Differential Encoding of DFAs for Fast Regular Expression Matching

Domenico Ficara, *Member, IEEE*, Andrea Di Pietro, *Student Member, IEEE*, Stefano Giordano, *Senior Member, IEEE*, Gregorio Procissi, *Member, IEEE*, Fabio Vitucci, *Member, IEEE*, and Gianni Antichi, *Member, IEEE*

***Abstract*—Deep packet inspection is a fundamental task to improve network security and provide application-specific services. State-of-the-art systems adopt regular expressions due to their high expressive power. They are typically matched through deterministic finite automata (DFAs), but large rule sets need a memory amount that turns out to be too large for practical implementation. Many recent works have proposed improvements to address this issue, but they increase the number of transitions (and then of memory accesses) per character. This paper presents a new representation for DFAs, orthogonal to most of the previous solutions, called delta finite automata (δFA), which considerably reduces states and transitions while preserving a transition per character only, thus allowing fast matching. A further optimization exploits Nth order relationships within the DFA by adopting the concept of "temporary transitions."**

***Index Terms*—Deep packet inspection, differential encoding, finite automata (FAs), pattern matching, regular expressions.**

I. Introduction

NOWADAYS, pattern matching is required in an increasing number of network devices (intrusion prevention systems, traffic monitors, application recognition systems). Traditionally, the inspection was done with common multiple-string matching algorithms, but state-of-the-art systems use regular expressions (regexes) [1] to describe signature sets. They are adopted by well-known tools, such as Snort [2] and Bro [3], and in devices by different vendors such as Cisco [4].

Typically, finite automata (FAs) are employed to implement regexes matching. In particular, deterministic FAs (DFAs) allow for fast matching by requiring one state transition per character, while nondeterministic FAs (NFAs) need more transitions per character. The drawback of DFAs is that for the current regex sets they require an excessive amount of memory. Therefore, many works have been recently presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regex sets. However, most of these solutions can require more than one memory reference, thus lowering search speed.

Manuscript received March 12, 2009; revised August 07, 2009 and July 01, 2010; accepted September 27, 2010; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor T. Wolf. Date of publication November 22, 2010; date of current version June 15, 2011.

The authors are with Dipartimento di Ingegneria dell'Informazione, Università di Pisa, 56122 Pisa, Italy (e-mail: Domenico.Ficara@iet.unipi.it; Andrea.DiPietro@iet.unipi.it; Stefano.Giordano@iet.unipi.it; Gregorio.Procissi@iet.unipi.it; Fabio.Vitucci@iet.unipi.it; Gianni.Antichi@iet.unipi.it)

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2010.2089639

This paper, an extension of the work in [5], introduces a novel compact representation scheme (named δFA), which is based on the observation that since most adjacent states share several common transitions, it is possible to delete most of them by taking into account the different ones only (the δ in δFA just emphasizes that it focuses on the differences between adjacent states). Reducing the redundancy of transitions appears to be very appealing since the recent general trend in the proposals for compact and fast DFAs construction (see Section II) suggests that the information should be moved toward edges rather than states. In particular, our idea comes from D^{2}FA [6], which introduces default transitions (and a "path delay") for this purpose.

With respect to [5], we add the concept of "temporary transition," thus improving δFA: Instead of specifying the transition set of a state with respect to its direct parents (also defined 1-step ancestors), relaxing this requirement to the adoption of N-step "ancestors" increases the chances of compression. As we will show in the following, the best approach to exploit this Nth-order dependence is to define the transitions of the states between ancestors and child as "temporary." This, however, introduces a new problem during the construction process: The optimal construction (in terms of memory or transition reduction) appears to be an NP-complete problem. Therefore, a direct and oblivious approach is chosen for simplicity. Results on real rule-sets show that our simple approach does not differ significantly from the optimal (if ever reachable) construction. Since this optimized technique is an extension to δFA that exploits Nth-order dependence, we name it δ^NFA.

While many other proposed algorithms for DFA compression require more transitions per character, δFA and δ^NFA examine one state per character only, thus reducing the number of memory accesses and speeding up the overall lookup process (as already done in [7] and [8], although with some limitations). This improvement comes at the cost of wider memory accesses (with respect to previous schemes). However, it is quite common today for DRAM memories to allow for large memory reads (easily up to 512 bits). For this reason, if compared to D^{2}FA, where a state may not store a transition and we could need to *sequentially* perform other memory reads (even if only 32 bits long) to learn the next state, our approach of using a single *wide* memory access can provide better timing performances.

Since a FA-walk is, by nature, a process paced by slow memory accesses and very short computation (if any), reducing the number of memory references is crucial to the development of a faster system. In a threaded architecture, usually each thread performs a FA-walk on a different data packet. When a thread issues a memory read, other threads take control and perform their operations (again few computations and a memory read). However, because of the very short computations of regular DFAs and because of the large latency of DRAMs, it is quite common for the processor to be in idle state for a consistent percentage of time. This observation introduces a degree of freedom that we exploit in this work: While reading from DRAM, our code executes other operations on a fast and small local memory.

Moreover, our techniques share the same property of many other proposed schemes: They are orthogonal to several previous algorithms (even the most recent XFAs [9] and H-cFA [10]), thus allowing for higher compression rates.

In addition, a new encoding scheme for states is proposed that exploits the association of many states with a few input characters. Such a compression scheme can be efficiently integrated into our algorithm, allowing a further memory reduction with a negligible increase in the state lookup time.

In summary, the main contributions of this paper are:

- a novel compact representation of DFA states (δFA) that allows for iterative reduction of the number of transitions and for faster string matching;
- an optimization (δ^NFA) that exploits Nth-order relationships within the DFA, thus saving further memory;
- a new state encoding scheme based on input characters.

The remainder of the paper is organized as follows. In Section II, related works about DFAs are discussed. Section III accurately describes δFA, by starting from a motivating example, Section IV presents the optimization of δ^NFA, and Section V proves the integration of the proposed schemes with the previous ones. Then, in Sections VI and VIII, the novel encoding scheme for states and the integration with δFA are illustrated. Finally, Section IX presents the experimental results.

II. Related Work

Deep packet inspection consists of processing the packet payload and identifying a set of predefined patterns. Many algorithms of standard pattern matching have been proposed [11]–[14], but nowadays state-of-the-art systems replace string sets with the more powerful regular expressions.

Traditionally, in order to search for regexes, DFAs and NFAs are exploited. While DFAs have predictable (yet large) memory consumption and fixed memory references per character, NFAs consume lower memory, but may require several memory accesses per symbol in order to track all the states that are active at a given time. Because of their parallel nature, NFAs are usually the preferred solution in hardware platforms such as FPGAs, as shown first in [15]. However, for software-based systems such as network processors, the current trend in industry is to use DFAs to represent regular expressions because of their deterministic behavior, which is appealing especially in parallel systems. Indeed, in parallel platforms as modern network devices, the overall performance of packet processing is heavily affected by the processing times of the slowest component. Therefore, industries are adopting for pattern-matching deterministic solutions as DFAs, thus a large part of the academic research is following the same trend.

However, it has been proved that DFAs corresponding to a large set of regexes can blow up in space, and many recent works have been presented with the aim of reducing their memory footprint. The size problem is due to two main reasons: encoding (a naive encoding of DFA states is largely redundant) and state-explosion (when combining different regexes, the number of states in the resulting DFA can increase exponentially). Our contribution proposes a solution to the first problem.

DFA encoding is also discussed in [6], where Kumar *et al.* introduce the delayed input DFA ($\mathrm{D^2FA}$), a new representation for regexes that reduces space requirements. Since many states in DFAs have similar sets of outgoing transitions, redundant transitions can be replaced with a single default one. The drawback of this approach is the traversal of multiple states when processing a single input character, which entails a memory bandwidth increase to evaluate regular expressions. However, a bound B on the number of default transitions to be taken by a single character in $\mathrm{D^2FA}$ can be defined—generally, larger values of B (hence many memory accesses per byte) correspond to higher memory compression.

To address the issues of too many accesses per character and slow lookup in $\mathrm{D^2FA}$, an improved algorithm is introduced in [16] (we will call it Bec-Cro) that achieves lower provable bounds on memory bandwidth. This work is based on the observation that all regexes evaluations begin at a single starting state, and the vast majority of transitions among states lead back either to the starting state or its near neighbors. Such an automaton requires at most $2N$ state traversals when processing a string of length N. In this paper, we will show that the memory compression of $\mathrm{D^2FA}$ can be obtained also with a single memory access per character, as shown also by [7] and [8].

In particular, the work in [7] proposes a technique that allows nonequivalent states to be merged, thanks to a scheme where the transitions in the DFA are labeled. The authors merge states with common destinations regardless of the characters which lead those transitions (unlike $\mathrm{D^2FA}$), creating opportunities for more merging and thus achieving higher memory reduction. Moreover, the authors regain the idea of bitmaps for compression purposes, which necessarily increase the lookup cost by requiring two subsequent memory accesses (first to the bitmap, and then to the transition set).

Instead, Kumar *et al.* [8] show how to increase the speed of $\mathrm{D^2FA}$s by storing a large amount of information (on subsequent reachable transitions) on the edges in an automaton called $\mathrm{CD^2FA}$. While this reduces the lookup cost of $\mathrm{D^2FA}$s, it also requires a construction based on perfect hash functions that may be time-consuming. The idea of storing more information on the edges appears to be a general trend in the literature, and it has been proposed in different ways: In [8], transitions carry data on the next reachable nodes; in [7], edges have different labels; in [10], a sort of history buffer (i.e., a small and fast cache) stores additional information in order to efficiently follow multiple partially matching signatures, thus yielding the state blow-up; in [9], a finite scratch memory is used to remember various types of information relevant to the progress of signature matching (e.g., counters of characters) in order to keep the transition history and reduce the number of states.

Because of a patent protecting the state-merging work [7], this work is not considered in our evaluation tests.

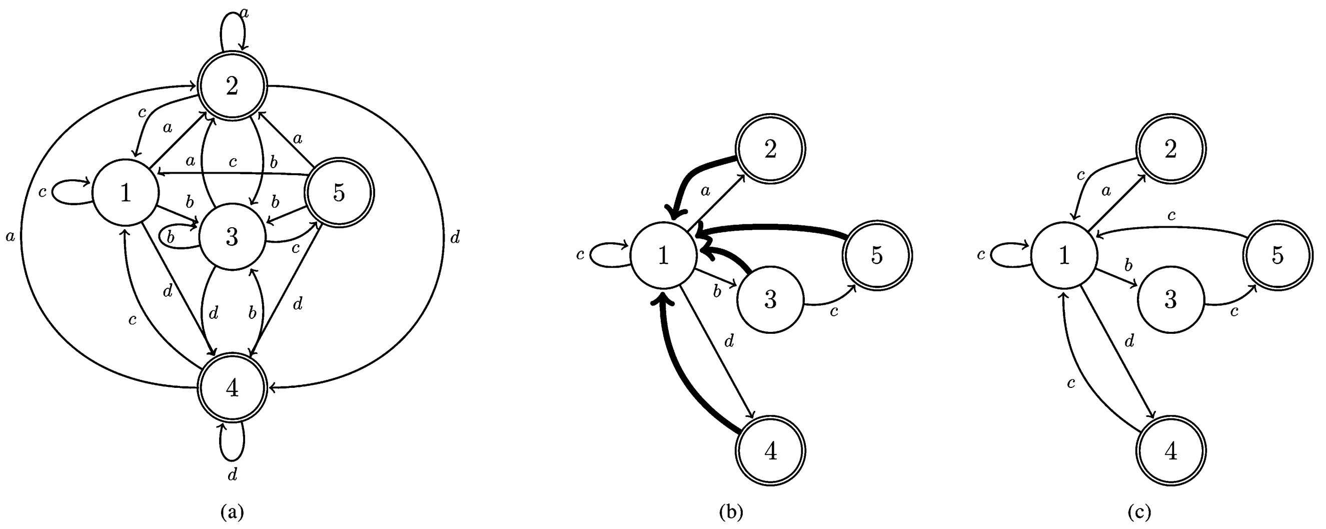

Fig. 1. Automata recognizing (a^{+}), $(b^{+}c)$, and $(c^{*}d^{+})$. (a) DFA. (b) D^{2}FA. (c) δFA.

Also, the issue of state explosion has been treated in recent works. For instance, observing that NFAs can alleviate the memory problem but lead to a potentially large bandwidth requirement, in [17] a hybrid DFA-NFA solution is proposed: When constructing the hybrid-DFA, any nodes that would contribute to state explosion retain an NFA encoding, while the remaining ones are transformed into DFA nodes. The target is a data structure with a size nearly that of an NFA, but with the predictable and small memory bandwidth requirements of a DFA.

III. Delta Finite Automaton: δFA

As discussed, several works in the recent years have focused on memory reduction of DFAs by trading size for number of memory accesses. The most important and cited example of such a technique is D^{2}FA [6], where an input character (hereafter simply "char") can require a (configurable) number of additional steps through the automaton before reaching the right state.

A. Motivating Example

In this section, we introduce δFA, a D^{2}FA-inspired automaton that preserves the advantages of D^{2}FA while requiring a single memory access per input char. In order to make clearer the rationale behind δFA construction and the differences with D^{2}FA, we start by analyzing the same example of [6]. Fig. 1(a) represents a DFA on the alphabet $\{a, b, c, d\}$ that recognizes the regular expressions (a^{+}), $(b^{+}c)$ and $(c^{*}d^{+})$.

In Fig. 1(b), the D^{2}FA for the same set of regexes is shown. The main idea is to reduce the memory footprint of states by storing only a limited number of transitions for each state and by taking a default transition for all input chars for which a transition is not defined. When, for example, in Fig. 1(b) the state machine is in state 3 and the input is d, the default transition to state 1 is taken. State 1 has a specified transition for char d, therefore we jump to state 4 (as in the standard DFA).

In this example, taking a default transition costs one more hop (one more memory access) for a single input char. However, it may happen that also after taking a default transition, the destination state for the input char is not specified and another default transition must be taken, and so on. In the example, the number of transitions was reduced to nine in the D^{2}FA (while the DFA has 20 edges), thus achieving a remarkable compression.

However, observing the graph in Fig. 1(a), it is evident that most transitions for a given input lead to the same state, regardless of the starting state. In particular, adjacent states share the majority of the next-hop states associated with the same input chars. Then, if we jump from state 1 to state 2 and we "remember" (in a local memory) the entire transition set of 1, we will already know all the transitions defined in 2 (which has the same set of 1). This means that state 2 can be described with a very small amount of bits. Instead, if we jump from state 1 to 3, and the next input char is c, the transition will not be the same as the one that c produces starting from 1. Then, state 3 will have to specify its transition for c.

The result of what we have just described is depicted in Fig. 1(c) (except for the local transition set), which is the δFA equivalent to the DFA in Fig. 1(a). We have eight edges only in the graph, and every input char requires *a single state traversal.*

B. Main Idea of δFA

As shown in the previous section, the target of δFA is to obtain a similar compression as D^{2}FA without giving up the *single state traversal per character* of DFA. The idea of δFA comes from the following observations on D^{2}FAs and DFAs.

- As shown in [16], most default transitions are directed to states closer to the initial state.
- In a DFA, most transitions for a given input char are directed to the same state.

Therefore, it becomes evident that most adjacent states share a large part of the same transitions. Thus, we can store only the differences between adjacent states. This requires, however, the introduction of a supplementary structure that locally stores the transition set of the current state. The main idea is to let this local transition set evolve as a new state is reached. If there is no difference with the previous state for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. In all cases, as a new state is read, the local transition set is updated with all the stored transitions of the state. The δFA in Fig. 1(c) only stores the transitions that *must* be defined for each state in the original DFA.

C. Construction

In Algorithm 1, the process for creating a δFA from an N-states DFA (for a char set of C elements) is shown. The algorithm works with the *transition table* $t[s,c]$ of the input DFA (i.e., an $N \times C$ matrix that has a row per state and where the ith item in a given row stores the state number to reach upon the reading of input char i). The final result is a "compressible" transition table $t_c[s,c]$ that stores the transitions required by the δFA only. All the other cells of the $t_c[s,c]$ matrix are filled with the special LOCAL_TX symbol and can be simply eliminated through a bitmap, as suggested in [14]. The details of our implementation can be found in Section VIII.

Algorithm 1: Creation of the transition table t_c of a δFA.

```
1: for $c \leftarrow 1, C$ do
2:     $t_c[1,c] \leftarrow t[1,c]$
3: for $s \leftarrow 2, N$ do
4:     for $c \leftarrow 1, C$ do
5:         $t_c[s,c] \leftarrow \text{EMPTY}$
6: for $s_{\text{parent}} \leftarrow 1, N$ do
7:     for $c \leftarrow 1, C$ do
8:         $s_{\text{child}} \leftarrow t[s_{\text{parent}}, c]$
9:         for $y \leftarrow 1, C$ do
10:            if $t[s_{\text{parent}}, y] \neq t[s_{\text{child}}, y]$ then
11:                $t_c[s_{\text{child}}, y] \leftarrow t[s_{\text{child}}, y])$
12:            else
13:            If $t_c[s_{\text{child}}, y] == \text{EMPTY}$ then
14:                $t_c[s_{\text{child}}, y] \leftarrow \text{LOCAL\_TX}$
```

The construction requires a step for each transition (C) of each pair of adjacent states $(N \times C)$ in the input DFA, thus it costs $O(N \times C^2)$ in terms of time complexity. The space complexity is $O(N \times C)$ because the structure upon which the algorithm works is another $N \times C$ matrix. In detail, the construction algorithm first initializes the t_c matrix with EMPTY symbols and copies the first (root) state of the original DFA in the t_c (it will act as base for subsequently storing the differences). Then, the algorithm observes the states in the original DFA one at a time. It refers to the observed state as *parent*. Then, it checks the *children* states (i.e., the states reached in one transition from parent state). If, for an input char c, the child state stores a different transition than the one associated with any of its parent nodes, we cannot exploit the knowledge we have from the previous state, and this transition must be stored in the t_c table. On the other hand, when all of the states that lead to the child state for a given character share the same transition, then we can omit to store that transition. In Algorithm 1, this is done by using the special symbol LOCAL_TX.

After the construction, since the number of transitions per state is significantly reduced, it may happen that some of the states have the same identical transition set. If we find j identical states, we can simply store one of them, delete the other $j-1$, and substitute all the references to those with the single state we left. Notice that this operation again creates the opportunity for a new state-number reduction because the substitution of state references makes it more probable for two or more states to share the same transition set. Hence, we iterate the process until the duplicate states end.

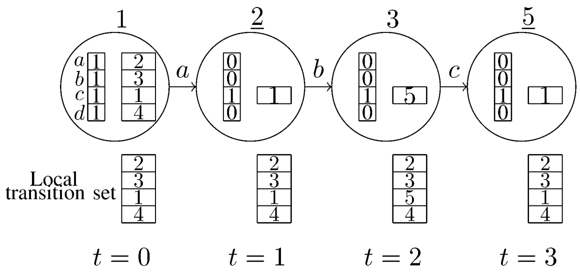

Fig. 2. δFA internals: a lookup example.

D. Lookup

The lookup in a δFA is shown in Algorithm 2. First, the current state must be read with its whole transition set. Then, it is used to update the local transition set t_{loc}: For each transition defined in the set read from the state, we update the corresponding entry in the local storage. This procedure comes at virtually no cost since it requires a number of operations on a fast local memory and its execution is easily masked in threaded systems or hardware implementations. Finally, the next state s_{next} is computed by simply observing the proper entry in the local storage t_{loc}.

While the need to read the whole transition set may imply more than one memory access, wide memory reads are quite common today for DRAMs, as discussed in the introduction, and the adoption of δFA and a novel encoding scheme (that we discuss in Section VII) allows to solve this issue.

The lookup requires a maximum of C elementary operations (such as shifts and logic AND or popcounts), one for each entry to update. However, in our experiments, the number of updates per state is around 10. Even if the actual processing delay strictly depends on many factors (such as clock speed and instruction set), in most cases the computational delay is negligible with respect to the memory access latency.

Algorithm 2: Pseudocode for the lookup in a δFA. The current state is s and the input char is c.

```
procedure Lookup$(s, c)$
1: read$(s)$
2: for $i \leftarrow 1, C$ do
3:     if$t_c[s,i] \neq \text{LOCAL\_TX}$ then
4:         $t_{\text{loc}}[i] \leftarrow t_c[s,i]$
5: $s_{\text{next}} \leftarrow t_{\text{loc}}[c]$
6: return $s_{\text{next}}$
```

In Fig. 2, we show the transitions taken by the δFA in Fig. 1(c) on the input string *abc*: A circle represents a state, and

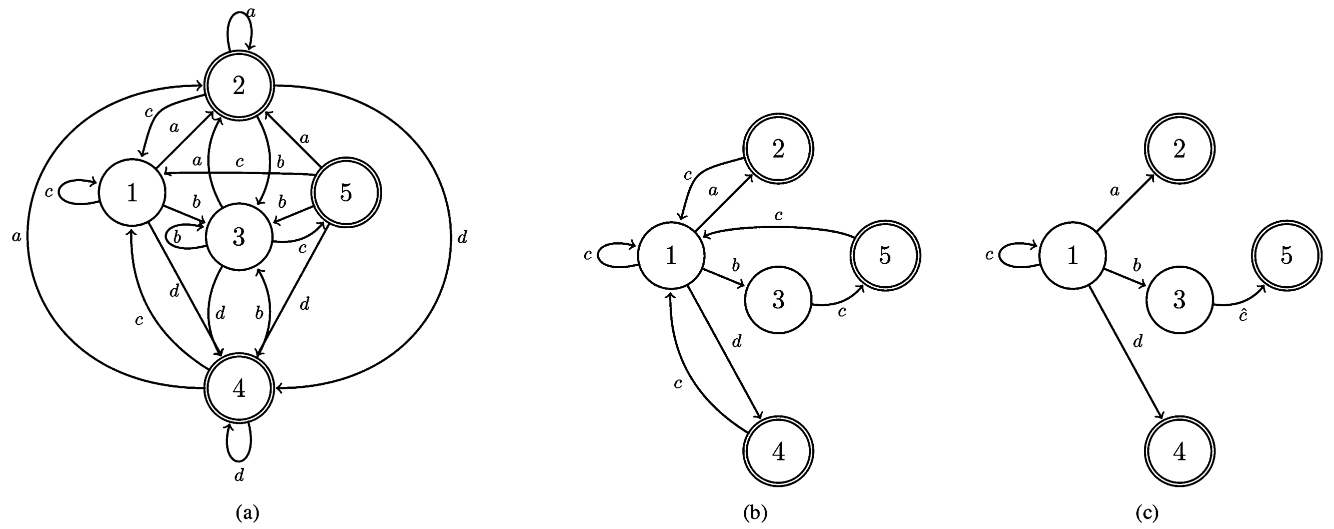

Fig. 3. Automata recognizing (a^+), (b^+c), and (c^*d^+). (a) DFA. (b) δFA. (c) δ^NFA.

its internals include the transition set and a bitmap to indicate which transitions are specified, as defined during construction. We start $(t = 0)$ in state 1, which has a fully specified transition set. This is copied into the local transition set (below). Then we read the input char a and move $(t = 1)$ to state 2, which specifies a single transition toward state 1 on input char c. This is also an accepting state (underlined in figure). Then, we read b and move to state 3. Note that the transition to be taken now is not specified within state 2, but it is in our local transition set. Again, state 3 has a single transition specified that this time changes the corresponding one in the local transition set. As we read c, we move to state 5, which is again accepting.

IV. NTH-ORDER DELTA FINITE AUTOMATON: δ^NFA

Instead of specifying the transition set of a state with respect to its direct parents, the adoption of N-step "ancestors" increases the chances of compression. This is the approach of δ^NFA, which is an improved version of δFA.

A. *Main Idea*

Let us use again the DFA in [6]. Although the δFA in Fig. 3(b) shows a remarkable saving in terms of transitions with respect to the standard DFA, its main assumption (all parents must share the same transition for a given character) somewhat limits the effectiveness of the compression. In the example, all the transitions for character c are specified (and hence stored) for all the five states because of a single state 3 that defines a different transition (the transition for c is directed to state 1 for states 1, 2, 4, and 5, while 3 defines an edge to 5). Notice that this is due to the strict definition of δFA rules that do not "see" further than a single hop: The transition set of a state is stored as the difference with respect to all its direct parents.

Intuitively, just as a D^{2}FA with long default-transitions paths compresses better than a bounded D^{2}FA with $B = 2$ [6], by relaxing the definition of "parents" to "ancestors" (i.e., N-step neighbor nodes), the effectiveness of the δFA approach increases because of the larger number of possibilities.

However, a blind adoption of this concept does not provide better results. As an example, in Fig. 3(b), defining the transitions for c as the difference with respect to all the "grandparents" (second-order ancestors) still would not allow to eliminate any new transition. Moreover, such a scheme would require to store two local transition sets (doubling the amount of local memory needed).

A better approach is, instead, to define the transition for c in state 3 as "temporary" in the sense that it does not get stored in the local transition set. In this way, we force the transition to be defined uniquely within state 3 and not to affect its children. This means that, whenever we read state 3, the transition for c in the local transition set is not updated, but it remains as it was in its parents. Then, we can avoid storing the transitions for c in states 2, 4, and 5, as shown in Fig. 3(c), where the temporary transition is signaled with $\hat{c}$.

Notice that by defining temporary transitions, we efficiently exploit Nth-order relationships among states, without incurring in the need for N-times larger local memories.

B. *Lookup*

The lookup in a δ^NFA differs very slightly from that of δFA. The only difference concerns temporary transitions, which are valid within their state, but they are not stored in the local transition set. Therefore, the lookup time complexity of δ^NFA is practically the same as that of δFA. Fig. 4 shows also an example of the lookup process for a δ^NFA. At $t = 0$, the whole transition set of state 1 is copied into the local transition set. Then, by char a, we move $(t = 1)$ to state 2, which does not specify any transition. When we read b $(t = 2)$, we move to state 3, where a temporary transition (dashed box) is specified; this transition is valid only within state 3. Finally, $(t = 3)$ we read c, take the temporary transition, and end up in state 5.

C. *Construction*

The construction process of the δ^NFA requires the corresponding δFA to be constructed beforehand and used as input. Then, the process works by recognizing subsets of nodes where

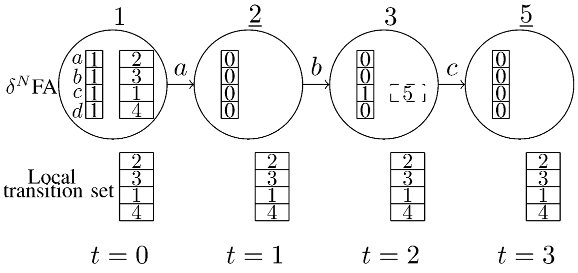

Fig. 4. δ^NFA internals: a lookup example.

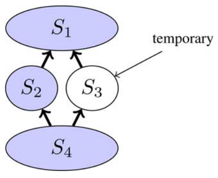

Fig. 5. Schematic view of the problem. Same color means same transitions (for a given character).

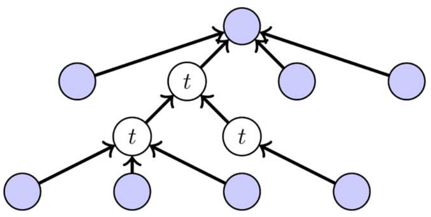

Fig. 6. Backward DFS process. Same color means same transition for character x. States with temporary transitions are denoted by t.

a transition for a given character can be defined as temporary, as shown in Fig. 5. In the picture, nodes are shown divided in sets according to their parent–child relationships (defined by the bold arrows) and their transitions (for a given character). In details, all nodes with the same transition for a given char share the same color: Sets S_1, S_2, and S_4 provide the same transition for the char x, while S_3 defines a different next state. If we set all the transitions for x in S_3 as temporary, we can avoid storing the transitions for x in S_1.

In the construction phase, in order to recognize the nodes where a transition can be defined as temporary, for each char x of each state s in the δFA, a depth-first search (DFS) is performed, starting from s as root. The DFS walks through the δFA backwards (from children to parents) and halts when all the leaves are ancestor states that share the same transition (equal to the one in s for x), as depicted in Fig. 6. The depth of the tree corresponds to the “order” i of the δ^iFA we are building.

For example, in the δ^2FA of Fig. 3(c), if we start the DFS from the transition for c in state 3, the DFS stops when it reaches the ancestor leaves 1, 2, 4, and 5. Indeed, for all the ancestors, char c leads a transition towards state 1.

Now, all the transitions for x in the internal nodes (nonleaves) of the DFS tree can be set as temporary (as happens for the white states in Fig. 6). In the example in Fig. 3, only the state 3 is not a leaf, and its transition for c is set as temporary.

Since ancestor leaves provide the base for the transition set of their subsequent states, in the construction we have to make sure that, in the DFS tree for character x, no ancestor leaf has a temporary transition for x. Hence, this process introduces some constraints and, as usual when dealing with constraints on graphs, this creates new problems. As described above, when setting a subset z of transitions as temporary, we must rely on some other transitions (the leaves y of the DFS tree rooted in z) to be nontemporary. This can be classified as a graph-coloring problem, which is known to be NP-hard.

Algorithm 3: Pseudocode for the creation of the transition table t_c of a δ^iFA from the transition table t of a δFA.

```
1:  $t_c \leftarrow t$
2:  **for all** state $s$ in $\delta$FA **do**
3:      **for all** char $c$ **do**
4:          **if** $\exists\ t[s,c]$ **then**
5:              $\mathrm{res} = \mathrm{DFS}(\mathrm{root} = s, \max\_\mathrm{depth} = i, s_{\mathrm{int}}, s_{\mathrm{leaf}})$
6:              **if** $(\mathrm{res} == \mathrm{SUCCESSFUL})$ **then**
7:                  **for all** state $j \in s_{\mathrm{int}}$ **do**
8:                      set $t_c[j,c]$ temporary
9:                      $s_{\mathrm{child}} \leftarrow$ all children of $j$
10:                     **for all** state $k \in s_{\mathrm{child}}$ **do**
11:                         **if** $t[k,c] == t[s,c]$ **then**
12:                             delete $t_c[k,c]$
```

Therefore, we adopt a straight construction: We build the δ^NFA in a single run by observing all the transitions, by creating the DFS tree when possible, and by blocking the DFS when a constraint is found. This solution is very fast because it does not explore the whole solution domain; it simply gives up the idea of optimality. While this may appear unusual and is certainly nonoptimal, it is however motivated by a number of experimental results (reported in the following section) where this approach does not differ significantly from the optimal setting (if ever reachable) in terms of transitions reduction.

Algorithm 3 describes the construction of a δ^iFA from a δFA. The first step is the copy of the whole transition table t of the δFA into the one t_c of the δ^iFA. Then, for each transition in t, we trigger a DFS with a maximum depth of i. If the DFS is successful (i.e., all the leaves are ancestor states that share the same transition), we set the transitions of the internal nodes (s_{int}) as temporary, while we delete the transitions of the children s_{child} of the temporary states if they are the same as the root.

The complexity of δ^iFA construction (to be added to that of δFA, which is of the order of $O(N \times C^2)$, as shown in Section III-C) is bounded by $O(N \times C \times \bar{d}^i)$, where N is the number of DFA states, C is the number of chars, $\bar{d}$ represents the average connectivity degree of a state, and i is the order of the δ^iFA (hence, $\bar{d}^i$ is an upper bound on the number of nodes required by each DFS tree). Notice that this construction is not much taxing since the order i does not increase significantly, as confirmed by our experimental results on real data sets.

V. Application to H-cFA and XFA

One of the main advantages of δFA and δ^NFA, shared also by other algorithms, is that they are orthogonal to many other schemes. Indeed, very recently, two major DFA compressed techniques have been proposed, namely H-cFA [10] and XFA [9]. Both these schemes address, in a very similar way, the issue of state blow-up in DFA for multiple regular expressions, thus candidating to be adopted in platforms that provide a limited amount of memory as network processors, FPGAs, or ASICs. The idea behind XFAs and H-cFA is to trace the traversal of some certain states that correspond to closures by means of a small scratch-memory. Normally, those states would lead to state blow-up; in XFAs and H-cFA, flags and counters are used to significantly reduce the number of states.

The application of δFA and δ^NFA to H-cFA and XFA (which is tested in Section IX) is obtained by storing the "instructions" specified in the edge labels only once per state. Moreover, in the construction of δFA, edges are considered "different" also when their specified "instructions" are different.

To better clarify the idea, an example of the application to H-cFA (again taken from a previous paper [10]) is reported in Fig. 7(a). The aim is to recognize the regular expressions $\cdot^{*}ab[^{\wedge}a]^4c$ (where 4 is a length restriction) and $\cdot^{*}\mathrm{def}$; labels also include conditions and operations that operate on a flag (set/reset with ± 1) and a counter n (for more details, refer to [10]). A standard DFA would need 20 states and a total of 120 transitions, while the corresponding H-cFA [Fig. 7(a)] uses 6 states and 38 transitions.

By applying our solutions, with the previous new definition of "different" states, we further reduce the number of transitions: the δFA representation of the H-cFA [Fig. 7(b)] requires 20 transitions, and the δ^NFA [Fig. 7(c)] 14 only.

VI. Compressing Char-State Pairs

In a δFA or in a δ^NFA, the size of each state is not fixed because an arbitrary number of transitions can be present, and therefore state pointers are required, which generally are standard memory addresses. They constitute a large part of the memory occupation associated with the DFA data structure, so we propose here a compression technique that remarkably reduces the number of bits required for each pointer. Such an algorithm is fully compatible with δ,FA, δ^NFA, and most of the other solutions for DFA compression already shown in Section II.

The proposed algorithm (hereafter referred to as *Char-State compression* or simply *C-S*) is based on the observation of several standard rule sets: In most cases, the edges reaching a given state are labeled with the same character. Table I shows, for different available data sets (see Section IX for more details on sets), the percentage of nodes that are reached only by transitions corresponding to a single character over the total number of nodes.

As a consequence, a consistent number of states in the DFA can be associated with a single char and referred to by using a "relative" address (hereafter simply *rel-id*). Since the number of such states will be smaller than the number of total states, a *rel-id* will require a lower number of bits than an absolute address. In addition, as the next state is selected on the basis of the next input char, only its *rel-id* has to be included in the state transition set, thus requiring less memory space.

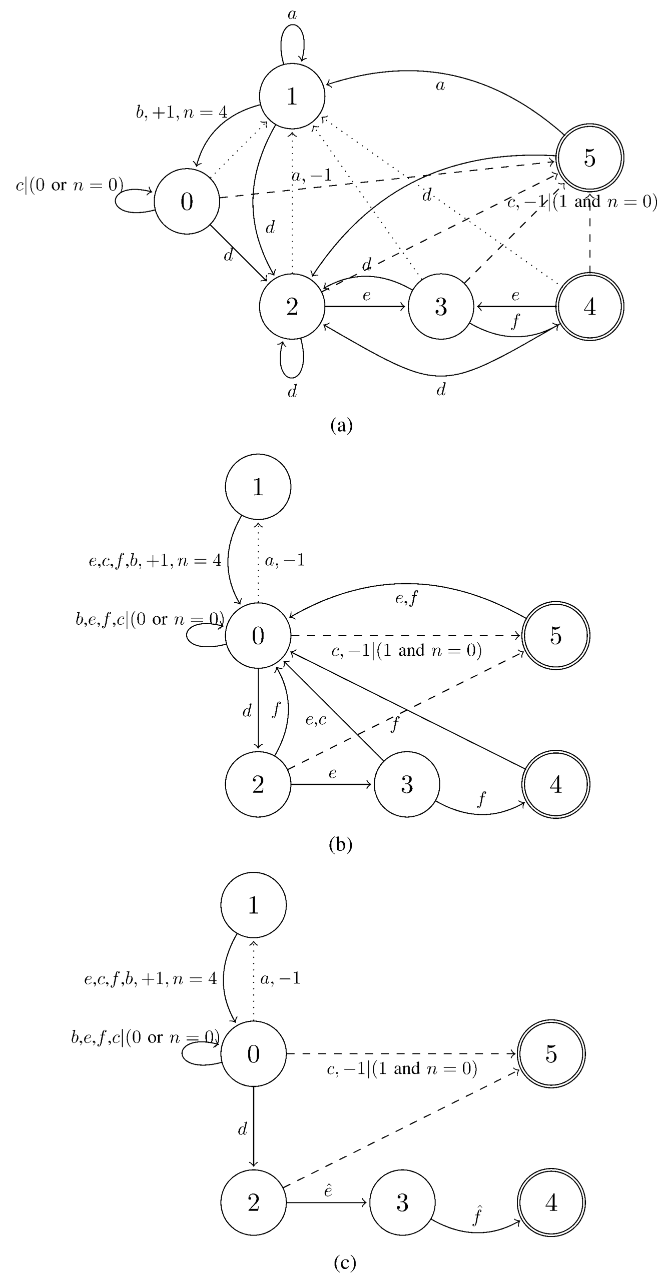

Fig. 7. Automata recognizing $\cdot^{*}ab[^{\wedge}a]^4c$ and $\cdot^{*}\mathrm{def}$. (a) H-cFA. Dashed and dotted edges have same labels, respectively $c, -1\,|\,(1 \text{ and } n = 0)$ and $a, -1$. Not all edges are shown to keep the figure readable. The real number of transitions is 38. (b) δFA applied to H-cFA. Here, all the 20 transitions are shown. (c) δ^NFA applied to H-cFA. All the transitions are shown: Six transitions have been deleted, and two have become temporary.

TABLE I

% of States Reached by Edges With the Same One Label ($p_{1\mathrm{char}}$), *C-S* Compression (r_{comp}), Average Number of Scratchpad Accesses Per Lookup (η_{acc}), and Indirection-Table Size (T_{S})

Data set	p_{1char} (%)	r_{comp} (%)	η_{acc}	T_S (KB)
Snort34	96	59	1.52	27
Cisco30	89	67	1.62	7
Cisco50	83	61	1.52	13
Cisco100	78	59	1.58	36
Bro217	96	80	1.13	11

As an example, in Fig. 8, character a produces transitions to states 1 and 2 only. Therefore, 1 bit of *rel-id* suffices to encode

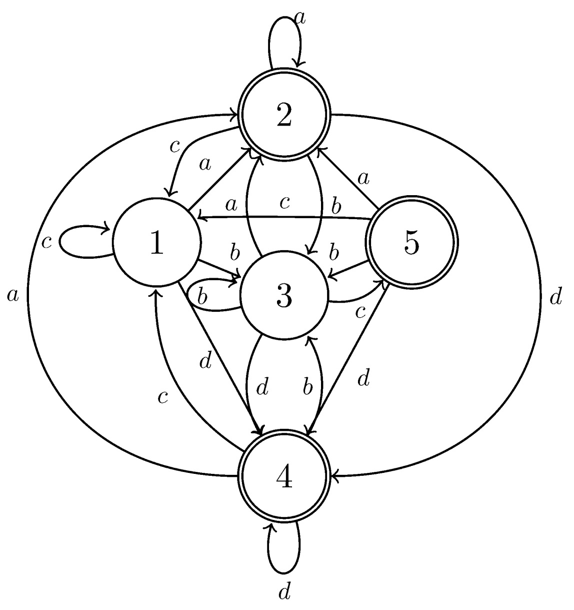

Fig. 8. Automata recognizing (a^+), (b^+c), and (c^*d^+).

its two possible next-states. Associating state 1 with *rel-id* 0 and state 2 with *rel-id*, we only need to store two elements in the indirection table for the character a. During lookup, the character a is adopted as column-index of the indirection table, and the *rel-id* as index of the row.

The absolute address of the next state will be retrieved by using a small indirection table, which, as far as our experimental results show, will be small enough to be kept in local memory, thus allowing for fast lookup. It is clear that such a table will suffer from a certain degree of redundancy. Some states will be associated with several *rel-ids*, and their absolute address will be reported more than once. In the next section, we then propose a method to cope with such a redundancy, in the case it leads to an excessive memory occupation.

Fig. 9 shows the distribution of the number of bits that may be used for a *rel-id* when applying our compression scheme to standard rule sets. As it can be noticed, next state pointers are represented in most cases with very few bits (less than 5). Even in the worst case, the number of bits is always below 10. In the second column of Table I, we show the compression rate achieved by *C-S* with respect to a naive implementation of DFA for the available data sets: The average compression is between 60% and 80%.

As for the indirection table required by *C-S*, if several states with multiple *rel-ids* are present, this might be an issue. Our simple solution is to use an "adapting" double indirection scheme: When a state has a unique *rel-id*, its absolute address is written in the *C-S* table; otherwise, if it has multiple *rel-ids*, for each one of them the table reports a pointer to a list of absolute addresses. This scheme is somewhat self-adapting since, if few states have multiple identifiers, most lookups will require a single local-memory access, while, if many states require multiple ids, the translation will likely require two accesses, but the table size will be consistently reduced. The fifth column of Table I shows the size of this indirection table when applying

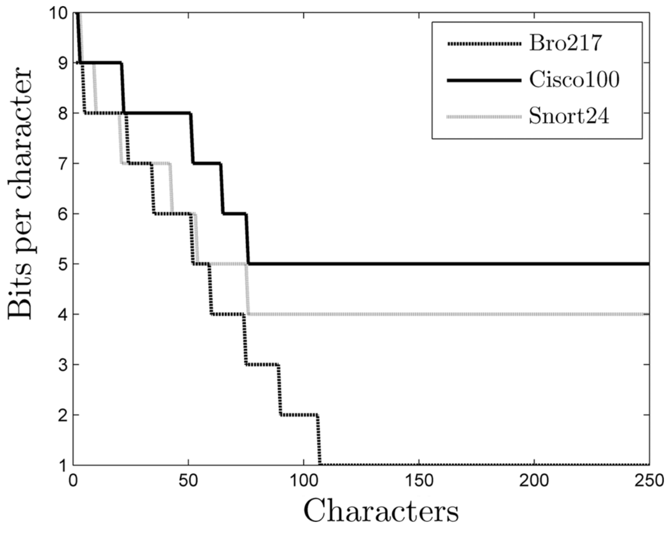

Fig. 9. Distribution of the number of bits used for a *rel-id* with our compression scheme for standard rule sets.

δFA to the different data sets and confirms its placement in local table.

The memory accesses and size requirements of this scheme experimented on a number of data sets are reported in Table I.

VII. C-S in δFA and δ^NFA

The *C-S* can be easily integrated within the δFA (or δ^NFA) scheme, and both algorithms can be cross-optimized. Indeed, *C-S* helps δFA by reducing the state size, thus allowing the read of a whole transition set in a single memory access on average. On the other hand, *C-S* can take advantage of the same heuristic of δFA: Successive states (or child $-N$-step ancestors) often present the same set of transitions. As a consequence, it is possible to parallelize the retrieval of the data structure corresponding to the next state and the translation of the relative address of the corresponding next-state in a sort of "speculative" approach.

More precisely, let s and $s+1$ be two consecutive states, and let us define A_s^c as the relative address of the next hop of the transition departing from state s and associated with the character c. According to the previously mentioned heuristic, it is likely that $A_s^c = A_{s+1}^c$. Since, according to our experimental data (see Section IX), 90% of the transitions do not change between two consecutive states, we can consider such an assumption to be verified with a probability of roughly 0.9. As a consequence, when character c is processed, it is possible to parallelize two memory accesses:

- retrieve the data structure corresponding to state $s+1$;
- retrieve the absolute address corresponding to A_{s+1}^c in the local indirection table.

In order to roughly evaluate the efficiency of our implementation in terms of the state lookup time, we refer to a common underlying hardware architecture (described in Section VIII). It is quite common [18] that the access to a local memory block is more than twice as fast as that of an off-chip memory bank. As a consequence, even if a double indirection is required, the address translation will be ready when the data associated with the next state will be available. If, as it is likely, $A_s^c = A_{s+1}^c$, it will be possible to directly access the next state (say $s + 2$) through the absolute pointer that has just been retrieved. Otherwise, a further lookup to the local indirection table will be necessary.

Such a parallelization can remarkably reduce the mean time needed to examine a new character. As an approximate estimation of the performance improvement, let us suppose that our assumption (i.e., $A_s^c = A_{s+1}^c$) is verified with probability $p = 0.9$, that one access to on-chip memory takes $t_{\text{on}} = 4T$ and to an external memory $t_{\text{off}} = 10T$ [18], and that an address translation requires $n_{\text{trans}} = 1.5$ memory accesses (which is reasonable according to the fourth column of Table I). The mean delay will then be

$$\overline{t_{\text{par}}} = (1 - p)(t_{\text{off}} + n_{\text{trans}} \times t_{\text{on}}) + p \times t_{\text{off}} = 10.6T.$$

This means that, even with respect to the implementation of δFA, the *C-S* scheme increases the lookup time by a limited 6%. On the contrary, the execution of the two tasks serially would require

$$\overline{t_{\text{ser}}} = (t_{\text{off}} + n_{\text{trans}} \times t_{\text{on}}) = 16T.$$

The parallelization of tasks results then in a rough 50% speedup gain.

VIII. Implementation of Our Solutions

In order to outline some guidelines for an efficient implementation of our solutions on an hardware architecture, we make some general assumptions on the system architecture, satisfied by many network processing devices (e.g., the Intel IXP network processors [19]).

In particular, we assume our system to be composed of:

- a standard 32-bit processor provided with a fairly small local memory (a few kilobytes); the access time to such a memory is of the same order of the execution time of an assembly level instruction (less than 10 clock cycles);
- an on-chip fast access memory block (the "scratchpad") with higher storage capacity (in the order of 100 kB) and with an access time of a few dozens of clock cycles;
- an off-chip large memory bank with a storage capacity of dozens of megabytes and with an access time in the order of hundreds of clock cycles.

We consider both δFA and *C-S* algorithms (the considerations about δ^NFA are equivalent). As for the former, two structures are needed: a unique local transition set and a set of data structures representing each state (kept in the external memory). The local transition set is an array of 256 pointers (one per char) that refer to the external memory location of the data structure associated with the next state for that input char.

A δFA state is stored as a variable-length structure. In its most general form, it is composed of a 256-bit-long bitmap (specifying which valid transitions are already stored in the local transition set and which ones within the state) and a list of the pointers for the specified transitions. In many cases, because of the effects of the δFA transition reduction, a simple linear encoding with a list of pairs (char, next-state) suffices and requires less memory than a fixed-size bitmap.

TABLE II
Characteristics of the Rule Sets Used for Evaluation

Dataset	# of regex	ASCII range	% regex w/ *,+,?	Original DFA	
				# states	# transitions
Snort24	24	6-70	83.33	13886	3554816
Cisco30	30	4-37	10	1574	402944
Cisco50	50	2-60	10	2828	723968
Cisco100	100	2-60	7	11040	2826240
Bro217	217	5-76	3.08	6533	1672448

TABLE III
Simple versus Optimal Approach: Ratio of Deleted and Temporary Transitions

Dataset	Cisco30	Cisco50	Snort24	Snort31	Bro217
Del. ratio	99%	89%	100%	100%	100%
Temp. ratio	85%	76%	100%	100%	100%

Since in a state data structure a pointer is associated with a unique character, in order to integrate *C-S* in this scheme, it is sufficient to substitute each absolute pointer with a *rel-id*. The only additional structure consists of a character-length correspondence list, where the length of the *rel-ids* associated with each character is stored. Such information is necessary to parse the pointer lists in the node and in the local transition set. However, since the maximum length for the identifiers is generally lower than 16 bits (as it is evident from Fig. 9), 4 bits for each char are sufficient. The memory footprint of the character-length table is well compensated by the corresponding compression of the local transition set, composed of short *rel-ids* (our experimental results show a compression of more than 50%).

Furthermore, if a double indirection scheme for the translation of *rel-ids* is adopted, a table indicating the number of unique identifiers for each char will be necessary, in order to parse the indirection table. This last table (that will be at most as big as the compressed local transition table) can be kept in local memory, thus not affecting the performance of the algorithm.

IX. Experimental Results

In this section, we perform a series of experimental runs by using some data sets of the Snort [2] and Bro [3] intrusion detection systems and Cisco security appliances [4]. In detail, such data sets, presenting up to hundreds of regular expressions, have been randomly reduced in order to obtain a reasonable amount of memory for DFAs and to mimic the size of other sets used in literature [6], [16], [17].

Some statistical characteristics of such data sets are summarized in Table II, where we list, for each data set, the number of rules, the ascii length range (minimum and maximum length of regexes), and the percentage of rules including "wildcards symbols" (i.e., $*$, $+$, ?). Moreover, the table shows the number of states and transitions in the original DFAs.

As a first set of results, in order to motivate the simplistic approach to the construction of δ^NFA, we compare our approach to a fictious "optimal" construction (which is invalid, but serves as a bound for our evaluation). Since the ultimate goal of this work is to come up with an efficient way to further reduce the number of transitions to be stored in a δFA, the comparison is expressed in terms of deleted transitions. Therefore, the results in Table III show the ratio between the number of deleted (and temporary) transitions of our simple approach and the number of

TABLE IV
COMPRESSION OF THE DIFFERENT ALGORITHMS. (a) TRANSITIONS REDUCTION (%). (b) MEMORY COMPRESSION (%)

(a)

Dataset	D^2FA		Bec-Cro	CD^2FA	δFA		δ^NFA
	DB=∞	DB=2			trans	dup. states	
Snort24	98.92	89.59	98.71	82.25	96.33	0	97.01
Cisco30	98.84	79.35	98.79	86.59	90.84	7.12	92.37
Cisco50	98.76	76.26	98.67	87.19	84.11	1.1	87.05
Cisco100	99.11	74.65	98.96	93.08	85.66	11.75	86.90
Bro217	99.41	76.49	99.33	93.01	93.82	11.99	94.32

(b)

Dataset	D^2FA		Bec-Cro	CD^2FA	δFA	δ^NFA
	DB=∞	DB=2				
Snort24	97.56	80.66	97.29	97.63	95.02	96.02
Cisco30	97.99	70.05	97.93	89.71	91.07	92.76
Cisco50	98.35	70.22	98.25	91.27	87.23	89.54
Cisco100	98.94	71.48	98.73	93.82	89.05	90.3
Bro217	98.63	62.33	98.51	97.79	92.79	93.4

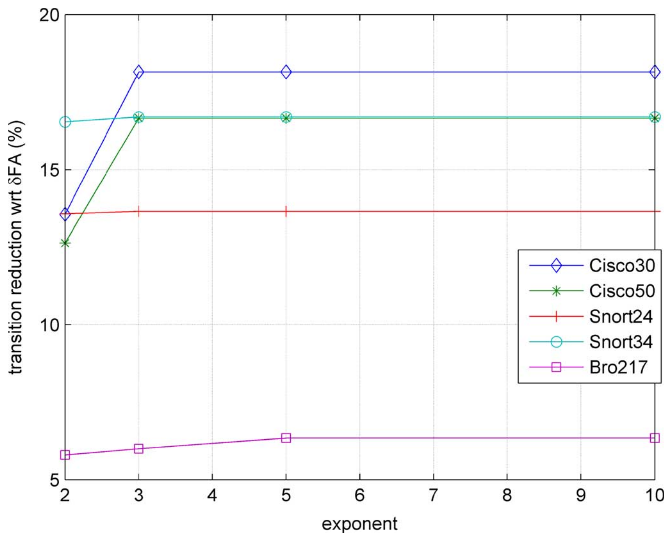

Fig. 10. Transition reduction (with respect to δFA) as a function of the exponent i in δ^iFA.

deleted (and temporary) transitions we may have in the optimal setting. The latter is computed by assuming that the construction of every DFS tree is successful (i.e., simply ignoring the actual constraints discussed above). Even two colliding trees[1] are accepted, which means that we can mistakenly count an edge even more than once. The values in the table suggest that the simple approach is effective and provides very good results, reaching the maximum number of deleted transitions in almost all the cases.

Once our construction process is assessed, we move to the analysis of the effect of the "exponent" i of our δ^iFA. In Fig. 10, we report the percentage of transition reduction produced by δ^iFA (with $i = 1, 2, \ldots, 10$) with respect to the regular δFA. As mentioned, the exponent i represents the maximum depth of the DFS trees and hence the maximum order supported; this is obtained by simply stopping the construction whenever the tree grows more than i levels deep.

While the maximum order of the delta-encoding grows, the average order in all the tests was always steadily between 2 and 2.12. This is consistent with the results in Fig. 10, where it is evident that the advantages of a large exponent are very limited and usually a δ^2FA or δ^3FA provides the best results.

[1]Two trees A and B collide if their definitions of a transition collide, for instance if A requires a transition to be nontemporary and B sets it as temporary.

Table IV shows a performance comparison among δFA and δ^NFA and the previous best known algorithms. More precisely, the table shows the compression in terms of transitions and amount of memory for a standard DFA that recognizes such data sets, as well as the percentage of duplicated states. For δ^NFA, we adopt the exponent $N = 3$ (as discussed previously). For D^2FA and Bec-Cro, we use the code of *regex-tool* [20], which builds a standard DFA and then reduces states and transitions through the different algorithms. In particular, for the D^2FA the code runs with two different values of the bound B, which is a parameter that affects the structure size and the average number of state-traversals per character [6]. As for CD^2FA, since no public software is available as of today, we develop the code according to the specification details provided in [8].

The compression in Table IV(a) is simply expressed as the ratio between the number of deleted transitions and the number of original ones, while in Table IV(b) it is expressed considering the overall memory consumption, therefore taking into account the different state sizes and the additional structures as well. For fair comparison, all algorithms (with the exception of CD^2FA that does not store explicit state pointers, but rather retrieves them as the output of a hash function) have been applied in combination with the *C-S* technique.

Overall, our algorithms achieve a degree of compression comparable to that of D^2FA, Bec-Cro, and CD^2FA. Moreover, δ^NFA provides a valuable improvement with respect to δFA at the expense of a minimal change in the algorithm.

The major advantage with respect to D^2FA and Bec-Cro is represented by the higher lookup speed achieved by preserving one transition per character. This result is obtained by adopting a proper implementation model that effectively hides the latency involved in on-chip memory accesses. While the processor is waiting for the slow memory access to complete, it can update the local transition table with the information it reads from the previous state, so that the additional memory accesses do not in fact involve any additional processing delay.

In addition, remember that since our solutions are orthogonal to most of the previous algorithms (except for CD^2FA), a further reduction is possible by combining them.

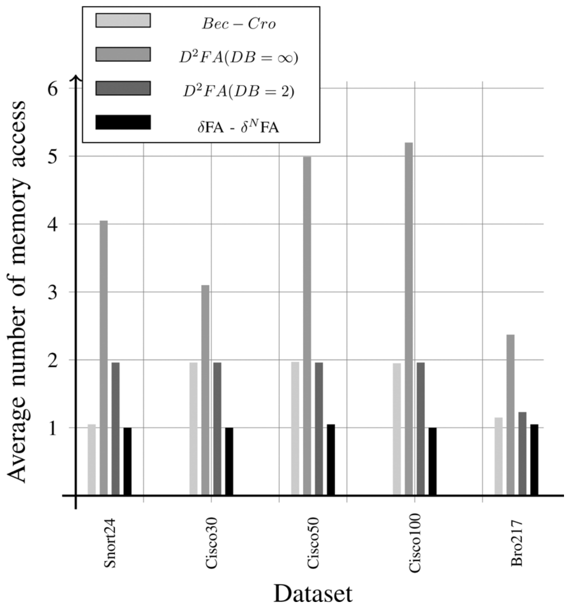

Fig. 11. Mean number of memory accesses.

TABLE V
MEMORY ACCESSES WIDTH (BITS)

Dataset	D^2FA		Bec-Cro	CD^2FA	δFA	δ^NFA
	DB=∞	DB=2				
Snort24	120.76	123.80	164.99	84.8	114.0	95.2
Cisco30	122.36	127.84	134.58	370.1	234.0	198.0
Cisco50	122.04	127.93	135.74	279.7	301.0	246.3
Cisco100	122.26	129.70	133.26	168.1	291.0	267.1
Bro217	122.02	89.81	80.60	75.5	213.0	197.6

Fig. 11 shows the average number of memory accesses required to perform pattern matching through the compared algorithms. It is worth noticing that, while δFA (just as δ^NFA) needs about <1.05 accesses (more than 1 because of the integration with the *C-S* scheme), the other algorithms reported in the figure require more accesses, thus increasing the lookup time. Notice that the figure does not report CD^2FA as, according to [8], even in case of no cache, it always requires a single memory access.

Table V reports the width of memory accesses for all evaluated algorithms in terms of average number of bits involved per state access. In order to do that, we had to make some assumptions on the implementation of the benchmark data structures. In particular, we assumed that, as in our case, a state with a variable number of transitions is encoded either as a list of (char, next-state) pairs (in case a few transitions are specified) or as a bitmap marking which transitions are defined plus a list of next-states (in case most of the transitions are specified). As for CD^2FA, we stuck to the description provided in [8].

From such a comparison, it appears that our algorithms generally need larger memory accesses with respect to the others as they need to fetch the state as a whole in order to keep the local transition table updated. In all cases, though, a state can be usually retrieved with a single memory read operation (as it also emerged from the memory accesses comparison) if a reasonable access width of 256–512 bits is assumed (and, in fact, this is the case for several embedded network processing architectures, such as Intel IXP network processors). Notice, however, that the average state-size value can be somewhat misleading,

TABLE VI
NUMBER OF TRANSITIONS AND MEMORY COMPRESSION BY APPLYING δFA + C-S TO XFA (THUS OBTAINING A δXFA)

Dataset	# of states	# of trans. XFA	# of trans. δXFA	Mem. Compr. (%)
c2663-2	14	3584	318	92
s2442-6	12	3061	345	74.5
s820-10	23	5888	344	94.88
s9620-1	19	4869	366	92.70

as it is heavily influenced by the presence of outliers, i.e., states that have most of their transitions specified, and thus showing a much larger size with respect to the others.

All the above results shows that our approach presents similar performance with respect to the other algorithms in terms of number of accesses and compression, while it generally involves wider memory accesses that, however, do not involve higher latency in most current memory blocks (as discussed). Indeed, our scheme looks more suitable to embedded processor architectures, where low-level programming allows to explicitly use the cache hierarchy. For example, if compared to CD^2FA, both approaches add some additional processing with respect to previous solutions in order to achieve a better performance tradeoff. In the case of CD^2FA, such an overhead consists of computing additional hash functions, while in our case it basically involves accessing and updating the local transition table. In the first case, the additional computation delay cannot be avoided, as it is crucial to the determination of the next state. In our case, on the contrary, the update of the local table can be carried out while the processor is waiting for the next state to be fetched from the slow memory, and the additional processing latency can be completely hidden.

As a final experiment, Table VI reports the results we obtained by applying δFA and *C-S* to one of the most promising approach for regular expression matching: XFAs [9]. The data set (courtesy of Randy Smith) is composed of single regexes with a number of closures that would lead to a state blow-up. The XFA representation limits the number of states (as shown in the table). By adopting δFA and *C-S*, we can also reduce the number of transitions with respect to XFAs, thus achieving a further size reduction. In detail, the reduction is more than 90% (except for a single case) in terms of number of transitions, which corresponds to a rough 90% memory compression (last column in the table). The memory requirements, both for XFAs and δXFAs, are obtained by storing the "instructions" specified in the edge labels only once per state.

X. CONCLUSION

In this paper, we have presented a new compressed representation for deterministic finite automata, called Delta Finite Automata. The algorithm considerably reduces the number of states and transitions, and it is based on the observation that most adjacent states share several common transitions, so it is convenient to store only the differences between them. Moreover, we have presented an improvement to δFA that exploits the Nth-order dependence between states and further reduces the number of transitions by adopting the concept of temporary transition. Both the schemes are orthogonal to most of the previous solutions, thus allowing for higher compression rates.

A new encoding scheme for states has been also proposed (which we refer to as char state), which exploits the association of many states with a few input chars. Such a compression scheme can be efficiently integrated into δFA and δ^NFA, allowing a further memory reduction with a negligible increase in the state lookup time. The experimental runs have shown remarkable results in terms of lookup speed as well as the issue of excessive memory consumption.

References

[1] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proc. ACM CCS*, 2003, pp. 262–271.

[2] "Snort: Lightweight intrusion detection for networks," Sourcefire, Inc., Columbia, MD [Online]. Available: http://www.snort.org/

[3] "Bro: A system for detecting network intruders in real time," Lawrence Berkeley National Laboratory, Berkeley, CA [Online]. Available: http://www.bro-ids.org

[4] J. William and W. Eatherton, "An encoded version of reg-ex database from Cisco Systems provided for research purposes," 2005.

[5] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. DiPietro, "An improved DFA for fast regular expression matching," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 29–40, 2008.

[6] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. ACM SIGCOMM*, 2006, pp. 339–350.

[7] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proc. IEEE INFOCOM*, 2007, pp. 1064–1072.

[8] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. ANCS*, 2006, pp. 81–92.

[9] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 207–218, 2008.

[10] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. ACM ANCS*, 2007, pp. 155–164.

[11] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[12] B. Commentz-Walter, "A string matching algorithm fast on the average," in *Proc. ICALP*, 1979, pp. 118–132.

[13] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Dept. Comput. Sci., Univ. Arizona, Tucson, Tech. Rep. TR-94-17.

[14] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. IEEE INFOCOM*, 2004, pp. 333–340.

[15] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. FCCM*, 2001, pp. 227–238.

[16] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. ACM ANCS*, 2007, pp. 145–154.

[17] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. ACM CoNEXT*, 2007, pp. 1–12.

[18] G. Varghese, *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*. San Mateo, CA: Morgan Kaufmann.

[19] E. J. Johnson and A. R. Kunze*, IXP2400–2800 Programming: The Complete Microengine Coding Guide*. Santa Clara, CA: Intel Press, 2003.

[20] M. Becchi, "Regex tool," 2009 [Online]. Available: http://regex.wustl.edu/

Domenico Ficara (M'10) received the Ph.D. degree in information engineering from the Department of Information Engineering, University of Pisa, Pisa, Italy.

During his Ph.D. studies, he collaborated with Cisco Systems, San Jose, CA, on deep packet inspection research and development projects. His main research interests are deep packet inspection and network topology discovery techniques.

Andrea Di Pietro (S'10) received the Master's degree in telecommunication engineering from the University of Pisa, Pisa, Italy, in April 2007 and is currently pursuing the Ph.D. degree with the NetGroup of the University of Pisa.

From May 2007 to December 2008, he was a Research Assistant with the NetGroup of the University of Pisa. His research interests are in network tomography and network performance measurement.

Stefano Giordano (SM'10) received the Master's degree in electronics engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in 1990 and 1994, respectively.

He is an Associate Professor with the Department of Information Engineering, University of Pisa, where he is responsible for the telecommunication networks laboratories. His research interests are telecommunication networks analysis and design, simulation of communication networks and multimedia communications.

Dr. Giordano is Secretary of the Communication Systems Integration and Modeling (CSIM) Technical Committee. He is Associate Editor of the *International Journal on Communication Systems* and of the *Journal of Communication Software and Systems* technically cosponsored by the IEEE Communication Society. He is a member of the Editorial Board of the *IEEE Communication Surveys and Tutorials*. He is one of the referees of the European Union, the National Science Foundation, and the Italian MIUR and MAP Ministries.

Gregorio Procissi (M'10) received the graduate degree in telecommunication engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in 1997 and 2002, respectively.

From 2000 to 2001, he was a Visiting Scholar with the Computer Science Department, University of California, Los Angeles. In September 2002, he became a Researcher with Consorzio Nazionale Inter-Universitario per le Telecomunicazioni (CNIT) in the Research Unit of Pisa, Italy. Since 2005, he has been an Assistant Professor with the Department of Information Engineering, University of Pisa. His research interests are measurements and performance evaluation of IP networks.

Fabio Vitucci (M'09) received the Master's degree in telecommunication engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in October 2004 and June 2008, respectively.

He currently conducts research with the Department of Information Engineering, University of Pisa, in the areas of packet classification, pattern matching, and network processors.

Gianni Antichi (M'10) received the Master's degree in telecommunication engineering with a thesis on implementation of a high-performance IP traffic generator in September 2007 from the University of Pisa, Pisa, Italy, where he has been a Ph.D. candidate with the Department of Information Engineering since January 2008.

He is currently doing research in the area of packet classification, network processors, and FPGA.