One Automaton to Rule Them All: Beyond Multiple Regular Expressions Execution

Luisa Cicolini, Filippo Carloni, Marco D. Santambrogio, Davide Conficconi,
Dipartimento di Elettronica, Informatica e Bioingegneria, Politecnico di Milano, Milan, Italy
luisa.cicolini@mail.polimi.it; {filippo.carloni, marco.santambrogio, davide.conficconi}@polimi.it

***Abstract*—Regular Expressions (REs) matching is crucial to identify strings exhibiting certain morphological properties in a data stream, resulting paramount in contexts such as deep packet inspection in computer security and genome analysis in bioinformatics. Yet, due to their intrinsic data-dependence characteristics, REs represent a complex computational kernel, and numerous solutions investigate pattern-matching efficiency in different directions. However, most of them lack a comprehensive ruleset optimization approach to truly push the pattern matching performance when considering multiple REs together. Thus, exploiting REs morphological similarities within the same dataset allows memory reduction when storing the patterns and drastically improves the dataset-matching throughput.**

Based on this observation, we propose the Multi-RE Finite State Automata (MFSA) that extends the Finite State Automata (FSA) model to improve REs parallelization by leveraging similarities within a specific application ruleset. We design a multi-level compilation framework to manage REs merging and optimization to produce MFSA(s). Furthermore, we extend iNFAnt algorithm for MFSAs execution with the novel iMFAnt engine. Our evaluation investigates the MFSA size-reduction impact and the execution throughput compared with the one of multiple FSA in both single- and multi-threaded configurations. This approach shows an average 71.95% compression in terms of states, introducing limited compilation time overhead. Besides, best iMFAnt achieves a geomean $5.99\times$ throughput improvement and $4.05\times$ speedup against single and multiple parallel FSAs.

***Index Terms*—pattern matching, multi-level compilation, regular expressions, automata merging, parallel execution**

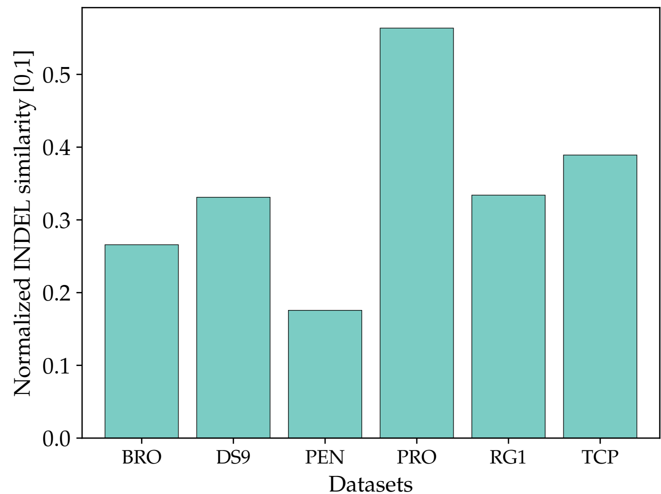


Fig. 1: Average normalized INDEL index for different REs datasets [29], [30] illustrating a proxy of REs similarities. Computed by averaging the normalized INDEL values for every couple of REs within the same dataset.

I. Introduction

Regular Expressions (REs) are a powerful computational kernel describing simple and complex data patterns allowing the identification of characters sub-sequences by matching a set of structural requirements in the analyzed data. Their use is intrinsic in a wide range of practical applications, ranging from computer security [1]–[6] to genome analysis [7], natural language processing [8], and database management [9], [10]. In these applications, REs are essential to identify data segments presenting specific characteristics in a large data stream, e.g., when looking for malicious signature during packet inspection [2], [11]. However, to do so, REs rely on complex operators, e.g., describing (un)bounded repetitions of characters and sub-REs. These characteristics make pattern matching more complex than basic string matching with its simple character concatenation [12]–[14], which is instead a well-defined problem addressed by various existing algorithms [15]. In this context, the parallelization [6], [16]–[23] of pattern matching represents a valuable opportunity, increasing the throughput - in terms of number of patterns simultaneously analyzed - thus yielding significant improvements to time- and performance-critical applications such as packet filtering [2], [3], [19], [24], [25]. A first naive approach to this issue could rely on the parallel execution of multiple REs via multiple threads, i.e., distributing the REs in a ruleset among a set of threads [26]–[28]. Nevertheless, this approach is limited by the number of available cores within the matching engine, which bounds the number of REs that can be executed in parallel. This severely impairs actual applications and dataset applicability, which rely on the analysis of hundreds of REs [26], [29], [30]. A different approach exploits regex decomposition to split complex patterns into disjoint sets of string and FSA components, thus alleviating the computation load by delaying FSA execution until the string matching analysis is required [6].

However, the literature lacks approaches extensively taking advantage of REs similarities within a dataset. Figure 1 quantitatively estimates the average string similarities within two REs using the normalized INDEL metric as a representative proxy, showing an average morphological similarity ratio of 0.34 out of 1. In particular, the normalized INDEL similarity ratio is calculated as $1 - INDEL$, where INDEL represents

the insertion-deletion distance[1] between any two different strings within the same dataset, normalized to the length of the strings. For example, consider strings $s_1 = lewenstein$ and $s_2 = levenshtein$. The insertion-deletion distance between s_1 and s_2 is 3, while the strings length is $10 + 11$. Therefore, $INDEL = \frac{3}{11+10} = 0.1428$ and the corresponding similarity ratio is $1-0.1428 = 0.8572$. Considering the potential benefits of these similarities, we observe that grouping sub-patterns of single Finite State Automata (FSA) shared across multiple FSAs into a single optimized representation would enable more parallel execution of REs with a single exploration.

For these reasons, to fully take advantage of the datasets' morphological characteristics, we propose a novel approach based on merging a common sub-RE among REs within a dataset, such that their matching is performed only once. Specifically, we look for common sub-paths among different FSAs, to be iteratively merged into a single Multi-RE Finite State Automata (MFSA). These sub-paths must exhibit identical morphological characteristics, i.e., they must describe equivalent sub-FSAs recognizing the same language. By introducing the concept of **activation function**, an MFSA can recognize all the languages the concerned FSAs describe, allowing for their distinction. Replacing FSAs with MFSAs in RE-matching reduces the required states and transitions while increasing the ruleset matching throughput.

On top of this, we propose an extensive merging-based optimization technique producing MFSAs, which stands on a formal model extending FSAs one. This procedure is embedded in a compilation framework managing REs conversion and optimization into executable MFSAs. Altogether, our framework comprises both lexical and syntactical analyses of REs, ensuring their correctness, their transformation into equivalent FSA, their subsequent optimization and merging into a single MFSA, and their eventual translation into Automata Network Markup Language (ANML) format, to enable their execution. In addition, we propose an *ad-hoc* extended version of iNFAnt pattern matching algorithm [32], called iMFAnt, able to correctly handle and execute MFSAs in this format.

Experimental results show that the merging approach significantly reduces the states and transitions by 71.95% and 38.88% on average, compared to the equivalent FSAs set, thanks to the merging procedure (§VI-A). Notably, this reduction is effective without resorting to alphabet reduction techniques [33]. Moreover, combining the MFSAs with the proposed iMFAnt algorithm improves at most the single-FSA (single-thread) throughput with a geometric mean of $5.99\times$. Finally, we analyze iMFAnt thread number scalability and compare against the parallel multi-threaded FSAs approach showcasing a best geomean $4.05\times$ speedup. On top of this, MFSAs and iMFAnt demonstrate to achieve better execution times while leveraging fewer threads than the parallel multi-threaded FSAs approach (§VI-C).

The contribution of this work can be summarized as:

- A **merging-based optimization procedure**, taking advantage of REs similarities to build a MFSA out of an initial set of standard FSAs (§III-A).
- A **formal model** of MFSAs, extending FSAs to take into account datasets' morphological characteristics (§III-B).
- An extensive **multi-level compilation framework** comprising the analysis of input REs, their conversion into FSAs, their optimizations and merging into MFSA, and their translation into ANML representation (§IV).
- An **extension** of a current state-of-the-art **pattern matching algorithm** to enable the support of MFSAs from an execution perspective (§V).

II. Background Knowledge

We summarize the formal FSA characteristics, including the differences between Deterministic FSA (DFA) and Non-Deterministic FSA (NFA).

a) Fundamentals of FSAs: REs optimization and minimization algorithms often exploit the FSA equivalent representation which is easier to manipulate [34]. A FSA, specifically a NFA[2], is described by a tuple:

$$a = (Q, \Sigma, \delta, q_0, F)$$

Where Q is the set of states, Σ is the set of allowed input symbols, δ is a state transition function, q_0 is the initial state and $F \subseteq Q$ is the set of final states [35]. A state transition function $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ defines the automaton behavior, where $\mathcal{P}(Q)$ represents the power set of the states set, i.e., the set of all subsets of Q. A move from a state q_1 to a state q_2 upon reading a symbol c is represented as follows:

$$(q_1, cw) \vdash_{FSA} (q_2, w)$$

where w represents the remainder of the input string after consuming (i.e., reading) character c [36]. A standard FSA a recognizes all the strings belonging to the specific language $\mathcal{L}(a)$. Moreover, FSAs enjoy graph properties, such as the *isomorphism* between FSAs $\mathcal{U} : a_1 \rightarrow a_2$ that defines a biunivocal mapping between equivalent FSAs [37]. That is, a_1 and a_2 induce the same family of mappings on the same alphabet. Taking advantage of isomorphisms among different FSAs is at the core of our merging-based optimization.

b) DFA versus NFA: The intrinsic differences between DFAs and NFAs represent a core challenge for automata-based pattern matching. On the one hand, the traversal of DFAs shows an upper complexity limit strictly related to the time required for a single transition traversal. At the same time, DFAs introduce exponential state-explosion issues [38]. To overcome this limitation, DFAs compression algorithms rely on *default transitions*, i.e., transitions reaching a certain state with a certain character starting from different starting states. The representation of default transitions is negligible in the compressed FSA since they are stored once and assumed to

[1] equivalent to Levenshtein distance with insertions and deletions only [31].

[2] NFA and DFA recognize the same language [34].

apply by default for any other state in the FSA unless otherwise specified [33], [34], [39], [40]. On the other hand, NFAs have a significantly lower memory footprint compared to DFAs. However, their execution requires the simultaneous activation of multiple transitions, leading to severe bandwidth limitations. A set of works addressing this issue targets NFAs execution on hardware accelerators [13], [21], [41], [42], exploiting *ad-hoc* algorithms [27], [28], [43]. These algorithmic solution take advantage of NFAs multiple active states by parallelizing the automaton traversal [32]. However, they require non-negligible pre-processing methods.

III. MFSA: A Merging Approach to REs Optimization

The core idea of our approach consists of searching common sub-paths (i.e., sub-patterns) among two or more REs describing equivalent sub-languages over the same alphabet. Upon finding sub-paths common to a sub-set of FSAs $a_1, ..., a_M$, we propose an iterative merging procedure producing a final MFSA preserving the initial FSAs morphology and the languages they describe (§III-A). This approach is equipped with a formal model supporting MFSAs correctness (§III-B).

A. Merging FSAs: Search for Common Sub-Patterns

We design an algorithm aimed at identifying common sub-patterns in a set of input REs. We adopt the FSA-based representation to look for sub-sets of *isomorphic* transitions describing identical sub-languages, i.e., transitions describing paths with the same *morphology* and labels.

Algorithm 1 Merging a set of FSAs into a single MFSA

```
1:  function MERGE_MULTI(ToBeMerged A[])
2:    z ← new MFSA
3:    generateNew(z, A[1])                  ▷ add first automaton
4:    for a : a ∈ A[2 : end] do
5:      ms ← new MS[ ]
6:      for i : i ∈ N_{TS,a} do
7:        for j : j ∈ N_{TS,z} do
8:          if idx_a[i] == idx_z[j] then
9:            r ← trans(i)
10:           t ← trans(j)
11:           while r.idx == t.idx and
12:                 isPath(r − 1, r) and isPath(t − 1, t)) do
13:             ms.push(r, t)
14:             r ← next(r)
15:             t ← next(t)
16:           end while
17:         end if
18:       end for
19:     end for
20:     relabel(ms, a)
21:     generateNew(mrg, a)
22:   end for
23: end function
```

We merge a set A of FSAs $a_1, ...a_M$, in a cascaded fashion as in Algorithm 1, generating a single MFSA $z_{1\leftarrow M}$. For each M-sized group of REs, the first automaton a_1 is copied *as-is* into the MFSA z (line 3, in Figure 2: z_1 is a copy of a_1). Starting from the second FSA a_2, the algorithm compares each incoming FSA a with the evolving MFSA z (line 4). To this end, we represent automata via their adjacency matrix in Coordinate Format (COO): for each transition $q_1 \xrightarrow{c} q_2$ we store the starting state q_1 in vector `row`, the arriving state q_2 in vector `col`, and the single character c or Character Class (CC) C enabling the transition in vector `idx`, where CC is a set of allowed characters. Additionally, MFSAs include a vector preserving the derivation of each transition from the corresponding initial FSAs. To reference this FSA derivation, we say that a transition *belongs to* a FSA. Thus, the `bel` vector of Figure 2 contains the identifiers[3] of the FSAs each transition belongs to. A transition *belongs to* multiple FSAs when it is common to all of them. Figure 2 exemplifies this representation, with an initial MFSA z_1, an incoming FSA a_2, and the result of their merging $z_{1\leftarrow 2}$. To search for a common sub-path, the algorithm iterates over the `idx` vector of this representation (line 6 and 7). Upon finding two transitions enabled by the same single character or CC, (in Figure 2: $0 \xrightarrow{a} 1$ in z_1 and $2 \xrightarrow{a} 3$ in a_2) the algorithm explores the subsequent transitions as long as they describe identical paths (in Figure 2: $0 \xrightarrow{a} 1 \xrightarrow{[gj]} 2$ in z_1 and $2 \xrightarrow{a} 3 \xrightarrow{[gj]} 4$ in a_2). To perform this check, an additional loop (lines 11-16) checks the equivalence of sub-paths, transition-by-transition and stops at the first difference. Cocerning the comparison of standard transitions, given a MFSA z_1 and a FSA a_2, the algorithm looks for two couples of states $(q_{i,1}, q_{j,1}) \in z_1$ and $(q_{n,2}, q_{m,2}) \in a_2$ such that there exists a character $c \in \Sigma_1 \wedge c \in \Sigma_2$ for which $\delta_1(q_{i,1}, c) = q_{j,1} \wedge \delta_2(q_{n,2}, c) = q_{m,2}$.[4]

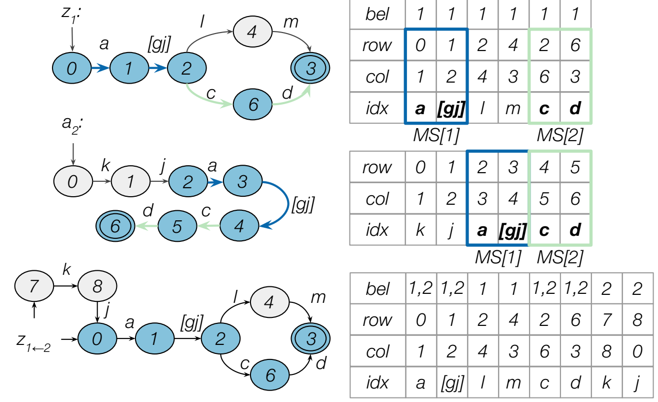


Fig. 2: Given a MFSA z_1 and a FSA a_2, the merging algorithm stores each identical sub-path in its Merging Structure (MS) (e.g., $0 \xrightarrow{a} 1 \xrightarrow{[gj]} 2$ for z_1 and $2 \xrightarrow{a} 3 \xrightarrow{[gj]} 4$ for a_2 in $MS[1]$). The combination of the MSs and their merging yields $z_{1\leftarrow 2}$.

[3]The identifier of a FSA is unique and corresponds to the index of FSA a in set $A[]$, e.g., FSA a_j has identifier $j \in [1, M]$.

[4]In all the equations the second part of the pedex indicates the *belonging* of the state, e.g., state $q_{i,k}$ is the *i-th* state of FSA k

Overall, this search yields a set of 4-tuples:

$$
\begin{aligned}
X = \{&(q_{i,1}, q_{j,1}, q_{n,2}, q_{m,2}) | (q_{i,1}, q_{j,1}) \in z_1 \\
&\wedge (q_{n,2}, q_{m,2}) \in a_2 \wedge \exists\, c \in \Sigma_1, c \in \Sigma_2 \ s.t. \\
&(\delta_1(q_{i,1}, c) = q_{j,1} \wedge \delta_2(q_{n,2}, c) = q_{m,1}\}
\end{aligned} \tag{1}
$$

where Σ_1 and Σ_2 are the alphabets of z_1 and a_2, respectively. Transitions enabled by CCs require a thorough comparison to check all the characters in the CC. In this case, the algorithm analyzes every transition $q_{i,1} \rightarrow q_{j,1}$ described by a character class $CC_{h,1} \in z_1$ and compares it with the transitions in a_2 described by a character class $CC_{l,2}$. The purpose of this phase is to check whether any two classes $CC_{h,1}$ and $CC_{l,2}$ comprehend the same characters, i.e. $CC_{h,1} \equiv CC_{l,2}$. The output of this step is a set of 4-tuples:

$$
\begin{aligned}
Y = \{&(q_{i,1}, q_{j,1}, q_{n,2}, q_{m,2}) | (q_{i,1}, q_{j,1}) \in z_1 \\
&\wedge (q_{n,2}, q_{m,2}) \in a_2 \wedge \exists\, CC_{h,1}, CC_{l,2} \\
&s.t.\ (\delta_1(q_{i,1}, CC_{h,1}) = q_{j,1} \\
&\wedge \delta_2(q_{n,2}, CC_{l,2}) = q_{m,1} \wedge CC_{l,2} \equiv CC_{h,1})\}
\end{aligned}
$$

The tuples in X and Y describe equivalent sub-paths, in fact:

$$
\begin{aligned}
&\forall (q_{i,1}, q_{j,1}, q_{n,2}, q_{m,2}), (q_{j,1}, q_{p,1}, q_{m,2}, q_{s,2}) \in X \cup Y \\
&\exists\, k, w, v \ s.t.\ \delta_1(q_{i,1}, kwv) = q_{j,1} \\
&\wedge \delta_1(q_{j,1}, wv) = q_{p,1} \wedge \delta_2(q_{n,2}, kwv) = q_{m,2} \\
&\wedge \delta_2(q_{m,2}, wv) = q_{s,2}
\end{aligned} \tag{2}
$$

Overall, combining the mergeable tuples returned by X and Y yields the sets of transitions that describe isomorphic paths between the initial FSAs. In both cases, if multiple equivalent sub-paths exist, each of them is stored in an *ad-hoc* structure, called Merging Structure (MS). A MS stores exactly the 4-tuples in sets X and Y to enable the subsequent relabeling phase. In fact, combining the so-obtained MSs yields the subset of transitions and states to merge. Based on this subset, the algorithm exploits a *relabeling function* (line 19) to relabel the incoming FSA a's state labels without changing its morphology. Specifically, states involved in a MS are relabeled to be identical to MFSA z state labels in the corresponding MS (in Figure 2: relabel state 2 in a_2 with 0), while the remaining ones are relabeled not to overlap current MFSA states (in Figure 2: relabel state 0 in a_2 with 7).

Subsequently, Algorithm 1 updates the MFSA z with the transitions and states of the incoming FSA a (line 20). It updates the *belonging* of merged transitions with the incoming FSA identifier (e.g., in Figure 2: update the belonging of transition $0 \xrightarrow{a} 1$ in z_1 to contain $id = 2$) and copies the non-merged transitions to z (e.g., in Figure 2: copy $7 \xrightarrow{k} 8$ to z_1). Preserving the *beloging* of each transition to the corresponding FSAs is fundamental to distinguish one language from another.

To sum up, the search for common sub-paths has three possible outcomes:

a) There are no common sub-REs, i.e., no states to merge: the incoming FSA a is entirely copied into the MFSA, ensuring that all its relabeled states are disjoint from those of the MFSA to avoid states overlapping in the MFSA.

b) There are some common sub-REs: the *belonging* field of common transitions is updated in the MFSA with the incoming FSA identifier, while non-merged states are relabeled not to overlap with existing state labels.

c) The incoming FSA and the MFSA are identical: for all transitions, the algorithm updates the *belonging* field in the MFSA with the incoming FSA identifier.

This procedure merges a set of M FSAs into a single MFSA, and we can repeat it to merge a set of FSAs in M-sized groups, generating a set of MFSAs.

Overall, this procedure constructs *correct* MFSAs, since the morphology of initial FSAs is respected, and no transition is removed nor changed. We approximate the merging Algorithm 1 time complexity as:

$$
\mathcal{O}(4M \cdot N_{TS}^2 + 8N_{TS}^3)(M-1)) \tag{3}
$$

where N_{TS} is the number of transitions in the single FSA and M is the number of merged FSAs (i.e., the merging factor). In Equation (3), the quadratic term is due to the comparisons the algorithm performs between any couple of transitions in the two automata[5]. Instead, the cubic term derives from the multiple iterations over the set of MSs (which includes $N_{TS}-1$ transitions at most), necessary to relabel each state. For most state-of-the-art datasets, the leading terms M and N_{TS} of a single FSA have comparable size ($N_{TS} \sim M$ for actual use cases), as Table I will summarize. With this assumption, the average complexity becomes $\mathcal{O}(M^4)$.

The proposed merging approach suits any dataset, with no requirements on the form of the involved REs and no limitations to the possibly mergeable REs.

B. The Formal Model of the MFSA

The new features of the MFSA model require to extend the formal FSA one to enable the simultaneous description and recognition of multiple languages belonging to the set of merged FSAs. Indeed, the proposed merging procedure differs from classical operations, such as intersection and union, because it does not generate a new language class. Specifically, an MFSA has to *recognize* and *distinguish* all the languages described by the merged FSAs that compose it.

For this purpose, we introduce the new concept of **activation function**, which allows the MFSA to keep track of the active matching FSAs. An activation function J encodes the set of all the valid (i.e., active) FSAs during the traversal of a transition for every state in the recognition path. The activation of a FSA j on a certain transition depends on a set of rules enforcing the correctness of the traversed paths. This feature is essential to MFSAs to prevent incorrect matches. For example MFSA $z_{1 \leftarrow 2}$ in Figure 2 is the result of merging a_1 recognizing $\mathcal{L}(a_1) = a[gj](lm|cd)$ and a_2 recognizing $\mathcal{L}(a_2) = kja[gj]cd$. However, with no further precaution, $z_{1 \leftarrow 2}$ could recognize strings belonging to neither of the languages a_1 and a_2 describe, e.g., $s = kjaglm$, allowing new unwanted recognized languages. To ensure matched strings correctness

[5] $N_{TS,m}$ is at most $M \cdot N_{TS,a}$.

in these cases, we introduce the activation function, that tracks the origin of each transition, i.e., the FSAs a transition belongs to, according to the MFSA construction, and compares it with the active FSAs at traversal time, according to a set of rules. Upon moving from state q_1 to state q_2, after reading character c, the set of active FSAs identifiers changes depending on the number of FSAs the traversed transition belongs to:

- if state q_1 is initial for some FSA j in the set of merged FSAs, identifier j is pushed to the set of active FSAs (Equation (4)),
- if state q_2 is final for some FSA j, as long as j is already active on the departure state q_1, such that $j \in J(q_1)$, label j is popped from the set of active FSAs identifiers, and a match occurs for that RE (Equation (5)),
- if an FSA that is active at starting state q_1 (i.e., $j \in J(q_1)$) comprises no transition $q_1 \xrightarrow{c} q_2$, label j is popped from the set of active FSAs, meaning that the current transition does not belong to FSA j (Equation (6)).

At the beginning of the MFSA recognition, the set J is empty. A transition $q_1 \xrightarrow{c} q_2$ is valid if the active FSAs set returned from arriving state $J(q_2)$ shares at least one common label j with the starting one $J(q_1)$ ($J(q_1) \cap J(q_2) \neq \emptyset$). If this happens, the path (i.e., set of subsequent transitions) analyzed so far is *consistent* with at least one of the paths in the set of merged FSAs, i.e., it belongs to at least one FSA. In mathematical terms, for a transition $q_1 \xrightarrow{c} q_2$, the set of active function on the destination state q_2 changes as follows:

$$q_1 = q_{0,j} \;\; for \;\; j \in \mathcal{R} \Rightarrow J(q_2) = J(q_1) \cup \{j\} \tag{4}$$

$$q_2 \in F_j \;\; for \;\; j \in \mathcal{R} \wedge j \in J(q_1) \Rightarrow J(q_2) = J(q_1) \setminus \{j\} \tag{5}$$

$$\forall j \in J(q_1), \forall c \in \Sigma : \delta_j(q_1, cw) = \emptyset \Rightarrow J(q_2) = J(q_1) \setminus \{j\} \tag{6}$$

Where $\mathcal{R}$ is the set of merged FSAs identifiers, F_j is the set of final states for FSA j, i_j is the initial state of FSA j (unique by definition), δ_j is the standard transition function of FSAs j. The extension of the standard transition function δ follows from these equations to support the update of the *activation function* throughout the traversed path, formalized as:

$$\Delta : (Q \times \Sigma \times \mathcal{P}(\mathcal{R})) \rightarrow (\mathcal{P}(Q) \times \mathcal{P}(\mathcal{R})) \tag{7}$$

For every transition $q_1 \xrightarrow{c} q_2$, function Δ maps a starting state $q_1 \in Q$, a character $c \in \Sigma$ and a set of active FSAs identifiers on starting state q_1 (i.e., $\mathcal{P}(\mathcal{R}) = J(q_1)$) to a set of arrival states $\mathcal{P}(Q)$ with an updated set of active FSAs $\mathcal{P}(\mathcal{R}) = J(q_2)$. The extension of moves formalization follows from these concepts. For every move $q_1 \xrightarrow{c} q_2$, MFSA requires the update of the set of active FSAs depending on the character read and the active FSAs on the departure state, such that:

$$(q_1, aw, J(q_1)) \vdash_{MFSA} (q_2, w, J(q_2)) \tag{8}$$

Eventually, a sequence of moves from an initial to a final state yields a match for a certain FSA j if it traverses a set of *consistent* transitions, i.e., if at least one FSA j is always active during their traversal:

$$\exists\, j \in \mathcal{R} : j \in J(q) \; \forall q \in matching_path \tag{9}$$

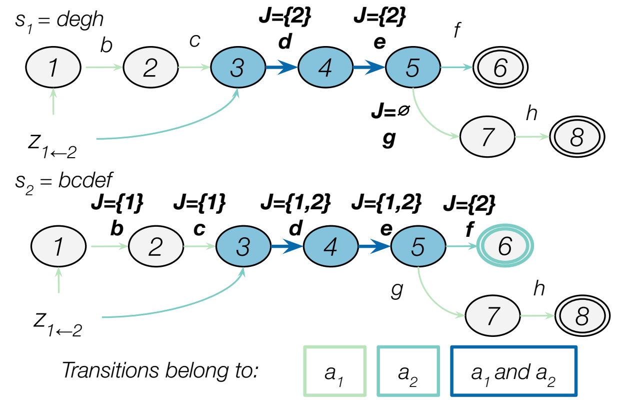


Fig. 3: Activation function J behavior during path traversal: depending on the branch taken the set of active FSAs is updated. The coloring of the transitions determines their derivation from either a_1, a_2, or both a_1 and a_2.

We can now complete the formal model of a MFSA, expanding that of standard FSAs [34]:

$$z = (Q, \Sigma, \Delta, I, F, J, \mathcal{R}) \tag{10}$$

A MFSA comprises a set Q of states, an alphabet Σ, a transition function Δ, a set of initial states $I \subseteq Q$ representing each merged FSA initial state q_0, a set of final states $F \subseteq Q$, the activation function J, and the merged FSAs identifiers $\mathcal{R}$.

To conclude, Figure 3 exemplifies MFSAs behavior during path traversal, considering a MFSA $z_{1\leftarrow 2}$ resulting from the merging of FSAs a_1 and a_2, recognizing $\mathcal{L}(a_1) = bcdegh$ and $\mathcal{L}(a_2) = def$, respectively. We analyse the behavior of $z_{1\leftarrow 2}$ against input strings $s_1 = degh$ and $s_2 = bcdef$. s_1 activates a_2 starting from state 3, matching characters d and e, which leads to state 5. State 5 marks a branch: upon reaching it the only active FSAs is a_2. However, no transition strarting from 5 with character g keeps a_2 active, yielding an empty set J of active FSAs and thus no matches. Instead, s_2 causes the activation of a_1 on state 1 and the match of characters b and c, leading to state 3. Upon reaching this state, a_2 is activated too, since 3 maps a_2's initial state. Subsequently, characters d and e are matched, for both FSAs 1 and 2. Upon reading character f at the branch starting from state 5, label 1 is discarded from the active set, since the transition $5 \xrightarrow{f} 6$ activates a_2 only. Eventually, reaching the final state 6 produces a match for a_2.

IV. Overall Compilation Framework

To fully exploit the proposed merging approach, we designed a multi-level compilation framework managing the input REs, their optimization, and tailoring towards iMFAnt. Figure 4 presents the sequence of compilation steps and their corresponding inputs and outputs. In brief, our framework comprises five steps: (1) lexical and syntactical analyses of REs, (2) conversion from RE to FSA, (3) single-FSAs optimization, (4) merging, and (5) ANML generation.

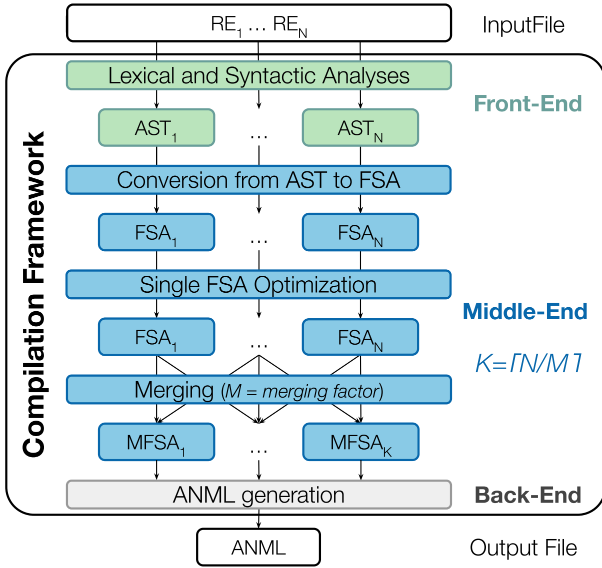
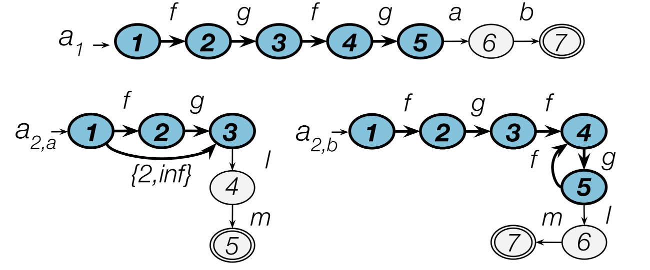
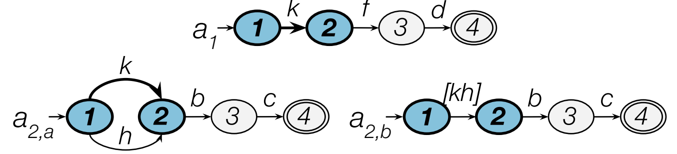


Fig. 4: Proposed compilation framework overview: Front-End for syntax and grammar checking; the Middle-End converts the REs into FSAs form, performs application-specific FSA optimizations and the proposed MFSA optimization; the Back-End generates an ANML representation suitable for iMFAnt execution. Keep in mind merging factor $M = \#merged\ FSA$



(a) Compressed loops ($a_{2,a}$) make the merging less effective, while expanded ones ($a_{2,b}$) maximize mergeable a_1 and a_2 transitions.



(b) a_1 and a_2 recognize $(k|h)bc$ and kfd, respectively. Merging transitions $1 \xrightarrow{k} 2$ in $a_{1,a}$ and $1 \xrightarrow{k} 2$ in a_2 would yield an incorrect MFSA recognizing hfd. Instead, transforming $a_{1,a}$ into $a_{1,b}$ prevents their merging, since labels $[kh]$ and k are different.

Fig. 5: Examples of how applying different pre-merging FSAs transformations facilitate FSAs merging.

A. Front-End - Lexical and Syntax Analysis

The first step of the framework is the Front-End, which includes the lexical and syntactical analyses of the input REs [35], checking their compliance with POSIX ERE [44] standard. This step outputs an Abstract Syntax Tree (AST) for each input RE, containing all the tokenized elements in a high-level syntactic structure.

B. Mid-End - from Regular Expressions to FSAs

The AST, resulting from the Front-End, is the starting point of the conversion to FSAs. For this purpose, we exploit a Thompson-like construction algorithm [34], [45]. Specifically, an AST provides all the information concerning characters and sub-pattern relationships that concur to defining the morphology a FSA. Each operator in the AST maps to a well-defined structure in the output FSA. Therefore, exploiting the advantages of the AST tree structure, the FSAs construction relies on a depth-first search to find nodes with no children (i.e., leaves), representing atomic sub-expressions by construction. After encoding an atomic sub-expression into a sub-FSA, as referenced by a leaf node, by jumping back in the tree to the parent node, it is possible to retrieve the operator acting on the sub-expression. This procedure yields FSAs that are non-deterministic for a lightweight representation.

C. Mid-End - Optimizing and Processing FSAs Before Merging

FSAs deriving from ASTs are singularly optimized and transformed to simplify the subsequent merging procedure. Specifically, these transformations are: (1) **ϵ-arcs removal** - our Thompson-like construction algorithm exploits ϵ-arcs (e.g., to connect branches of sub-REs alternation to a common final state). We remove them to simplify the merging and ANML generation[6]. In fact, ϵ-arcs add no useful information and their removal ensures that the MFSAs contain non-empty transitions only. (2) **loops expansion** - during FSAs generation, an *ad-hoc* data structure saves each loop (i.e., quantified sub-REs) and its characteristics (upper and lower bounds, involved states). This optimization expands loop structures according to their boundaries to optimize and maximize possibly mergeable states by providing additional merging paths, as in Figure 5a. (3) **simplification of arcs with multiplicity greater than 1** (i.e., single characters alternation) - we identify the multiplicity of a transition as the number of alternative paths between a couple of states, i.e., when considering single characters alternation or CCs. With no additional optimizations, merging transitions with multiplicity greater than 1 can generate incorrect paths in the final MFSA. For instance, Figure 5b shows that merging the transitions connecting states 1 and 2 would yield a MFSA indefinitely matching $(k|h)bc$, kfd, but also $(k|h)fd$, makes the MFSA able to recognize a language which belongs to neither of the initial FSAs. To manage these cases, we turn transitions with multiplicity greater than 1 into range-like structures, describing a CC. A transition labeled by a CC (e.g., a range between characters) is enabled by any of the characters the reference of the range structure contains.

These optimizations prevent the generation of incorrect MFSAs (i.e., MFSAs recognizing languages different than those described by the input FSAs) while maximizing the number of possibly mergeable states.

[6]ANML does not support ϵ-moves.

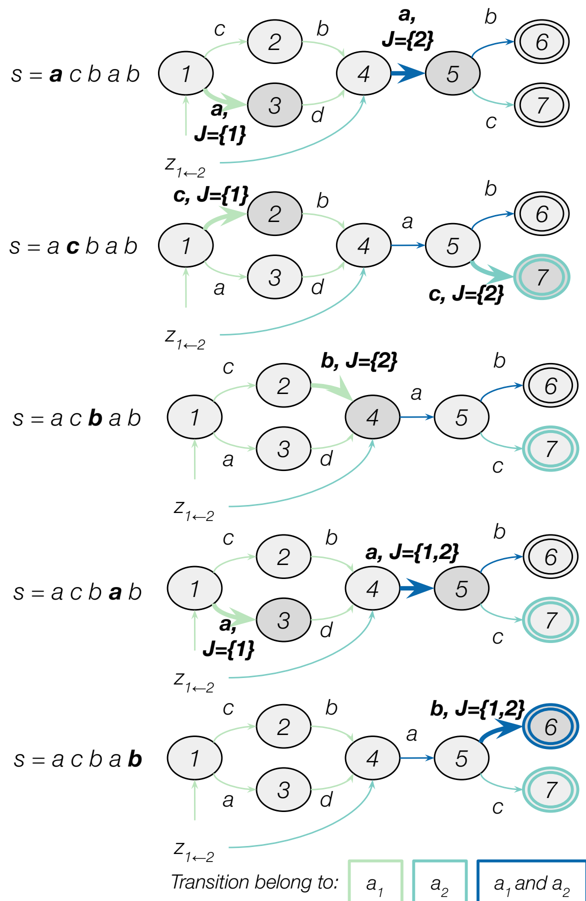


Fig. 6: Matching procedure exploiting iMFAnt, deriving from the merging of a_1 (recognizing $\mathcal{L}(a_1) = (ad|cb)ab$) and a_2 (recognizing $\mathcal{L}(a_2) = a(b|c)$). For every read character of s, iMFAnt enables all suitable arcs (i.e., initial ones or those starting from an active state). Transitions coloring indicates whether they belong to either a_1, a_2, or both a_1 and a_2.

D. Mid-End - Merging

The so-obtained optimized FSAs are then merged, depending on the input merging factor M, according to Algorithm 1 (§III-A, §III-B).

E. Back-End - Lowering the Intermediate Representation

The last step of the framework transforms the MFSAs into ANML representation suitable for the iMFAnt execution. Specifically, we extended the ANML standard to include the REs each transition belongs to support MFSAs activation function and ensure correctness of the matching procedure.

V. MFSAs Execution: from iNFAnt to iMFAnt

To enable the proper execution of the MFSA model, we extend the iNFAnt [32] algorithm, calling it iMFAnt. Standard iNFAnt [32] targets explicitly NFAs execution, enabling multiple active states simultaneously and providing high execution throughput. It relies on a data structure linking each symbol in a standard 256-characters alphabet to the transitions it enables on a FSA. Moreover, it employs a state vector sv describing whether the state at position i in the vector is active ($sv(i) = 1$) or not ($sv(i) = 0$). Since iNFAnt [32] natively supports the simultaneous exploration of all the transitions enabled by the same character, it naturally fits the simultaneous traversal of multiple active FSAs in the MFSA execution.

To support MFSAs activation function, we included in the state vector structure an additional field, indicating for each active state the result of the activation function upon reaching it. The proposed iMFAnt algorithm takes as input an extended ANML representation supporting the MFSA model, whose conversion into an iMFAnt-compliant structure is part of the algorithm pre-processing.

Intuitively, for each string input character, the iNFAnt algorithm evaluates every transition enabled by that character. If the transition starts in an initial or active state (i.e., a state reached after reading the immediately previous character), the algorithm enables the move on that transition. If no valid transition is allowed on a certain character, all active paths are discarded, and the matching procedure starts from scratch with the next character in the input string. The key requirement toward extending this algorithm supporting MFSAs is to include an additional check on transitions consistency. In particular, the algorithm must ensure that each move it performs updates the activation function J correctly, according to the rules in Equations (4) to (6).

Figure 6 details the iMFAnt behavior for a simple MFSA, deriving from the merging of a_1 and a_2, that recognize $(cb|ad)ab$ and $a(b|c)$, respectively.

1) Upon reading the first character a, the algorithm enables all the transitions the character allows ($1 \xrightarrow{a} 3$ with $J = \{1\}$, $4 \xrightarrow{a} 5$ with $J = \{2\}$), as long as they start in an initial state. Notably, although transition $4 \xrightarrow{a} 5$ belongs to both FSAs a_1 and a_2, this transition only activates the match for FSA 2, since state 4 is only initial for a_2.
2) Moving on to character c, the algorithm activates transitions $1 \xrightarrow{c} 2$ with $J = \{1\}$ (initial, active for A_1) and $5 \xrightarrow{c} 7$ with $J = \{2\}$ (state 5 was previously activated for a_2, which is consistent with this transition). The last transition determines a match on state 7 for a_2, i.e., the only active FSA on the current path. The previously active state 3 is then discarded since no valid transition starting in 3 exists for character c.
3) After reading character b, the algorithm enables the only available transition $2 \xrightarrow{b} 4$ with $J = \{1\}$. However, since state 4 is initial for a_2, J is updated upon reaching it with $J(4) = \{1, 2\}$.
4) Reading character a yields two transitions: $1 \xrightarrow{a} 3$ with $J = \{1\}$ (initial for A_1) and $4 \xrightarrow{a} 5$ with $J = \{1, 2\}$. The second transition now activates both 1 and 2 because the current matching path differs from the one at 1).
5) The final character b activates transition $5 \xrightarrow{b} 6$ with $J = \{1, 2\}$. This last transition yields a match for FSAs 1 and 2, both active upon reaching the final state 6.

TABLE I: Datasets main characteristics.

Dataset	Bro217	Dotstar09	PowerEN	Protomata	Ranges1	TCP-ext. homenet
Ref.	[30]		[29]		[30]	
Abbr.	BRO	DS9	PEN	PRO	RG1	TCP
Num. REs	217	299	300	300	299	300
Tot. N_S ≀	2863	12883	4726	3704	12913	9105
Tot. N_{TS} †	2645	12614	4554	3400	12644	8906
Tot. N_{CC} ‡	2791	2031	152	11905	1689	2341
Avg. N_S ≀	13.19	43.08	15.75	12.34	43.18	30.35
Avg. N_{TS} †	12.19	42.19	15.18	11.33	42.29	29.69

≀ Number of states † Number of transitions ‡ Length of CCs in the FSA

In conclusion, executing this MFSA against the input string $acbab$ yields three matches: ac and ab for a_2, $cbab$ for a_1.

The novel iMFAnt algorithm suits the MFSA *activation function* concept, preventing false positives over-matching during the algorithm execution.

VI. Experimental Evaluation

We implement the front-end of our framework through standard Flex and Bison while the middle-end, the back-end, and iMFAnt in C++ and compile with *-O3*. Firstly, §VI-A analyzes the transition and state compression percentage of the MFSA. Then, §VI-B describes the compilation time overhead due to our framework. Finally, §VI-C evaluates iMFAnt execution time and throughput by varying the merging factors M from 1 to the dataset size (i.e., 217 to 300 REs), and number of threads T, from 1 to 128.

We employ an Intel i7-6700 CPU (4 cores, 8 threads) for all the time measurements. We select six benchmarks based on widely employed benchmark suites: four from Becchi et al. [30] and two from Wadden et al. [29]. Table I details the datasets showing the number of employed REs, the abbreviations we adopted for the sake of compactness, the average and total number of states, transitions, and CCs with their length. Whenever applying the MFSA merging methodology with a merging factor M, we are sampling the input M REs sequentially from the dataset.

A. Automata Compression Evaluation

Automata optimization works focus on the compression as a metric directly impacting the representation of the FSAs, hence their memory footprint [33], [39], [40]. We asses the reduction of number of states and transitions with increasing merging factor M compared to standard FSAs characteristics (Table I). Given a dataset of FSAs $a \in \mathcal{A}$ to merge into a final set of MFSAs $z \in \mathcal{Z}$, we compute the compression percentage $\%comp$ as follows:

$$\%comp_{states} = \frac{\Sigma_{a\in\mathcal{A}} \#states_a - \Sigma_{z\in\mathcal{Z}} \#states_z}{\Sigma_{a\in\mathcal{A}} \#states_a} \cdot 100$$

$$\%comp_{trans} = \frac{\Sigma_{a\in\mathcal{A}} \#trans_a - \Sigma_{z\in\mathcal{Z}} \#trans_z}{\Sigma_{a\in\mathcal{A}} \#trans_a} \cdot 100$$

Figure 7 displays the compression performance of our framework with variable merging factor M, ranging from 1 (single-FSAs case) to the dataset size (all). Specifically, the left plot presents the state reduction, while the right one describes the variation in transitions number. On average, the merging factor configuration of $M = all$ is the best for compression as it achieves 71.95% and 38.88% in states and transitions number reduction, respectively. Moreover, Figure 7 shows how the impact of merging is significantly higher in state reduction. This trend is intrinsic to the merging algorithm, which represents common transitions once, without replicating them for each FSA they belong to. Merging one transition implies merging both its starting and arrival states, hence the increased impact concerning states.

Finally, in both transitions and states cases, there is a plateau in the compression trend. This behavior is because any alphabet has a finite number of characters. Upon building an MFSA with as many transitions as the characters in the dataset's alphabet, the space for further optimization decreases. A possible improvement stands in the CC transitions matching optimization. We currently merge CCs that describe the same exact set of characters, while it could be possible to partially merge two CCs based on the characters belonging to both. For instance, in CCs `[abce]` and `[bcd]` it could be possible to merge the common characters `[bc]` only. Although $INDEL$ metric (Figure 1) represents a preliminary estimate of REs merging potential, these results confirm the impact of exploiting similarities among REs.

To sum up, the merging optimization halves in average the necessary number of states representing a REs set.

B. Compilation Stages Time Analysis

The assessment of compilation stages evaluates the time to output a set of MFSA in ANML format from the entire REs dataset (Table I) with variable merging factor M. Figure 8 shows the total execution time of our framework considering a certain merging factor and the impact each compilation stage has in this process, from the initial REs analysis to the generation of an output ANML file. We obtained these results by averaging 30 executions of our compilation framework considering the multiple REs of the dataset. Figure 8 highlights how the impact of single-FSA-related stages (front-end, AST to FSA conversion, FSAs single optimization) is independent of M. In particular, considering all configurations in Figure 8, these steps take on average $1.29\,\mathrm{ms}$ (front-end), $1.33\,\mathrm{ms}$ (AST to FSA conversion), $2.03\,\mathrm{ms}$ (single-FSA optimization).

Instead, the merging stage is responsible for the significant variation in the overall compilation time. On average, in the most computationally-intensive configuration (i.e., $M = all$), the merging procedure takes $6.65\,\mathrm{s}$ to complete, yielding an entire compilation-time of $6.66\,\mathrm{s}$ on average.

Even if, as Figure 8 illustrates, the merging step during the dataset compilation has the highest impact, the overhead is limited concerning the execution times shown in §VI-C, emphasizing the benefits of merging multiple REs.

C. iMFAnt Throughput and MFSA Execution Impact

The execution impact evaluation considers the novel iMFAnt algorithm against a $1\,\mathrm{MB}$ data input stream with the previously

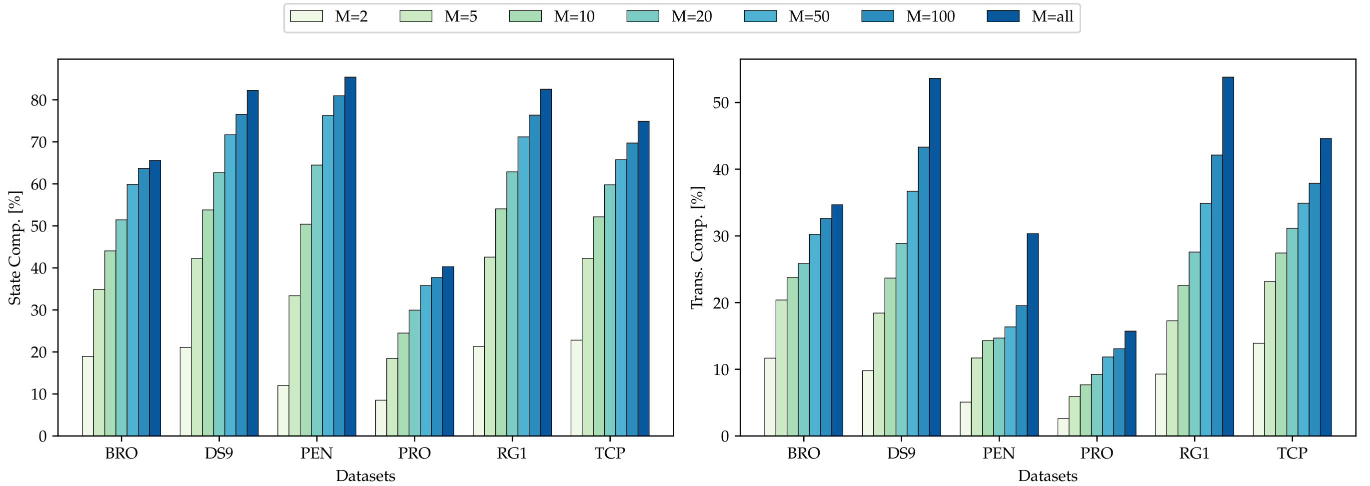


Fig. 7: Memory footprint reduction in terms of states and transitions number, considering different merging factors: $M = 1$ (i.e., no merging), $M = 2, 5, 10, 50, 100, all$ (the higher, the better).

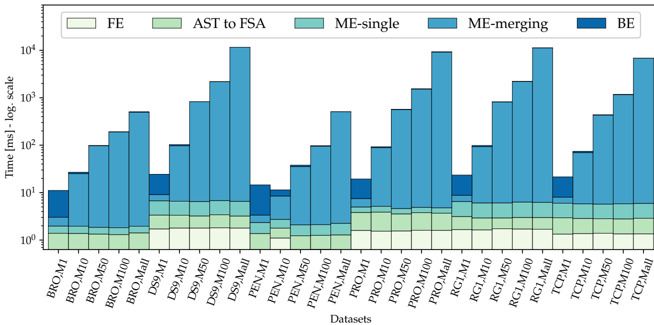


Fig. 8: Impact of each compilation stage in the overall procedure (the most significant M values), in logarithmic scale (the lower, the better): front-end (FE), AST-to-FSA conversion (AST to FSA), middle-end optimization for single FSAs (ME-single), middle-end merging optimization (ME-merging) and finally ANML generation (BE).

TABLE II: Datasets execution characteristics.

Abbr.	BRO	DS9	PEN	PRO	RG1	TCP
Avg. N_{ac}¶	10.73	38.02	21.27	101.8	6.55	4.55
Max N_{ac}¶	40	90	39	652	63	149

¶ Total number of active FSAs during MFSAs traversal ($M = all$)

compiled MFSAs. To increase the reliability of the results, we averaged the execution time of 15 iMFAnt runs. We compare against single-thread (§VI-C1) and multi-thread (§VI-C2) ruleset executions considering the execution time and the throughput improvement against $M = 1$ configuration with increasing values of M from 2 to the whole dataset. Specifically, the throughput is calculated as:

$$th_{MFSA} = \frac{\#RE_{exe} \cdot D_{size}}{Exe_time_{tot}} = \frac{\#MFSA \cdot M \cdot D_{size}}{Exe_time_{tot}} \quad (11)$$

where M is the merging factor, Exe_time_{tot} is the sum of all single REs execution time, and D_{size} is the size of the data to analyze. The throughput computation evaluates the number of processed REs against the entire input string. Specifically, $\#MFSA \cdot M$ represent the different analyzed REs.

1) Single-Threaded Execution: Figure 9 illustrates the iMFAnt scalability across different benchmarks for the single thread. Notably, the datasets display variable performance peaks: while in some cases (i.e., BRO, RG1, TCP, PEN) merging the entire dataset ($M = all$) yields the lowest execution time, others display a dissimilar optimum value of M (i.e., 100 and 10 for DS9 and PRO, respectively). Differently from the others, PRO and DS9 exhibit this behavior due to the high average number of active REs per read symbol, which spans from 40 to 100 approximately (Table II). This introduces additional complexity for iMFAnt to manage numerous active REs and partial matches spread in the MFSA. Considering the throughput, the MFSA with iMFAnt always leads an improvement against the single-FSA with a geometric mean spreading from 1.47× (M=2) to 5.44× (M=100). Considering the best MFSA configuration for each dataset, iMFAnt engine yields 5.99× of geomean improvement.

2) Multi-Threaded Execution: The straightforward approach to improve the throughput distributes multiple REs among an increasing number of threads. In this evaluation, for each benchmark, we distribute the MFSAs over a pool of a fixed number of available threads. Each thread manages different automata asynchronously, selecting an MFSA at a time from the remaining ones until all are executed. The measured execution time represents the latency to compute all the REs of a benchmark. Figure 10 summarizes the scalability (1 to 128 threads) evaluation of the naive approach against ours, executing MFSAs on the available threads. We consider two performance indicators: one to address time-critical applications, reducing the execution time as much as

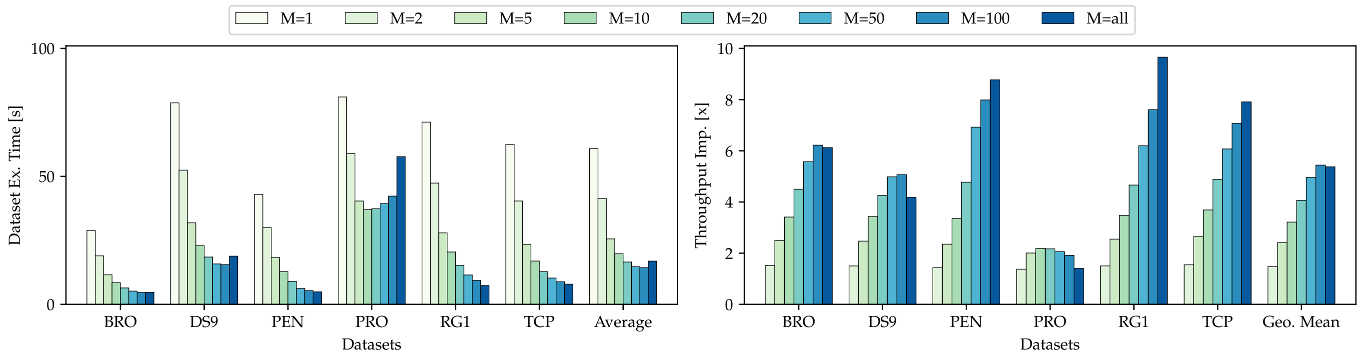


Fig. 9: Performance evaluation in terms of dataset execution time and throughput, considering iMFAnt algorithm against a 1 MB input stream with $M \in [1, all]$.

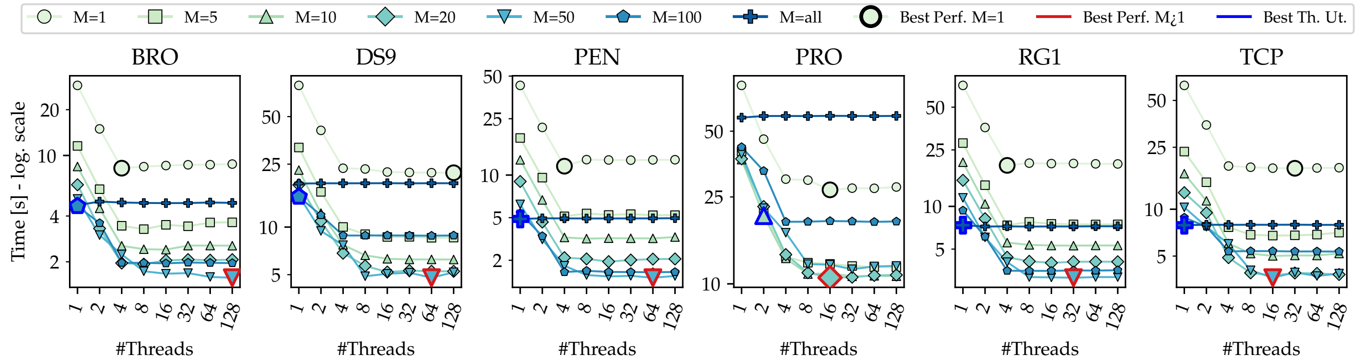


Fig. 10: iMFAnt execution time (logarithmic scale) considering merging factors $M \in [1, all]$ and scaling the number of threads $T \in [1, 128]$ (max hardware threads in the CPU 8). Highlighted markers describe the top performing single-FSAs configurations (Best Perf. M=1), the top performing MFSAs configuration (Best Perf. M>1), and the configuration reaching the same execution time as top performing single-FSAs with the least number of threads (Best Th. Ut.).

possible, and another for thread-critical applications, reducing the thread number without impairing the execution time. Firstly, we evaluate the speedup granted by MFSAs multi-threaded execution (red border in Figure 10) against the top-performing multi-threaded single-FSAs (bigger circle and a black border in Figure 10). Secondly, we point out MFSAs configurations reaching single-FSAs top performance with the least number of threads (blue border in Figure 10).

Although there is no pre-defined optimal M applying for every dataset, the majority of MFSAs with $M > 1$ overcomes parallel multi-threaded FSAs execution time. Moreover, $M = all$ always improves the execution time with a geomean speedup per benchmark ranging from 1.5× (DS9) to 3.47× (RG1). The only exception is PRO, where scaling the thread number for single-FSAs improves against the $M = all$. Nevertheless, all the other merging factors overcome the single-FSAs performance. Considering the optimal improvement our approach achieves, $M = 50$ is the best configuration in the majority of the datasets. Instead, PRO benchmark exhibits $M = 20$ as top merging factor. Overall, with the best configuration per the parallel multi-thread FSAs and our best MFSA per benchmark, we achieve top speedups ranging from 2.52× (PRO) to 6.18× (RG1) with a geomean of 4.05×.

Concerning threads utilization, MFSAs reach the same (or slightly better) top performance of the parallel multi-threaded FSAs with one or two threads at most. Indeed, the blue border of Figure 10 highlights that $M = all$ is the best configuration of thread-critical scenarios for three datasets (PEN, RG1, TCP). Instead, DS9, BRO, and PRO have in $M = 100$ (DS9, BRO) and $M = 20$ (PRO) their top thread utilization improvement.

Figure 10 also shows how the execution time halves on average for both FSAs and MFSAs when doubling the number of threads, with the upper bound of 4 physical cores. On top of this, the MFSA model also halves the execution time compared to single FSAs, across the increasing number of threads. This suggests that an MFSA-based approach can be beneficial with an increasing number of physical cores, as long as the number of REs to analyze remains sufficiently bigger than the number of cores, as for real-case scenarios.

To summarize, the analysis of iMFAnt behavior in multi-threaded configuration further highlights the benefits of exploiting MFSAs for pattern matching purposes. Specifically, MFSAs execution achieves and overcomes the same performance as single FSAs, even relying on a lower number of threads. This makes the use of MFSAs more advantageous in resource-constrained applications such as networking ones [2], [3], [19],

[24], [25], where MFSAs and iMFAnt can save precious CPU clock cycles.

VII. Related Work

To fully exploit FSAs executive potential, there exist various techniques fostering the optimization of both NFAs and DFAs that tackle NFAs partitioning [26], multi-stride DFAs [11], [28], [40], and DFAs compression [33], [46]. These optimization approaches involve both architectural [6], [13], [20]–[23], [26], [41], [47] and algorithmic aspects [11], [12], [14], [33]. An architectural approach to optimizing NFAs [26] relies on a toolchain to partition them, depending on the available hardware resources and units. Given a constraint on the number of NFA states that can fit onto a specific hardware component, the partitioning algorithm aims at splitting the NFA while minimizing the state replication. However, this approach misses generality, since it strongly depends on the underlying architecture specifications.

Concerning DFAs optimization, an algorithmic approach to reduce their memory footprint increases the number of bytes consumed in the text, per state-traversal, via an inverse homomorphism [11]. Such inverse homomorphism enables the faster construction of a k-step (i.e., multi-stride) DFA, an automaton that consumes k symbols per state-traversal. In general, multi-stride automata provide another common approach to optimizing DFA. However, their complexity [40] significantly affects their performance since it comprises all the k-characters combinations of adjacent transitions.

Concerning the compression of DFAs, an entire category of algorithms [33], [39], [48] relies on the presence of multiple incoming transitions with identical labels for a specific state q. One such transition can be assumed to be a *default transition* toward state q. Thus, its representation is negligible, yielding a DFA representation that only comprehends the non-default transitions. However, evaluating these works from an executive perspective is extremely difficult since it requires adapting pattern matching algorithms to the *default-transition*-dependent data structure.

Differently, Wang et al. [6] approach optimize pattern matching via REs decomposition to extract simpler strings out of complex ones, matching them separately and eventually re-composing their results. Others focus on the execution improvements of counting in REs and FSAs [12]–[14] focusing either on decomposing at the bit level the automaton or on software-hardware codesigned in-memory architectures.

VIII. Conclusions and Future Work

The presented work advocates that exploiting REs similarities within a ruleset pushes the REs matching further. Specifically, we present the MFSA model, detailing its construction and formal properties to enable multiple distinct REs recognition. We devise a multi-level compilation framework that takes REs rulesets in input, transforms and optimizes the FSA representation, merges into the MFSAs, and produces the ANML. Moreover, we illustrate an *ad-hoc* extended execution algorithm called iMFAnt to support MFSAs [49].

Overall, the MFSAs significantly compresses the state (71.95%) and transition (38.88%) number of FSAs equivalent representation, and, with iMFAnt, delivers a geomean of $5.99\times$ of throughput improvement and $4.05\times$ speedup for the best single-threaded and multi-threaded configuration.

Future Work. We will expand the support for MFSAs across different algorithms and architectures and considering even non-regular operators such as backreferences [50]. Moreover, we plan to devise a systematic similarity RE analysis for possible clustering techniques.

Acknowledgment

The authors are grateful to the anonymous reviewers and Eleonora D'Arnese for their insightful feedback.

iMFAnt CGO 2024 Artifact

A. Artifact Check-List (Meta-Information)

- **Compilation:** C++, Makefile, Bash
- **Data set:** REs from ANMLZoo [29] and from [30] and 1MB input streams
- **Run-time environment:** Ubuntu
- **Metrics:** Percentage compression (%), compilation time (ms) and execution time (s)
- **Output:** Raw results, PDF charts, 5GB of disk space
- **Experiments:** Replicate Figs. 7, 8, 9, 10
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 15 hours
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT License
- **Data licenses (if publicly available)?:** Refer to single benchmarks licenses
- **Archived (provide DOI):** https://zenodo.org/doi/10.5281/zenodo.10475372

B. Description

1) How to Access: Code publicly and benchmarks publicly available at https://github.com/necst/iMFAnt with the following DOI https://zenodo.org/doi/10.5281/zenodo.10475372

2) Software Dependencies:

- Requires Python 3.10, including numpy, pandas, and matplotlib for plots
- Requires OpenMP

3) Data Sets: The datasets are available in the corresponding repositories. A copy of the executed REs and input stream is available in the `dataset` and `input_stream` folder.

C. Installation

Clone the repo:

```
git clone https://github.com/necst/iMFAnt.git
```

Or build and run the docker:

```
run_docker.sh
```

Go to the scripts folder:

```
cd iMFAnt/scripts/
```

D. Experiment Workflow

The repository already contains the datasets necessary for the experimental evaluation. The scripts in folder `scripts` reproduce the figures reported in the paper.

1) `./compilation_time.sh`

   reproduces Fig. 8 (ETA: 30 min with 30 reps)

2) `./compression.sh`

   reproduces Fig. 7 (ETA: 5 min)

3) `./iMFAnt_performance.sh`

   reproduces Figs. 9 and 10 (ETA: 12 h with 15 reps)

E. Evaluation and Expected Results

The scripts in folder `scripts/` reproduce the experimental results reported in the paper. These include the automatic generation of the plots with the default merging factors (1, 2, 5, 10, 20, 50, 100, all).

To reproduce the results run the desired scripts. The resulting plots will be in `iMFAnt/plots` folder. To reduce the time required modify the repetitions with `-r`.

Figure 9 and 10 are expected to display different optimal merging factor according to testing machine characteristics. Anyhow, the MFSAs are expected to improve the simple multi-thread scaling according to the trends in the paper.

F. Experiment Customization

- To customize the generation of automata enter the compilation framework and build the compiler:

  ```
  cd framework/compiler && bison -d compiler.yy -v
      flex -o compiler.yy.cc compiler.lex
  ```

  Run

  ```
  g++ -o compiler compiler.yy.cc compiler.tab.cc
      ast.cpp ../re2automata/re2automata.cpp -
      DSTATES=1 -DSTACK_TIME=0 -w -std=c++11 -g -
      O3
  ```

  Set `STATES=1` to evaluate states compression, `0` otherwise. Set `STACK_TIME = 1` to measure compilation time. Run

  ```
  python merging.py -r REPS -b M1 ... Mn
  ```

  to generate the automata for the benchmark datasets. Flag `-r` indicates the number of repetitions to evaluate the compilation time, and `-b` is followed by the list of merging factors `M1 ... Mn`. Any value for M is valid. To merge the entire dataset, set M=0. To run the merging engine, the merging factors in `matching/imfant.py` must be consistent with the ones of the generated `MFSA`.

- To analyze a specific folder of MFSAs, first compile the matching algorithm:

  ```
  cd matching/ && make clean all
  ```

  To change the number of repetitions (default: 15), enter the Makefile (`matching/Makefile`) and change the value of `-DREPS` variable as desired, in the compilation command. Then, run:

  ```
  export OMP_NUM_THREADS=N; make &&
  ./multithreaded_imfant STREAM_IN MFSA_DIR NUM
      OUTPUT_FILE
  ```

  where `N` is the number of threads, `STREAM_IN` is the input stream to be matched, `MFSA_DIR` is the directory of the folder containing the MFSAs to match, `NUM` is the number of MFSAs to match in that directory, and `OUTPUT_FILE` contains the results of the matching (matching time, #matches)

- To merge and analyze a new dataset save the RE file and the input stream file in `dataset` and `input_streams` folders, respectively. Add the RE directory into `framework/compiler/merging.py` file in `inputRE` variable and run it to perform the merging, setting flags `-r` and `-r` as desired. To pattern-match a new dataset, add the corresponding `mfsa` and `input_streams` directories to `matching/imfant.py` file and run it, ensuring that the merging factors are consistent with the ones employed during the merging.

G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] C. R. Panigrahi, M. Tiwari, B. Pati, and R. Prasath, "Malware detection in big data using fast pattern matching: A hadoop based comparison on gpu," in *Mining Intelligence and Knowledge Exploration*. Springer, 2014, pp. 407–416. [Online]. Available: https://doi.org/10.1007/978-3-319-13817-6_39

[2] C. Xu, S. Chen, J. Su, S.-M. Yiu, and L. C. Hui, "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 2991–3029, 2016. [Online]. Available: https://doi.org/10.1109/COMST.2016.2566669

[3] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 1083–1100. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[4] Cisco, "Snort - open source intrusion prevention system (ips)," 2023. [Online]. Available: https://www.snort.org/

[5] ——, "Clamav®: An open-source antivirus engine for detecting trojans, viruses, malware & other malicious threats." 2023. [Online]. Available: http://www.clamav.net

[6] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: A fast multi-pattern regex matcher for modern {CPUs}," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 631–648. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[7] T. Tanjo, Y. Kawai, K. Tokunaga, O. Ogasawara, and M. Nagasaki, "Practical guide for managing large-scale human genome data in research," *Journal of Human Genetics*, vol. 66, no. 1, pp. 39–52, 2021. [Online]. Available: https://doi.org/10.1038/s10038-020-00862-1

[8] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Jagadish, "Regular expression learning for information extraction," in *Proceedings of the 2008 conference on empirical methods in natural language processing*, 2008, pp. 21–30. [Online]. Available: https://doi.org/10.5555/1613715.1613719

[9] Z. István, D. Sidler, and G. Alonso, "Runtime parameterizable regular expression operators for databases," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 204–211. [Online]. Available: https://doi.org/10.1109/FCCM.2016.61

[10] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating pattern matching queries in hybrid cpu-fpga architectures," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 403–415. [Online]. Available: https://doi.org/10.1145/3035918.3035954

[11] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "Faster dfas through simple and efficient inverse homomorphisms," in *IEEE INFOCOM 2009*. IEEE, 2009, pp. 2851–2855. [Online]. Available: https://doi.org/10.1109/INFCOM.2009.5062245

[12] L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar, "Regex matching with counting-set automata," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020. [Online]. Available: https://doi.org/10.1145/3428286

[13] L. Kong, Q. Yu, A. Chattopadhyay, A. Le Glaunec, Y. Huang, K. Mamouras, and K. Yang, "Software-hardware codesign for efficient in-memory regular pattern matching," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 733–748. [Online]. Available: https://doi.org/10.1145/3519939.3523456

[14] A. Le Glaunec, L. Kong, and K. Mamouras, "Regular expression matching using bit vector automata," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 492–521, 2023. [Online]. Available: https://doi.org/10.1145/3586044

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022. [Online]. Available: http://mitpress.mit.edu/9780262046305/introduction-to-algorithms/

[16] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

[17] J. Qiu, X. Sun, A. H. N. Sabet, and Z. Zhao, "Scalable fsm parallelization via path fusion and higher-order speculation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 887–901. [Online]. Available: https://doi.org/10.1145/3445814.3446705

[18] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 529–542. [Online]. Available: https://doi.org/10.1145/2541940.2541988

[19] J. van Lunteren and A. Guanella, "Hardware-accelerated regular expression matching at multiple tens of gb/s," in *Proceedings of IEEE INFOCOM*. IEEE, 2012, pp. 1737–1745. [Online]. Available: https://doi.org/10.1109/INFCOM.2012.6195546

[20] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783747

[21] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014. [Online]. Available: https://doi.org/10.1109/TPDS.2014.8

[22] D. Conficconi, E. Del Sozzo, F. Carloni, A. Comodi, A. Scolari, and M. D. Santambrogio, "An energy-efficient domain-specific architecture for regular expressions," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 1, pp. 3–17, 2022. [Online]. Available: https://doi.org/10.1109/TETC.2022.3157948

[23] D. Parravicini, D. Conficconi, E. D. Sozzo, C. Pilato, and M. D. Santambrogio, "Cicero: A domain-specific architecture for efficient regular expression matching," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–24, 2021. [Online]. Available: https://doi.org/10.1145/3476982

[24] S. Miano, A. Sanaee, F. Risso, G. Rétvári, and G. Antichi, "Domain specific run time optimization for software data planes," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1148–1164. [Online]. Available: https://doi.org/10.1145/3503222.3507769

[25] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[26] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi, "Demystifying automata processing: Gpus, fpgas or micron's ap?" in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–11. [Online]. Available: https://doi.org/10.1109/IPDPS.2014.51

[27] H. Liu, S. Pai, and A. Jog, "Why gpus are slow at executing nfas and how to make them faster," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 251–265. [Online]. Available: https://doi.org/10.1145/3373376.3378471

[28] M. Avalle, F. Risso, and R. Sisto, "Scalable algorithms for nfa multi-striding and nfa-based deep packet inspection on gpus," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1704–1717, 2015. [Online]. Available: https://doi.org/10.1109/TNET.2015.2429918

[29] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan *et al.*, "Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/IISWC.2016.7581271

[30] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 79–89. [Online]. Available: https://doi.org/10.1109/IISWC.2008.4636093

[31] H. Hyyrö, Y. Pinzon, and A. Shinohara, "New bit-parallel indel-distance algorithm," in *Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005. Proceedings 4*. Springer, 2005, pp. 380–390. [Online]. Available: https://doi.org/10.1007/11427186_33

[32] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "infant: Nfa pattern matching on gpgpu devices," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 20–26, 2010. [Online]. Available: https://doi.org/10.1145/1880153.1880157

[33] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, 2007, pp. 145–154. [Online]. Available: https://doi.org/10.1145/1323548.1323573

[34] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001. [Online]. Available: https://dl.acm.org/doi/10.5555/1196416

[35] A. V. Aho and J. Ullman, "The theory of parsing, translation and compiling. parsing, vol. i," 1972. [Online]. Available: https://dl.acm.org/doi/book/10.5555/578789

[36] S. C. Reghizzi, L. Breveglieri, and A. Morzenti, *Formal languages and compilation*. Springer, 2013. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4471-5514-0

[37] V. M. Glushkov, "The abstract theory of automata," *Russian Mathematical Surveys*, vol. 16, no. 5, p. 1, 1961. [Online]. Available: https://dx.doi.org/10.1070/RM1961v016n05ABEH004112

[38] J. Patel, A. X. Liu, and E. Torng, "Bypassing space explosion in high-speed regular expression matching," *IEEE/ACM Transactions on Networking*, vol. 22, no. 6, pp. 1701–1714, 2014. [Online]. Available: https://doi.org/10.1109/TNET.2014.2309014

[39] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved dfa for fast regular expression matching," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008. [Online]. Available: https://doi.org/10.1145/1452335.1452339

[40] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 50–59. [Online]. Available: https://doi.org/10.1145/1477942.1477950

[41] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching," in *2020 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2020, pp. 86–98. [Online]. Available: https://doi.org/10.1109/HPCA47549.2020.00017

[42] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *Proceedings of the 48th International Symposium*

on Microarchitecture, 2015, pp. 533–545. [Online]. Available: https://doi.org/10.1145/2830772.2830809

[43] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong, "Gpu-based nfa implementation for memory efficient high speed regular expression matching," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 129–140. [Online]. Available: https://doi.org/10.1145/2145816.2145833

[44] . e. I. S. .-. R. o. I. S. .-. The Open Group Base Specifications Issue 7, "Portable operating system interface (posix). base definitions and headers, section 9, regular expressions." 2018. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/

[45] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE transactions on Electronic Computers*, no. 1, pp. 39–47, 1960. [Online]. Available: https://doi.org/10.1109/TEC.1960.5221603

[46] F. Carloni, D. Conficconi, I. Moschetto, and M. D. Santambrogio, "Yarb: a methodology to characterize regular expression matching on heterogeneous systems," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ISCAS46773.2023.10181547

[47] F. Carloni, L. Panseri, D. Conficconi, M. Sironi, and M. D. Santambrogio, "Enabling efficient regular expression matching at the edge through domain-specific architectures," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2023, pp. 71–74. [Online]. Available: https://doi.org/10.1109/IPDPSW59300.2023.00023

[48] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *ACM SIGCOMM computer communication review*, vol. 36, no. 4, pp. 339–350, 2006. [Online]. Available: https://doi.org/10.1145/1151659.1159952

[49] L. Cicolini, F. Carloni, M. D. Santambrogio, and D. Conficconi, "Artifact repository of one automaton to rule them all: beyond multiple regular expressions execution," 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10475372

[50] D. Moseley, M. Nishio, J. Perez Rodriguez, O. Saarikivi, S. Toub, M. Veanes, T. Wan, and E. Xu, "Derivative based nonbacktracking real-world regex matching with backtracking semantics," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1026–1049, 2023. [Online]. Available: https://doi.org/10.1145/3591262