

# Regular Expression Acceleration at Multiple Tens of Gb/s

*Jan van Lunteren, Jon Rohrer, Kubilay Atasu, Christoph Hagleitner*

IBM Research, Zurich Research Laboratory  
8803 Rüschlikon, Switzerland  
email: jvl@zurich.ibm.com

## 1. INTRODUCTION

The frequency of network attacks increases every year, and at the same time the attack methods are becoming more sophisticated. To keep up with these trends, signature-based NIDSs such as Snort [1] apply more powerful and flexible content-filtering rules, often involving pattern conditions defined by regular expressions. This has triggered a substantial amount of research and product development in the area of hardware-based accelerators for pattern matching, as this seems to be the only viable approach for scanning network data against the increasingly complex patterns at wire-speed processing rates of multiple tens of gigabits per second.

Although pattern-matching schemes have been studied for a long time and have been covered by many publications, the majority of these, and in particular those schemes intended for hardware implementation, have been limited to simple patterns comprised of strings and basic regular expressions [2]-[10]. More complex regular expressions are addressed by recent work in the field of FPGA-based pattern matching, however, most of this work is based on the automatic generation of dedicated match functions in hardware for a given set of patterns [11]-[15]. For NIDS applications, this has the inherent shortcoming that the hardware has to be regenerated for each pattern update. Another drawback is that several of these schemes rely on large numbers of smaller engines (e.g., NFAs) that each handle only a few patterns as part of a “divide and conquer” approach. The resulting large aggregate state of all these engines makes it impossible to provide efficient multi-session support for handling many streams in an interleaved fashion, where at any given time the number of open sessions can be on the order of millions for typical networking environments.

What seems to be lacking are hardware-based pattern-matching schemes that (1) efficiently support advanced regular expression features (2) do this by only applying a limited amount of parallelism to keep a small session state, (3) are programmable through modification of a data structure in an SRAM, and (4) provide wire-speed match performance of multiple tens of gigabits per second.

At the IBM Zurich Research Laboratory, we are try-

ing to fill this gap by investigating novel concepts for accelerating a wider range of regular-expression features in hardware. The key focus is on improving the storage efficiency, as this will allow larger portions of the data structure to be fit into fast on-chip caches, with the resulting increases in hit rate directly translating into higher scan rates. Our approach is based on a novel type of programmable state machine in hardware, called BaRT-based Finite State Machine (B-FSM), with BaRT being the name of the underlying search algorithm [16]. The B-FSM-based pattern-matching scheme, introduced in [17], is one of the most storage-efficient schemes reported in literature, capable of achieving deterministic processing rates of multiple tens of gigabits per second in ASIC technology, while scanning input streams against tens of thousands of patterns in parallel.

Although the basic B-FSM scheme provides efficient support for string and simple regular-expression matching, recent and ongoing research has resulted in multiple extensions for an efficient handling of advanced regular-expression features. One essential extension was to have the compiler select an optimum hash function for each individual state for mapping the corresponding transition rules in the B-FSM data structure, instead of using one hash function to map an entire cluster of states as was done in the original scheme. The original B-FSM engine has also been extended with direct support for flexible match conditions involving character classes, case-insensitive matches, and similar conditions. Both extensions enable substantially more compact representations of the state transitions in the data structure.

We have furthermore developed a novel distribution function that exploits simulated annealing to achieve an optimized allocation of patterns to the B-FSM engines involved in the same scan operation. This distribution function is able to effectively reduce the total storage requirements even for a modest number of B-FSM engines, which is necessary to keep the session state small as described above. A similar pattern distribution approach is described in [18]. The main difference between our work and the work of [18] is that we use a weighted graph to represent the compatibility between patterns based on a novel energy function (see Section 3.3),

whereas [18] makes use of an unweighted graph representation. Additionally, [18] relies on a relatively simple heuristic algorithm, whereas our approach tries to identify closer to optimal solutions based on a stochastic iterative optimization technique.

To limit the size of the state space, various other optimizations can be applied, including the use of auxiliary variables and instructions that operate on these auxiliary variables [19, 20]. The bit-split DFA [9, 21] is another approach that enables further optimizations by exploiting the bit-level parallelism. These approaches are orthogonal to ours, and can be combined with our technique.

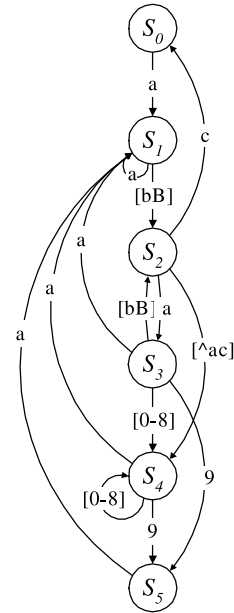
## 2. B-FSM TECHNOLOGY

### 2.1. Transition Rules

The B-FSM engine is a fast programmable Deterministic Finite Automaton (DFA) implementation in hardware that constitutes the basic building block of our pattern-matching engine. At the core of the B-FSM technology is the concept of specifying state transitions using so-called transition rules. Each transition rule consists of a test part, containing match conditions for the current state and input values, and a result part, containing a next state and an optional output vector. The B-FSM engine, discussed below, ‘executes’ this specification by searching in each cycle for the highest-priority transition rule that matches the actual values of the state register and input, and uses the result part of that rule to update the state register and to generate output. The extended B-FSM supports the following match conditions on the input value: exact match, case-insensitive match, class match and don’t care. The class-match condition allows to test whether an input is part of a character class, which can be a predefined class, such as digit or alphanumeric, or any arbitrary class. Each condition can also be negated, meaning that the transition rule only matches if the input value does not match the condition.

The transition-rule concept is now illustrated by an example that involves the detection of all occurrences of the following ‘artificial’ pattern anywhere in an input stream:  $a[bB][^c][0-8]^*9$ . Fig. 1(a) shows a state transition diagram for this pattern-matching function, which can be generated using existing algorithms that map regular expressions on DFAs. State  $S_0$  is the initial state and state  $S_5$  corresponds to the detection of the pattern. Note that the ‘default’ transitions to state  $S_0$  from the other states are left out to keep the state diagram simple.

The transitions within the diagram can now be described using the transition rules listed in Fig. 1(b). For example, rule  $R_{10}$ , is a ‘conventional’ rule implementing the corresponding state transition from state  $S_4$  to state  $S_5$  in Fig. 1(a). A more flexible rule is rule  $R_1$ , which involves a don’t care condition for the state and therefore covers multiple state



(a) State-transition diagram  
 (“default” transitions to state  $S_0$  are not shown).

rule	current state	input	→	next state	priority
$R_0$	*	*	→	$S_0$	0
$R_1$	*	a	→	$S_1$	1
$R_2$	$S_1$	[bB]	→	$S_2$	2
$R_3$	$S_2$	*	→	$S_4$	2
$R_4$	$S_2$	a	→	$S_3$	3
$R_5$	$S_2$	c	→	$S_0$	3
$R_6$	$S_3$	\d	→	$S_4$	2
$R_7$	$S_3$	9	→	$S_5$	3
$R_8$	$S_3$	[bB]	→	$S_2$	3
$R_9$	$S_4$	\d	→	$S_4$	2
$R_{10}$	$S_4$	9	→	$S_5$	3

(b) Transition rules.

**Fig. 1.** Example.

transitions to state  $S_1$  in the diagram. Another special rule is rule  $R_6$  which tests whether an input value is part of a common class  $[0-9]$ , also denoted as  $\backslash d$ , to make a transition to state  $S_4$ . The latter class is likely to be shared by other patterns, instead of a more specific class  $[0-8]$  as shown in Fig. 1(a). This results in a correct operation, because rule  $R_6$  has a lower priority than rule  $R_7$ , which tests for an input value 9, and will therefore only be selected if the input is part of  $[0-8]$ . For similar reasons, rule  $R_3$  involves an input-don’t care condition instead of the more specific class  $[^ac]$  that was shown in Fig. 1(a), and rule  $R_9$  involves a class  $\backslash d$  instead of the more specific class  $[0-8]$ . Because all rules defined for the same states that involve overlapping input conditions have higher priorities, this results in a correct operation.

## 2.2. Transition-Rule Selection

The B-FSM engine searches for the highest-priority matching transition rule in each clock cycle using a data structure that is comprised of multiple so-called transition-rule tables. Each table contains all the transition rules related to one cluster. For each individual state, a separate hash function is used to index a transition-rule table to find the highest-priority rule for the current input value. This hash function has been derived from the Balanced Routing table (BART) search algorithm [16], hence the name BART-based Finite State Machine (B-FSM). The hash function for each state is defined by a mask vector that specifies how the hash index bits are extracted from the current state and input vectors according to the following bit-wise operation:

$$index = (state' \text{ and not } mask) \text{ or } (input' \text{ and } mask) \quad (1)$$

where **and**, **or**, and **not** are bit-wise operators, and  $state'$  and  $input'$  comprise (typically the least or most significant) segments of the current state and input vectors having sizes equal to the hash index. Each mask vector bit specifies whether a corresponding hash index bit is extracted from the state vector (mask bit equals zero) or from the input vector (mask bit equals one).

For each state, the mask vector is selected by the B-FSM compiler, which will be discussed below, such that the number of collisions for any hash index value is limited to a configurable bound  $P$ . The value of  $P$  is based on the memory width to ensure that the maximum number of  $P$  colliding rules for each hash index value can be retrieved using a single memory access. This is followed by a parallel comparison of all  $P$  rules with the actual state and input vectors to find a matching rule.

The relatively simple rule-selection function as described above enables a fast implementation in hardware involving a short critical path. Furthermore, optimization techniques applied by the B-FSM compiler on the state clustering and encoding result in a very high storage efficiency: for many match functions, an approximately linear relation between the number of transition rules and the actual storage requirements is achieved, resulting in one of the most storage-efficient matching schemes in the industry. More details on the hash function and optimization techniques can be found in [17].

Next, the rule selection for transition rules  $R_2$  to  $R_{10}$  shown in Fig. 1(b) will be illustrated. Note that rules  $R_0$  and  $R_1$  have a wildcard condition for the current state, and are handled separately, see next section. It is assumed that the state and input vectors have a width of 8 bits, whereas the mask vector and the hash index have a width of 7 bits, and that a maximum of  $P = 2$  transition rules is mapped on each index value. The latter value was chosen for illustrative purposes. In typical implementations, the number of rules mapped on a given index value is usually larger.

rule	current state	input	→	next state	priority
$R_2$	00000010b	0x100010b	→	$S_2$	2
$R_3$	00000001b	xxxxxxxxxb	→	$S_4$	2
$R_4$	00000001b	01100001b	→	$S_3$	3
$R_5$	00000001b	01100011b	→	$S_0$	3
$R_6$	00010000b	0011xxxxb	→	$S_4$	2
$R_7$	00010000b	00111001b	→	$S_5$	3
$R_8$	00010000b	01x00010b	→	$S_2$	3
$R_9$	00000100b	0011xxxxb	→	$S_4$	2
$R_{10}$	00000100b	00111001b	→	$S_5$	3

(a) Transition rules.

010000b	rule $R_7$	rule $R_6$
...		
000100b	rule $R_{10}$	rule $R_9$
000011b	rule $R_5$	rule $R_3$
000010b	rule $R_2$	
000001b	rule $R_4$	rule $R_3$
000000b	rule $R_8$	

(b) Transition-rule table entries.

**Fig. 2.** Transition-rule selection.

Based on the above parameters, all transition-rule tables contain a total of  $2^7 = 128$  lines, each storing  $P = 2$  rules, resulting in a total of 256 transition rules, which allows all possible values of the 8-bit input value to be covered within a single hash table. It is assumed that the B-FSM compiler encoded state  $S_0$  with a vector 00h (in hexadecimal notation),  $S_1$  with 02h,  $S_2$  with 01h,  $S_3$  with 10h and  $S_4$  with 04h. States  $S_1$ ,  $S_4$  and  $S_5$  involve at most  $P = 2$  transition rules that can therefore be mapped onto a single line in the transition rule table. For those states, a hash function is used based on a mask equal to 00h, implying that the hash index is extracted solely from the state vector, and consequently, the compiler can directly determine where those rules are mapped in the transition-rule table through the state encoding. States  $S_2$  and  $S_3$  have more than  $P = 2$  rules. For these states, at least one hash index bit needs to be extracted from the input value in order to distribute the corresponding rules over multiple lines in the transition-rule table such that there are at most  $P = 2$  rules mapped on the same line corresponding to a given hash index value. It is assumed that the compiler has selected a hash function based on mask values equal to 0000010b and 0010000b for states  $S_2$  and  $S_3$  respectively. As will be discussed below, these mask values realize the required distribution of the rules over multiple lines in the transition-rule table.

Fig. 2(a) lists transition rules  $R_2$  to  $R_{10}$  with the current state and input vectors that are part of the test part of each rule shown in binary notation. For rules covering multiple input values, e.g., due to character classes, the input value

bits that are variable for the given input value range are represented as 'x'. Note that case-insensitive matches on character values based on ASCII encoding, have an 'x' at bit position 20h which is the differing bit for upper and lowercase alphabetic characters (e.g., the ascii codeword 41h corresponds to 'A' and 61h to 'a'). Underscored bits in Fig. 2(a) correspond to the bit positions within the state and input vectors from which the hash index will be extracted according to (1) based on the mask vector assigned to that state as listed above, with  $state'$  and  $input'$  comprising the least significant seven bits of the state and input vectors.

Fig. 2(b) shows the transition-rule table contents for the above described encoded state vectors and masks. For example, as can be seen in Fig. 2(a), rules  $R_3$  to  $R_5$  are mapped on hash index values for which bits at positions 0 and 2 to 6 are extracted from the encoded state vector of state  $S_2$  (01h) and the bit at position 1 is extracted from the corresponding bit position in the input value as specified by the mask generated for state  $S_2$  (0000010b). In case the latter input bit would equal 0 then only rules  $R_3$  and  $R_4$  can potentially match when the B-FSM is in state  $S_2$ , whereas if it would equal 1 only rules  $R_3$  and  $R_5$  can potentially match. In the former case, equation 1 will result in an index value 1 and consequently rules  $R_3$  and  $R_4$  are mapped on that index value. In the latter case, an index value equal to 3 will be produced by the address generator, and therefore the corresponding transition-rule table line will contain rules  $R_3$  and  $R_5$ .

As can be seen in Fig. 2(b), the transition rules are ordered by priority within each line in the transition-rule table. Therefore, in the case of multiple matching rules, the first matching rule based on that order will have the highest priority, and its result part will be used to update the current state. The above example clearly shows the advantage of direct support for character classes by the B-FSM. In this case, a single transition-rule vector (having a typical size between 4 and 5 bytes) covers multiple input values that are part of a character class. This results in a storage-efficiency improvement over a conventional implementation of a programmable state machine based on a next state table and for which the same next state information would have to be duplicated for all entries related to the same character class. Obviously, the gain will be larger for rules involving larger character classes, e.g., all alphanumeric characters (\w).

This simple example involved only a single pattern and resulted in a transition-rule table that was only partially populated. For typical NIDS applications involving multiple thousands of patterns, substantially larger DFAs will be mapped on the B-FSMs. Our analysis has shown that many states in these large DFAs have relatively few transitions. Taken into account that in typical implementations, the memory width allows to map at least  $P = 3$  or  $P = 4$  transitions on a given line, this means that for many of those states all

transitions can be mapped onto a single line, using a hash function defined by a mask equal to 00h. As can be seen from the previous example, for those states the index bits are extracted directly from the encoded state vector, which means that the compiler has direct control of the placement of those rules in each given transition rule table when performing the state encoding. By exploiting this concept in combination with an efficient clustering of the states, the compiler is able in many cases to realize an optimal or near-optimal population of the transition-rule tables, corresponding to an approximate linear relation between the number of transition rules and the total storage requirements.

### 2.3. B-FSM Engine

Fig. 3 shows a block diagram of the B-FSM engine that implements the rule selection described above. It consists of the following three phases: In phase one, a hash index is derived by the address generator from the current state and input value based on the mask stored in the mask register according to (1). This index is then concatenated with the most significant address bits of the current transition-rule table contained in the table register to form a memory address. In phase two, this memory address is used to access the transition-rule memory and retrieve a transition-rule table entry containing  $P$  transition rules. In phase three, the rule selector will compare the test parts of these  $P$  transition rules in parallel with the actual state and input values using the match conditions corresponding to the rule types, and select the highest-priority matching rule and use that for updating the state, table and mask registers. Note that the table register is updated with a different value in the case of a switch to a different transition-rule table, which happens for transitions to states that have been mapped on a different cluster (see [17]). The output of the B-FSM engine consists of the identifiers of the patterns detected. These identifiers are determined externally to the phases described above by a lookup on the current table and state vectors using a so-called 'result table' not shown in Fig. 3 (see Section V.-B in [17]).

Transition rules involving a wildcard condition for the current state, such as rules  $R_0$  and  $R_1$  in Fig. 1(b), will be handled differently for storage-efficiency reasons, and are stored in the so-called 'default rule table' shown in Fig. 3. Because these rules only depend on the input value, a simple table lookup can be used to determine the highest-priority matching rule which is denoted as 'default rule'. The B-FSM compiler ensures that the transition rules stored in the default rule memory have a lower priority than those stored in the transition-rule memory. Based on that property, the rule selector will select the default rule as the highest-priority matching rule if no matching rule has been found in the transition-rule memory.

Fig. 4 shows the format in which transition rules are

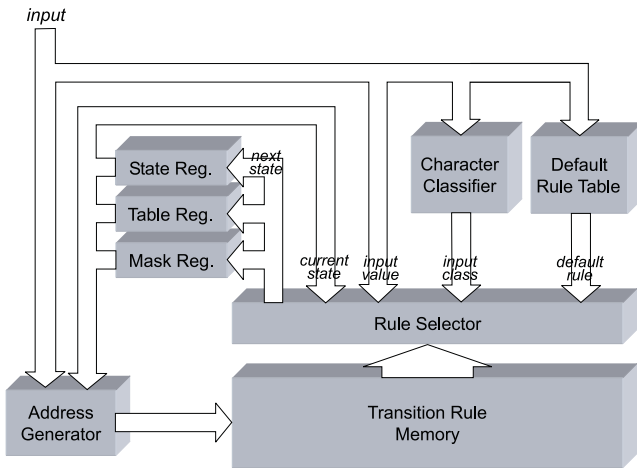


Fig. 3. B-FSM with programmable character classifier.

	test part			result part		
rule type	current state	input / class	next state	table address	mask	

Fig. 4. Transition-rule vector.

stored in the transition-rule memory. The rule-type field includes the definition of the match condition on the input value, which can be exact match, case-insensitive match, class match or don't care, either negated or not. The current-state field specifies the exact-match condition for the current state. The input/class field contains the operand value used to test the input value. For class-match conditions it contains a class identifier. The result part of the transition-rule vector contains the next state vector, table address and mask vectors that will be used to update the corresponding registers in Fig. 3 if the rule is selected by the rule selector.

The B-FSM extension for supporting character classes is based on the programmable character classifier function shown in Fig. 3, which for each input value determines the class(es) to which it belongs. This class information is then used by the rule selector to evaluate class-match conditions. For input streams involving a fixed-length character encoding with up to 8 bits per character, the character classifier function can be implemented as a simple lookup table with 256 entries, containing the class information for each character. The B-FSM scheme supports various ways to encode and test the character class information, including overlapping and non-overlapping character classes.

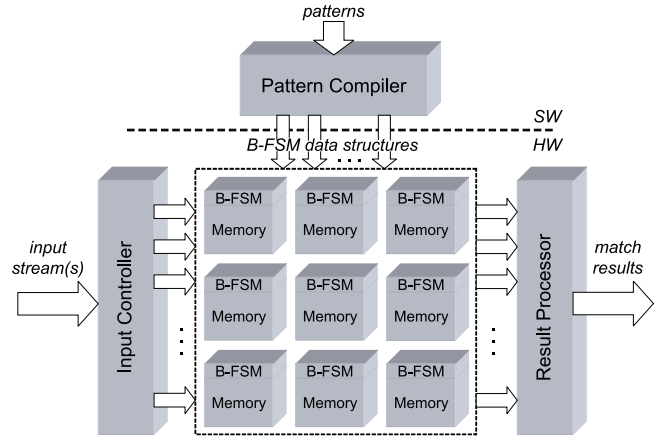


Fig. 5. Pattern-matching engine and compiler.

### 3. PATTERN-MATCHING ENGINE

#### 3.1. Match Engine

The pattern-matching engine is composed of an array of multiple B-FSM engines as shown in Fig. 5. The array typically contains between 4 and 32 B-FSMs, depending on the implementation technology and match requirements. The array can scan multiple input streams in parallel, with each input stream being scanned by a selected group of the B-FSMs and each B-FSM taking care of a disjoint subset of the patterns involved in the scan operation. The match results of the B-FSM array, i.e., the identifiers of detected patterns and the corresponding offsets, are post-processed by a result processor that generates the final match results that are provided to the application (see [17]).

#### 3.2. Pattern Compiler

The pattern compiler performs three functions: (1) it distributes the patterns over the B-FSM engines, (2) converts the patterns allocated to each B-FSM into a DFA description, and (3) compiles these DFA descriptions into the hash-table-based data structure that is directly executed by the B-FSM engines. The function implementing the third step is also denoted as B-FSM compiler.

The second step can be implemented using existing methods for compiling patterns into DFAs, for example, using the method in [23]. For more details on the steps (2) and (3), the reader is referred to [17].

#### 3.3. Pattern Distribution

The pattern-distribution function distributes the patterns of a given set over multiple DFAs. Each DFA will be executed by a B-FSM engine. The main objective of the distribution is to minimize the total storage requirements and thus increase

scan performance. It allows to keep larger pattern sets in the limited fast on-chip memory. If not sufficient on-chip memory is available, a larger fraction of the entire state diagram can be cached and the miss rate is reduced.

For large pattern sets, a brute-force solution to this problem is not feasible because of the excessive number of possible combinations. Moreover, to evaluate the memory consumption of a given distribution, the entire compilation needs to be executed. Such a compilation is relatively time consuming, taking in the order of seconds to minutes.

To obtain an optimal or near-optimal distribution within a reasonable time, we first approximate the distribution problem by a energy minimization problem and then apply a simulated annealing optimization strategy [24] to search the space of possible distribution functions.

The energy function is based on a metric that measures the compatibility of a pair of regular expressions as the increment (or decrement) of the number of transitions in the combined DFA representation compared with the sum of the transitions in the two DFAs for the individual regular expression. These metrics are precalculated at the beginning of the distribution process. To calculate the energy function for a given distribution, all the pair-wise metrics for regular expressions which are mapped to the same DFA are accumulated. Distributions with low energy can be expected to yield more compact DFAs. This approximation allows a more rapid assessment of the quality of a distribution compared to full compilation. However, the search space is still as large as before.

To find a distribution with low (ideally minimal) energy without trying all possible distributions, a simulated annealing optimizer is used. This optimizer tries to minimize the energy function iteratively. It is initialized with a random distribution. In each iteration, the distribution is slightly modified by remapping up to three patterns to different engines. The new distribution is accepted with a certain probability. This probability is higher the more the energy has decreased compared to the old distribution. If the energy has increased, the distribution may still be accepted, although the probability is lower. Such steps to higher energy values allow the optimizer to escape local optima.

To further reduce issues with local optima, the annealing process is started multiple times, each time with a different initial distribution. The distribution yielding the overall minimal energy is selected.

## 4. EXPERIMENTAL RESULTS

We have performed several experiments with proprietary pattern sets for intrusion detection. The next two subsections present the storage-efficiency improvements that were obtained by the direct support for flexible match conditions at the level of the B-FSM engines and by the novel distribution

**Table 1.** Storage efficiency impact of match conditions.

patterns	size	transition rules			
		A	B	C	D
set 1	800	7.0K	7.0K	4.3K	3.9K
set 2	421	32.0K	32.0K	23.3K	21.8K
set 3	275	13.3K	13.3K	9.8K	4.4K
set 4	245	15.6K	15.5K	11.0K	10.4K
set 5	180	6.5K	6.5K	5.0K	5.0K
set 6	199	5.0K	5.0K	3.5K	2.3K
set 7	89	42.0K	41.8K	31.0K	9.9K

function. The third subsection presents performance numbers based on synthesis experiments.

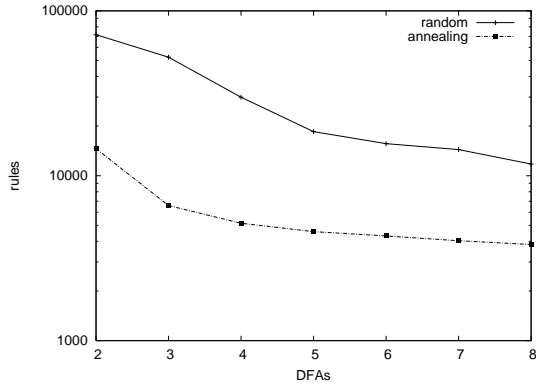
### 4.1. Character-Class support

The gain in storage efficiency achieved by direct support of character classes and other flexible match conditions in the extended B-FSMs was evaluated by performing several experiments in which we enabled and disabled support of these match conditions in the B-FSM compiler. Experiments were done for four configurations involving support for the following combinations of match conditions on the input value by the rule selector in the B-FSM:

- A) support for exact-match and don't care conditions
- B) option A plus support for negation
- C) option B plus support for case-insensitive matching
- D) option C plus support for character classes

These experiments have been performed using pattern sets derived from commercial NIDSs. Because of confidentiality reasons, only the results can be presented, but no specific characteristics of those pattern sets except for the number of patterns.

Table 1 lists the results as the number of transition rules generated by the pattern compiler. Although not shown, for these experiments the storage requirements (i.e., number of cluster tables) grew approximately linearly with the number of rules. Table 1 shows that the actual impact strongly varies per pattern-set. For the given pattern sets, on average a storage reduction of just above 50% was achieved by the most flexible match conditions (option D) over the least flexible match conditions (option A), with the largest gain being 76% achieved for pattern set 7, and the smallest gain 23% for pattern set 5. Support for exact match, don't care, negation and case-insensitive matching but not for character classes (option C) results in an average gain of about 27% for the tested pattern sets over option A.



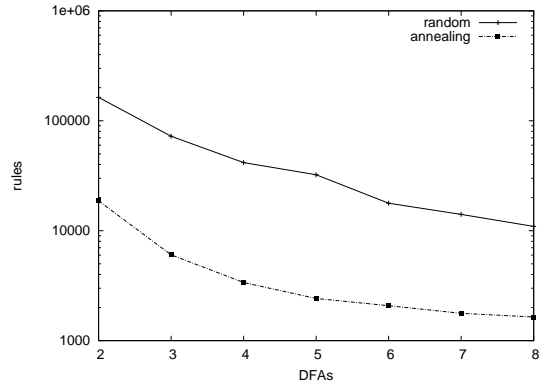
**Fig. 6.** Set 1 contains 184 expressions extracted from a proprietary intrusion-detection system.

#### 4.2. Distribution Performance

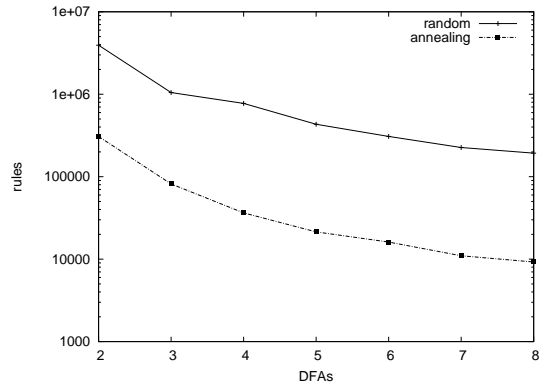
We evaluated the distribution function using three different sets of regular expressions. Set 1 contains 184 regular expressions extracted from a proprietary intrusion-detection system. Set 2 consists of 101 expressions from the L7-filter, a classifier for Linux’s Netfilter [22], and set 3 is a compilation of 166 patterns, which are part of a proprietary NIDS. Our experiments show how the total complexity of the state diagram (measured in the number of state transitions after base class mapping) is reduced by using additional DFAs. Although this effect can also be observed if the regular expressions are distributed randomly to the DFAs, our distribution strategy is significantly more efficient than a random distribution. For example, improvements of up to 20× can be achieved for set 3 on eight engines (Fig. 8). In general, our distribution yields a reduction of the number of transition rules of about an order of magnitude (Figs. 6 and 7). The numbers shown for random distributions are the median values of ten different random distributions.

#### 4.3. Pattern Scanner Performance

Several FPGA prototypes of the pattern-matching engine have been implemented using various Xilinx Virtex-4 devices. The prototypes included a B-FSM implementation based on six Block RAMs providing a total storage capacity of about 13 KB: four Block RAMs (9 KB) were used for the transition-rule memory, one Block RAM for a combined implementation of the (ASCII-based) character classifier and the default rule table as shown in Fig. 3, and one Block RAM for a so-called result table. An XC4VLX160-based prototype containing an array with 16 B-FSMs ran at 125 MHz, consuming 33% of the available Block RAMs. With each transition rule handling one byte per clock cycle, this resulted in a fixed processing rate of 1 Gb/s for each B-



**Fig. 7.** Set 2 consists of 101 regular expressions from the Linux Application Protocol Classifier (L7-filter).



**Fig. 8.** Set 3 consists of 166 regular expressions and is part of a proprietary NIDS.

FSM. This corresponds to a peak aggregate throughput of 16 Gb/s for the B-FSM array for the case that all B-FSMs work on different input streams, and an aggregate throughput of 4 Gb/s for the case that groups of four B-FSMs work on the same input stream.

Synthesis experiments for 45-nm CMOS technology have shown that a dual-threaded B-FSM implementation with dedicated SRAM can run at speeds beyond 2 GHz while processing one byte per clock cycle. This corresponds to a scan rate of at least 16 Gb/s per B-FSM. Consequently an array with 16 B-FSMs in which groups of four B-FSMs operate on a single input stream achieves a peak scan rate of at least 64 Gb/s.

## 5. CONCLUSIONS

In this paper, we have presented a pattern-matching engine for advanced regular-expression matching. Our engine builds

on an extended version of the B-FSM scheme, providing more flexible match conditions at the state-transition level, including direct support for character classes. This enables a substantially more compact data structure. Furthermore, a novel pattern-distribution algorithm based on simulated annealing was presented, which optimizes the storage efficiency also at the pattern level.

Despite the increased flexibility in match functions as indicated above, the relatively simple address-generation and rule-selection functions executed by the B-FSM engines enable fast FPGA and ASIC implementations of our B-FSM engine that achieve scan rates on the order of 1 Gb/s and 16 Gb/s per B-FSM, respectively, when running out of dedicated SRAMs. When operating these SRAMs as a cache, the improvements in storage efficiency enable high cache hit rates, thus allowing to scale to very large pattern collections while maintaining high scan rates.

## 6. ACKNOWLEDGEMENT

The authors would like to thank Charlotte Bolliger for her excellent help with the preparation of this manuscript.

## 7. REFERENCES

- [1] M. Roesch, "SNORT - Lightweight intrusion detection for networks," *Proc. LISA 99: USENIX 13th Systems Administration Conference*, pp. 229–238, November 1999.
- [2] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.
- [3] R.S. Boyer and J.S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.
- [4] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," *Technical report TR-94-17*, Department of Computer Science, University of Arizona, May 1994.
- [5] Z. K. Baker and V. K. Prasanna, "A methodology for synthesis of efficient intrusion detection systems on FPGAs," in *Proc. FCCM '04*, pp. 135–144, IEEE, 2004.
- [6] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.
- [7] I. Sourdis and D. Pnevmatikatos, "Predecoded CAMs for efficient and high-speed NIDS pattern matching," *Proc. IEEE FCCM*, pp. 21–23, April 2004.
- [8] Y. H. Cho and W. H. Mangione-Smith, "A pattern matching coprocessor for network security," in *Proc. DAC '05*, pp. 234–239, ACM, 2005.
- [9] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. ISCA '05*, pp. 112–122, IEEE, 2005.
- [10] B. Commentz-Walter, "A string matching algorithm fast on the average," *Proc. of the 6th Colloquium, on Automata, Languages and Programming*, pp. 118–132, July 1979.
- [11] R. Sidhu and V.K. Prasanna, "Fast regular expression matching using FPGAs," *Proc. IEEE FCCM*, pp. 227–238, 2001.
- [12] B.L. Hutchings, R. Frankling, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," *Proc. IEEE FCCM*, pp. 111–120, April 2002.
- [13] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos, "Implementation of a content-scanning module for an Internet firewall," *Proc. IEEE FCCM*, pp. 31–38, April 2003.
- [14] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of pattern matching circuits for regular expression on FPGA," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 12, pp. 1303–1310, 2007.
- [15] J. Bispo, I. Sourdis, J.M.P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," *Proc. IEEE FPT*, 2006.
- [16] J. van Lunteren, "Searching very large routing tables in wide embedded memory," *Proc. IEEE Globecom*, vol. 3, pp. 1615–1619, November 2001.
- [17] J. van Lunteren, "High-performance pattern-matching for intrusion detection," *Proc. IEEE INFOCOM*, April 2006.
- [18] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ANCS '06*, pp. 93–102, ACM, 2006.
- [19] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *ANCS '07*, pp. 155–164, ACM, 2007.
- [20] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *Proc. SIGCOMM '08*, pp. 207–218, ACM, 2008.
- [21] P. Piyachon and Y. Luo, "Design of high performance pattern matching engine through compact deterministic finite automata," in *Proc. DAC '08*, pp. 852–857, ACM, 2008.
- [22] "Application layer packet classifier for linux." <http://l7-filter.sourceforge.net/>.
- [23] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, 1968.
- [24] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.