starting at address $index$ and reading w consecutive words from the TTM, where w is given by:

$$w = \left\lceil \frac{count + index}{x} \right\rceil \tag{1}$$

The *index* and *count* values determine which entries in the first and last memory words participate in the column, as shown in Figure 11. As discussed below and shown in Figure 3, accesses to the Indirection Table and TTM can be pipelined with each other and with the other components of our design.

State Selection While the TTM offers compact storage of the compressed columns, state-selection logic becomes more complex for the following reasons:

- A compressed column may span multiple words of the TTM.
- Within a word, locating the apprpriate entry for a given state index requires computing prefix sums of the compressed entries.
- The start of a compressed column may begin in the middle of a TTM word. Thus, entries before the start must be suppressed for state selection.
- The end of a compressed column may occur before the end of a TTM word. Thus, entries after the end must be suppressed for state selection.

Because the run lengths are fixed inside each entry, the prefix sums can be precomputed. We store them, instead of the run-lengths, as the "coefficient" in each tuple. For example, given the tables in Figure 11, the precomputed sums are shown as the coefficients in Figure 12.

We also can reduce the amount of logic to determine the beginning and end of the compressed column as follows. The start of each column is specified in the Indirection Table using the *pointer* and *index* fields, which provide the TTM word containing the first entry and the index within that word of the entry. The number of words w occupied by the compressed column is then given by Equation 1. Each fully occupied word contains x entries of the compressed column. In the last word, the largest index occupied by the compressed column is given by:

$$(count + index - 1) \bmod x \tag{2}$$

Based on the above, Figure 13 shows the Indirection Table and TTM entries for the transition table given in Figure 15. Essentially, the old *count* value is translated into *Word Count* and *Terminal Index* values. This translation does not affect the contents of the TTM, but reduces the logic needed to process these tables.

The State Select block's logic is shown in Figure 14, operating on a transition table column containing two entries that spans one TTM word, where each word stores four entries.

The output of the *Word Counter* reflects the number of

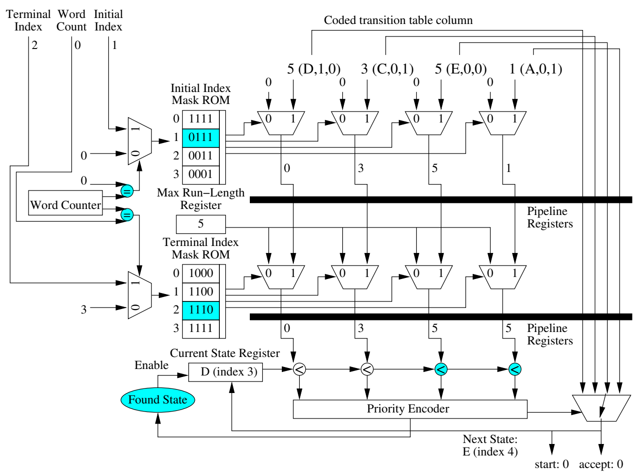


Figure 14. Design of the State Select Block

memory words that have been examined for the current transition table column. If the current memory word is the first word spanned by the column, then the *Initial Index* is used to retrieve a set of mask bits from the *Initial Index Mask ROM*. These bits are used to mask off preceding entries in the memory word that are not part of the transition table column. Masking is accomplished by forcing the run-length sums to be zero. Note that if the current memory word is not the first word spanned by the column, then the no entries are masked at this stage.

The next stage "masks" the entries in the last memory word that are not part of the transition table column. The run-length sums for entries that are not part of the transition table column are forced to the value of the *Max Run-Length Register*. This value records the maximum number of entries in a transition table column (*i.e.* the number of rows in the uncoded transition table; also the value of the run-length sum for the last entry in each coded transition table column). If the current memory word is the last memory word spanned by the transition table column (value of the *Word Counter* is equal to *Word Count*), then the *Terminal Index* is used as the address to the *Terminal Index Mask ROM*. If this is not the case, then no entries are masked during this stage. Forcing the run-length sums of trailing entries to be the maximum run-length sum value simplifies the *Priority Encoder* that generates the select bits for the multiplexor that selects the next state. This masking process produces an output vector from the less-than comparisons with the following property: the index of the left-most '1' bit is the index of the next state entry, and all bits to right of this bit will be set to '1'.

5. Optimizations

In this section, we describe some optimization problems posed by our pattern-matching architecture. We discuss the relevant trade-offs and present solutions that work well in practice. While the approach described in Section 4.5 fills the TTM as compactly as possible, we investigate here reorganizing the TTM to reduce the number of storage accesses required per cycle and thus gain better throughput.

Some of the optimization problems discussed in this section can be cast as classical *bin packing* or *knapsack* problems [7], where the number of entries per word (x) corresponds to the bin (or knapsack) size for the classical problem. An additional constraint is that the total number of words cannot exceed the size of a given TTM.

5.1 . Improving the effectiveness of run-length coding

Consider some column of a transition table and its disposition in a TTM. The greater the compression of that column, the fewer words necessary to hold the compressed column in the TTM. The number of words occupied by a compressed column can be reduced if the compression of the column can be improved. The number of memory accesses required for a search is determined by the disposition of compressed columns in the TTM and the pattern by which those columns are accessed. The pattern depends on the set of regular expressions in the engine and the particular input data processed through the engine. In a FSM^m, m input symbols are resolved to an ECI which induces one column lookup. The number of memory accesses depends on the the length of the columns in the coded transition table and the column access pattern. The column access pattern depends on the regular expression (or set of regular expressions) in the engine and the input data. The total number of memory accesses for a given search can be expressed as:

$$W = N \sum_{i=1}^{|K|} f_i w_i \tag{3}$$

where w_i is the number of words spanned by row i in Transition Table Memory, f_i is the relative frequency that column i is accessed, and N is the number of equivalence class identifiers produced by the input data. Figure 15 illustrates the benefits of state reordering for our running example, yielding 8 TTM entries instead of the original 10.

The approach we take is a greedy one. We wish to maximize the length of the runs for the most columns, thereby minimizing the length of each encoded column. We begin by creating a difference matrix using the algorithm shown in Figure 16. Given two states q and p, $D[q][p]$ records indicates the number of symbols (ECIs) for which the transition function from q and p differs in some way (state, restart flag, or accept flag).

State	ECI 0	1	2	3
A	(B,1,0)	(A,1,0)	(A,1,0)	(A,1,0)
C	(B,1,0)	(A,1,0)	(E,0,0)	(A,1,0)
E	(B,1,0)	(A,1,0)	(A,0,1)	(A,1,0)
B	(B,1,0)	(A,1,0)	(D,0,1)	(A,1,0)
D	(B,1,0)	(C,0,0)	(D,0,1)	(A,1,0)

(a)

State	ECI 0	1	2	3
A	5(B,1,0)	4(A,1,0)	1(A,1,0)	5(A,1,0)
C			1(E,0,0)	
E			1(A,0,1)	
B			2(D,0,1)	
D		1(C,0,0)		

(b)

Figure 15. (a) State reordering and (b) resulting optimized run-length coded transition table

function DIFFMATRIX$(Q, \Sigma, \delta) : D$
  foreach $q \in Q$ **do**
    foreach $p \in Q$ **do**
      foreach $s \in \Sigma$ **do**
        $D[q][p] \leftarrow D[q][p] + \delta(q,s) \neq \delta(p,s)$
end

Figure 16. Calculation of the difference matrix for optimizing the run-length encoding.

function RLOPTIM$(Q, q_0, \Sigma, \delta, D) : FSM' = (Q', \Sigma, q_0', \delta')$
  $Used \leftarrow \emptyset$
  $cur \leftarrow \text{MAX}(\sum_{i \in Q} D[i])$
  while $Used \neq Q$ **do**
    $next \leftarrow \text{MAX}(D[curr] - Used)$
    $Used \leftarrow Used \cup \{next\}$
    $cur \leftarrow next$
end

Figure 17. Optimization of run length encoding in the TTM

Indirection Table

symbol	ECI	pointer	initial index	terminal index	word count
$	0	0	0	0	0
.	1	0	1	2	0
[0–9]	2	1	0	0	1
~	3	2	1	1	0

Transition Table Memory

addr			
0	5 (B,1,0)	4 (A,1,0)	1 (C,0,0)
1	1 (A,1,0)	1 (E,0,0)	1 (A,0,1)
2	2 (D,0,1)	5 (A,1,0)	
3			

Figure 18. Indirection table for run-length coded transition table that minimizes the number of memory accesses.

We next order the states from some starting point based on the entries in the difference matrix. We choose the state that preserves the most run lengths to get the next label. The starting state we choose is the state that has the largest column-sum in the difference matrix. The idea for picking that state first is that it is the state that is the most different from all others. We want it moved to an end, rather than in the middle, to preserve the longest runs. This algorithm is outlined in Figure 17, and the resulting Indirection Table and TTM are shown in Figure 18.

5.2 . Memory packing

The table layout presented in Section 4.5 allows a compressed column of the coded transition table to span any sequence of contiguous locations in the TTM. Naïve packing of coded table columns into physical memory can thwart the aforementioned optimizations by incurring an extra memory access for each table column. For example, the compressed column associated with ECI 1 in Figure 11 occupies 3 entries and could fit in a single word of the TTM. However, the column began at a location that caused it to span 2 memory words instead of 1.

We can take advantage of the flexibility provided by the layer of indirection by ensuring that a coded transition table column spans at most w words, where $w \leq \left\lceil \frac{count}{x} \right\rceil$. In Figure 12, ECI 1 maps to a compressed column that happens to span two words. The entries can be rearranged so that each compressed column is still contiguous, and only ECI 2 spans the necessary two words.

We seek to pack the TTM as fully as possible while minimizing the number of TTM entries spanned by every column. This problem is a variant of the classical *fractional knapsack problem* [9] where w is the constraint or objective function [4].

In our algorithm shown in Figure 19, R is the set of run-length encoded columns, and x is the number of entries that can be held by a word of memory. The basic idea is to find the longest run-length encoded column and chose it first. We pack it into memory words guaranteeing that it achieves the best possible packing. We then take the number of remaining entries in the last column and apply *subset sum* on it with the remaining run-length encoded columns. This packs the memory as full as possible without causing additional memory accesses. We repeat this process until no encoded columns remain.

```
procedure ColPack(R, x)
    while R ≠ ∅ do
        longest ← MaxLength(R)
        R ← R − {longest}
        Pack(longest)
        sum ← x − (longest mod x)
        L_0 ← {0}
        foreach col ∈ R do
            L_i ← MergeLists(L_{i−1}, L_{i−1} + Length(col))
            remove from L_i every element that is
            greater than sum
        S ← Max(L_n)
        foreach col ∈ S do
            R ← R − {col}
            call Pack(col)
end
```

Figure 19. Optimal packing of memory words.

6. Related Work

Architectures for pattern-matching are generally evaluated in terms of their performance (throughput) and their efficiency (density), as follows:

Throughput refers to the sustained rate at which a data stream can be processed—the number of bytes per second that can be accommodated by a design and its implementation. There are two ways throughput can be increased:

- The clock speed can be increased, subject to the implementation's technology limitations and the complexity of the logic that must execute within a cycle. Results based solely on increasing a part's clock speed are generally applicable to whatever logic is hosted on that part, if the logic's critical path can operate at that clock speed. Thus, we do not consider clock speed alone as a primary basis for comparison.
- The architecture can itself be designed or parameterized to accommodate a given number of bytes ingested per cycle. By including the "width" of the data stream in the design, architectures can be fairly compared in terms of their performance, scalability, and cost.

Density is a measure of an implementation's capacity per unit area. Capacity is a measure of the size of the pattern-matching problem undertaken by the implementation circuitry. The circuitry required to realize a given approach can be measured in die area (mm^2), gates, or device-specific resources (**Lookup Tables** (LUTs) on an FPGA, memory cells).

Because we seek to compare our architecture both to other regular-expression pattern-matching approaches and to the simpler string-matching problem, we measure density based on the number of characters used to specify the instance of the search problem. This method works well, in that if a regular-expression approach is provided a set of strings, then the problem-specification size is the same as for approaches that accept only constant-string patterns.

6.1 . String-matching approaches

For some applications, patterns of interest are limited to a finite set of constant strings. Although our approach is more general, our performance rivals the following approaches to the simpler problem. One idea [5] for reducing resources while maintaining high throughput suggests factoring out common logic and using attached, fast memory to trade logic for lookups. Another idea [17] uses pipelining to search concurrently for matches at various offsets in the data stream.

Automata-based approaches use state in place of combinational logic to track matches in progress. One approach [16] deploys a state-based algorithm (due to Knuth-Morris-Pratt) in hardware, but this approach does not scale for high throughput on a given data stream, nor does it take advantage of commonality among strings in the pattern set. A similar approach [3] pipelines the algorithm and proves that a buffer of modest size allows their design to consume one byte per cycle without stalling.

Another interesting approach [18] suggests splitting a byte stream into (up to) 8 bit streams, with an (Aho-Corasick) automaton relegated to finding matches on each bit stream. This idea reduces the fan-out from each state (from up to 256 to just 2), but the throughput is scaled only by increasing the clock rate: rates in excess of 1.25 GHz are required to sustain an ingest rate of 10 Gbps. Using estimates based on a 0.13-μm process, the approach can achieve a density of up to 320 characters/mm^2. Another approach [1] also based on the Aho-Corasick algorithm reorganizes the state tables to decrease access time, and shows correspondingly improved throughput, but with an ingest rate of just one byte per cycle.

Strings can also be found using hashing techniques; one source [8] suggests a circuit based on Bloom filters to control the false-positive rate.

The above approaches make progress in reducing resources or improving throughput for pattern matching based on a sets of constant strings. We next turn to the more general class of regular expressions.

6.2 . Regular expressions

Approaches here can be characterized in terms of the extent to which they rely on reconfiguration to save resources or improve throughput. The most basic approach [14] consists simply of compiling a set of regular expressions into an automaton characterized in a **Hardware Definition Language** (HDL). The resulting synthesized logic is then loaded into an FPGA. The automaton ingests a data stream at one byte per cycle, and pattern-set changes require costly resynthesis. The long synthesis path can be optimized by compiling regular expressions directly into netlists [12].

Another approach [10] describes a tree-like, recursive architecture for pattern matching that cannot handle arbitrary regular expressions. Both this and the above approach derive their benefits from reconfiguration.

Clark and Schimmel [6] offer an approach that handles arbitrary regular expressions at scalable throughput. They propose position-specific character decoding up-front, to save on the otherwise replicated comparisons deployed throughout an automaton. The decoding takes place on multiple bytes concurrently, thus providing high throughput by width rather than by clock scaling. They also propose using *nondeterministic* FSMs, which can take substantially less space than their deterministic counterparts. However, they count on reconfiguration to change the wiring of the FSMs, and so their designs are nto suitable for an ASIC.

7. Performance Analysis and Experimentation

In this section, we evaluate our architecture and relate its performance to the extant work described in Section 6. While our architecture is amenable to implementation in an ASIC or FPGA, we present results from our prototype using a Xilinx Virtex-II FPGA with a PCI-X interface (64-bits at 133 MHz). As currently implemented, our circuit accepts four bytes per cycle, contains eight parallel engines, and is described in approximately 2700 lines of Verilog code. Using a modest clock frequency of 133 MHz, the prototype achieves sustained throughput of 4 Gbps.

We use the results of our prototype implementation to estimate area and performance for an ASIC implementation in current process technology. The four byte per cycle design requires a total of 96k bits of embedded memory per engine, which dominates the area requirements for the design. (The prototype implementation only requires 860 4-input LUTs per engine for logic operations.) Given that our architecture is memory-based to support dynamic pattern updates, the performance of our design is determined by the available memory bandwidth.

We assume that each 6T SRAM cell (one bit) consumes 2 μm^2 of die area, a conservative estimate for modern process technology [2]. We also conservatively assume that embedded SRAM may be clocked at 500 MHz. Finally, we assume a total die size of 200 mm^2 and allocate 10% of the area for overhead (glue logic and interfaces). Under these assumptions, an ASIC implementation of our architecture would contain 923 parallel engines (0.19 mm^2 per engine) and achieve a peak throughput of 16 Gbps. Scaling the clock speed or increasing the number of bytes consumed per cycle results in a linear increase in throughput.

To analyze the density of an ASIC implementation, we performed a experiments to determine the number of exact strings and regular expressions that each engine is capable of storing. As discussed in Section 6, the size of a regular expression is its specification size and not the size of the strings it denotes. Indeed, a regular expression can denote an infinite set of strings. Thus, the density results we present for regular expressions are influenced by the patterns in the tests we conducted. We consider this fair since we did not formulate the patterns.

There are a number of factors that influence the capacity of each engine, including the dimensioning of the ECI encoding tables and transition table memories. Figure 20 shows the results of our prototype implementation on several benchmarks of interest. A U.S. Government agency provided the 13 *forensic* patterns, and they denoted various personal information such as telephone, bank account, and credit card numbers. We divided the snort [15] (as of July 2005) set into those patterns that could be found using string matching and those that required a regular-expression engine. To compare our approach against string pattern-matchers, we generated random strings and packed them into our FSM engines. Our compiler sorts the strings to obtain more efficient ECI encodings, which result in smaller tables and more patterns per engine.

We observe that the density achieved by our architecture (228–438 chars per mm^2) is better than the density achieved by previous methods, even those limited to string matching. Our throughput is also better than the best known technique that operated at 10 Gbps and assumed a 1.5 GHz clock speed for embedded SRAM [18]. Our technique not only provides better performance at 500 MHz, but also provides the regular expression capability required for a variety of critical applications including network intrusion detection and data mining.

8. Conclusions

We have presented an architecture that solves the more general problem of regular-expression pattern matching with throughput and density rivaling the best known solutions to the simpler problem of string matching. Our experiments, conducted under modest technology assumptions, show that we can sustain a throughput of 16 Gbps at a density that supports nearly 1,000 regular-expression engines on a die.

Name	Patterns	Total Chars	Engines	Density (chars/mm^2)	Throughput (Gbps)
forensic regex	13	665	7	438	16.0
snort strings	1406	22445	279	423	16.0
snort regex	315	11126	287	204	16.0
random strings	2258	36128	833	228	16.0
Tan and Sherwood [18]	Bank Size 2048 Bytes			321	8.4
	Bank Size 64 Bytes			55	10.1

Figure 20. Performance results. Each of our engines occupies 0.19 mm^2 on an ASIC and can process streams at 16 Gbps when clocked at 500 MHz.

References

[1] M. Aldwairi, T. Conte, and P. Franzon. Configurable string matching hardware for speeding up intrusion detection. *SIGARCH Comput. Archit. News*, 33(1):99–107, 2005.

[2] F. Arnaud et al. A functional 0.69 μm^2 embedded 6t-sram bit cell for 65nm cmos platform. In *VLSI 2003*, 2003.

[3] Z. K. Baker and V. K. Prasanna. High-throughput linked-pattern matching for intrusion detection systems. In *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, pages 193–202, New York, NY, USA, 2005. ACM Press.

[4] P. E. Black. *Dictionary of Algorithms and Data Structures*. NIST, 2004.

[5] Y. Cho and W. Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[6] C. Clark and D. Schimmel. Scalable pattern matching for high speed networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.

[8] S. Dharmapurikar, M. Attig, and J. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[10] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. S. Jr., and O. R. Birkeland. A recursive MISD architecture for pattern matching. *IEEE Transactions on Very Large Scale Integration*, 12(7):727–734, 2004.

[11] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[12] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *FCCM'02: 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[13] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Springer Verlag, 2002.

[14] J. Moscola, M. Pachos, J. W. Lockwood, and P. Loui. Implementation of a streaming content search-and-replace module for an internet firewall. In *Symposium on High Performance Interconnects (HotI)*, pages 122–129, 2003.

[15] M. Roesch. Snort—lightweight intrusion detection for networks. In *LISA'99: 13th Systems Administration Conference*, pages 229–238, 1999.

[16] R. P. S. Sidhu, A. Mei, and V. K. Prasanna. String matching on multicontext FPGAs using self-reconfiguration. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 217–226, New York, NY, USA, 1999. ACM Press.

[17] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps FPGA-based network intrusion detection system. In *Proceedings of 13th International Conference on Field Programmable Logic and Applications*, 2003.

[18] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA'05: 32nd Annual International Symposium on Computer Architecture*, pages 112–122, 2005.

[19] D. Uluski, M. Moffie, and D. Kaeli. Characterizing antivirus workload execution. *SIGARCH Comput. Archit. News*, 33(1):90–98, 2005.