Software-Hardware Codesign for Efficient In-Memory Regular Pattern Matching

Lingkun Kong*
Rice University
USA
klk@rice.edu

Qixuan Yu*
Rice University
USA
qy12@rice.edu

Agnishom Chattopadhyay
Rice University
USA
agnishom@rice.edu

Alexis Le Glaunec
Rice University
USA
alexis.leglaunec@rice.edu

Yi Huang
Rice University
USA
781013488@qq.com

Konstantinos Mamouras
Rice University
USA
mamouras@rice.edu

Kaiyuan Yang
Rice University
USA
kyang@rice.edu

Abstract

Regular pattern matching is used in numerous application domains, including text processing, bioinformatics, and network security. Patterns are typically expressed with an extended syntax of regular expressions. This syntax includes the computationally challenging construct of bounded repetition or counting, which describes the repetition of a pattern a fixed number of times. We develop a specialized in-memory hardware architecture that integrates counter and bit vector modules into a state-of-the-art in-memory NFA accelerator. The design is inspired by the theoretical model of nondeterministic counter automata (NCA). A key feature of our approach is that we statically analyze regular expressions to determine bounds on the amount of memory needed for the occurrences of bounded repetition. The results of this analysis are used by a regex-to-hardware compiler in order to make an appropriate selection of counter or bit vector modules. We evaluate our hardware implementation using a simulator based on circuit parameters collected by SPICE simulation in TSMC 28nm CMOS process. We find that the use of counter and bit vector modules outperforms unfolding solutions by orders of magnitude. Experiments concerning realistic workloads show up to 76% energy reduction and 58% area reduction in comparison to CAMA, a recently proposed in-memory NFA accelerator.

CCS Concepts: • **Theory of computation → Formal languages and automata theory**; • **Hardware → Emerging architectures**.

Keywords: automata theory, computer architecture

ACM Reference Format:
Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. 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 (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523456

1 Introduction

Regular pattern matching, where the patterns are expressed with finite-state automata or regular expressions, has numerous applications in text search and analysis [1], network security [69], bioinformatics [9, 42], and runtime verification [6, 7]. Various techniques have been developed for matching regular patterns, many of which are based on the execution of deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). DFA-based techniques are generally faster, as the processing of an input element requires a single memory lookup, while NFA-based techniques are slower, as they involve extending several execution paths when processing one element. The advantage of NFAs over DFAs is that they are typically more memory-efficient, and there are cases where an equivalent DFA would unavoidably be exponentially larger [34].

*These authors contributed equally.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523456

Many applications require the processing of large and complex NFAs on real-time streams of data collected from sensors, networks, and various system traces. Energy efficiency and memory efficiency (in terms of the memory capacity or chip footprint needed for a given NFA) are highly desirable for both high-performance computing and battery-powered embedded applications. NFA processing requires frequent, yet irregular and unpredictable, memory accesses on general-purpose processors, leading to limited throughput and high power on CPU and GPU architectures [27, 30, 61]. Field Programmable Gate Arrays (FPGAs) offer high speed through hardware-level parallelism, but are often bottlenecked by routing congestion [40, 66] and their high power, area and cost prevent their use in mobile and embedded devices. Even with digital application-specific integrated circuit (ASIC) accelerators, the memory access bandwidth restricts the parallelism [31, 56]. The latest hardware technology that addresses these challenges is in-memory architecture, which processes the NFA transitions directly inside memories with massive parallelism and merged memory and computing operations. For instance, the Automata Processor (AP) from Micron [19, 64] outperforms x86 CPUs by 256×, GPGPUs by 32×, and the digital accelerator XeonPhi by 62× in the ANMLZoo benchmark suite [54, 61].

Classical regular expressions (regexes) involve operators for concatenation $\cdot$, nondeterministic choice $+$, and iteration (Kleene's star) *. They can be translated into NFAs whose size is linear in the size of the regex [21, 57]. However, the regexes used in practice have several additional features that make them more succinct. One such feature is *counting*, written as $r\{m, n\}$, which is also called *constrained* or *bounded repetition*. The pattern $r\{m, n\}$ expresses that the subpattern r is repeated anywhere from m to n times. This counting operator is ubiquitous in practical use cases of regexes. For example, we have observed that in several datasets for network intrusion detection (Snort [50] and Suricata [55]) and motif search in biological sequences (Protomata [39, 42]) counting arises in the majority of the patterns. The naive approach for dealing with counting operators is to rewrite them by unfolding. For example, $r\{n, n\}$ is unfolded into $r \cdot r \cdots r$ (n-fold concatenation) and results in an NFA of size linear in n (and therefore can produce a DFA of size exponential in n). Since n can grow very large, dealing with counting is one of the main technical challenges for successfully using hardware-based approaches to execute practical regular patterns.

Existing in-memory NFA architectures use this naive unfolding method to handle counting operators. This leads to the use of a large number of STEs[1] to support counting. In AP [19] and CA (Cache Automaton) [54], each STE uses 256 memory bits for 8-bit symbols. In the latest Impala [46] and CAMA[2] [26] designs, each STE requires 16 to 32 memory bits. Even with this improvement, a modest counting operator with upper limit 1024 requires at least 16384 memory bits, while the information required for implementing the operator may be only 10 bits in some cases. Unfolding counting operators results in large memory and energy usage. To circumvent these problems, we explore software and hardware co-design for integrating counter and bit vector modules into a state-of-the-art in-memory NFA architecture.

Our design is inspired by an extension of NFAs with counter registers called nondeterministic counter automata (NCAs). In an NCA, a computation path involves not only transitions between control states, but also the use of a finite number of registers that hold nonnegative integers. Such automata are a natural execution model for regexes with counting, as the counters can track the number of repetitions of subpatterns. When the counters are bounded, NCAs are expressively equivalent to NFAs, but they can be exponentially more succinct [34, 53]. Similar to how an NFA is executed by maintaining the set of active states, an NCA is executed by maintaining a set of pairs, which we call *tokens*, where the first component is the control state and the second component specifies the values of the counters. A key idea of our approach is that we can statically analyze an NCA to determine which states can carry a large number of tokens during execution. We call a control state ***counter-unambiguous*** if it can only carry at most one token and *counter-ambiguous* if it can carry more than one. In the case of counter-unambiguity for a state q with counter x, we know that we only need to record one counter value, which means that we need only one memory location whose size (in bits) is logarithmic in the range M of possible counter values. In the case of counter-ambiguity for q with counter x, we may have to record a large number of counter values (as large as M), and our insight is to use a bit vector v of size M, where $v[i] = 1$ (resp., $v[i] = 0$) indicates the presence (resp., absence) of a token at q with counter value i. So, identifying a state as counter-unambiguous enables a massive memory reduction for this state from $O(M)$ to $O(\log M)$.

We design a ***static analysis algorithm*** for checking the counter-ambiguity of NCAs and regexes by performing a systematic exploration of the space of reachable tokens to identify the existence of some input string for which two different tokens are placed on the same control state. This may lead to a large search space (exponential in the size of the regex), and the worst case is not easy to avoid since the problem is NP-hard. To handle difficult instances that involve large repetition bounds, we also provide an *over-approximate* algorithm that gives an inconclusive output for some instances, while still being able to identify cases of

[1] STE stands for State Transition Element [19]. It is a hardware element that roughly corresponds to the state of a homogeneous NFA. It contains a state bit (to indicate whether the state is active or not) and a memory array that represents a character class.

[2] CAMA abbreviates Content Addressable Memory (CAM) enabled Automata accelerator.

counter-unambiguity for most instances from real benchmarks. By combining the exact and over-approximate algorithms, we can statically analyze within milliseconds the vast majority of regexes in the benchmarks Snort [50], Suricata [55], Protomata [42], SpamAssassin [3], and ClamAV [16].

Using the insights about NCA execution mentioned earlier, we propose a ***hardware design*** that is based on existing in-memory NFA architectures (AP, CA, Impala, CAMA) augmented with (1) *counter modules* for counter-unambiguous states, and (2) *bit vector modules* for counter-ambiguous states. We use SPICE [52], an industry-standard simulator for integrated circuits, to perform hardware simulation for the counters and bit vectors and to integrate them into the CAMA architecture. We also provide a ***compiler*** that statically analyzes an input regex to determine counter-(un)ambiguity and then creates a representation of an automaton with counters and bit vectors using the MNRL format [2] that can be used to program the hardware. Several existing architectures like AP provide a counter module in their design, but they typically do not provide a compiler that translates regexes to hardware-recognizable programs. Also, counter registers alone cannot deal with the challenging instances of counting. Compared with prior works that do not provide a bit vector module, this paper proposes a novel design that can systematically handle counting and ensure correct compilation in both the easy (requiring counters) and difficult (requiring bit vectors) cases.

We modified the open-source simulator VASim [61] to simulate the hardware performance of our counter- and bit-vector-augmented CAMA design with implementation in TSMC 28nm process. In microbenchmarks, we evaluated the energy and area consumption of counters and bit vectors against their unfolded counterparts. The results show that our counter- and bit-vector-based design can reduce the energy usage by orders of magnitude and the area by large margins. Furthermore, we evaluated the performance of the augmented CAMA design using the Snort [50], Suricata [55], Protomata [42], and SpamAssassin [3] benchmarks. For applications involving regexes with large counting bounds, the results show as large as 76% energy reduction and 58% area reduction. For regexes with small counting bounds, the results show little to no overhead.

Contributions. The main contributions of this paper are summarized below:

(1) We use the notion of *counter-unambiguity* in order to identify instances of bounded repetition that can be handled with a small amount of memory. We describe both an exact and an over-approximate *static analysis* for counter-(un)ambiguity which, when combined, allow us to efficiently analyze the regexes that arise in several application domains.

(2) We propose a *hardware design* that augments the prior NFA-based CAMA architecture [26] with counter and bit vector modules, which are inspired from the execution of NCAs and the classification of states as counter-(un)ambiguous. This architecture achieves substantial energy and area reductions compared to prior designs.

(3) We provide a *compiler* that enables the high-level programming of the hardware using POSIX-style regexes. The compiler first performs the static analysis for counter-(un)ambiguity and then leverages the analysis results for producing a low-level description of the automaton.

2 Preliminaries

In this section, we will give a brief overview of several well-known concepts, including regular expressions with counting and nondeterministic counter automata (NCAs). We are not interested in NCAs with unbounded counters (which can recognize non-regular languages), so we focus on NCAs with bounded counters. These automata are an appropriate model for implementing regular expressions with counting. Differently from most definitions of NCAs in the literature, we allow each control state of the automaton to have a different number of counters. This flexibility allows us to carefully bound the memory needed for NCA execution.

Let Σ be a finite alphabet. A ***regular expression*** (or *regex*) over Σ is given by the grammar $r ::= \varepsilon \mid \sigma \mid r \cdot r \mid r + r \mid r^* \mid r\{m, n\}$, where $\sigma \subseteq \Sigma$ is a predicate over the alphabet and m, n are natural numbers. The expression $r\{m, n\}$ describes the repetition of r from m to n times, so we require that $0 \leq m \leq n$. We write $r\{n\}$ for $r\{n, n\}$. The concatenation symbol is sometimes omitted, i.e., we write $r_1 r_2$ instead of $r_1 \cdot r_2$. The *interpretation* of a regex r is a language $[\![r]\!] \subseteq \Sigma^*$, which is defined in the standard way.

Notation for predicates: A predicate over the alphabet is sometimes referred to as a *character class*. The predicate Σ contains all symbols in the alphabet. When we use a symbol $a \in \Sigma$ in a regex, it should be understood as the singleton predicate $\{a\} \subseteq \Sigma$. We will also use the notation $[a_1 \dots a_n]$ in a regex to represent the predicate $\{a_1, \dots, a_n\} \subseteq \Sigma$. We write $[\hat{}a_1 \dots a_n]$ for the predicate $\Sigma \setminus \{a_1, \dots, a_n\}$ that contains all symbols aside from $a_1, \dots, a_n$. For a predicate $\sigma \subseteq \Sigma$, we write $\bar{\sigma} = \Sigma \setminus \sigma$ to denote its *complement*.

We fix an infinite set *CReg* of counter registers or, simply, *counters*. We typically write $x, y, z, \dots$ to denote counter registers. For a subset $V \subseteq CReg$ of counters, we say that a function $\beta : V \to \mathbb{N}$, which assigns a value to each counter in V, is a *V-valuation*.

Definition 2.1. Let Σ be a finite alphabet. A *nondeterministic counter automaton (NCA)* with input alphabet Σ is a tuple $\mathcal{A} = (Q, R, \Delta, I, F)$, where

- Q is a finite set of *states*,
- $R : Q \to \mathcal{P}(CReg)$ is a function that maps each state to a finite set of counters,
- Δ is the *transition relation*, which contains finitely many transitions of the form $(p, \sigma, \varphi, q, \vartheta)$, where p is the source state, $\sigma \subseteq \Sigma$ is a predicate over the alphabet, $\varphi \subseteq (R(p) \to$

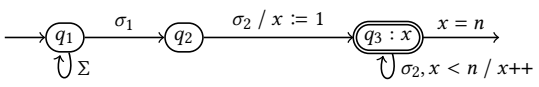
$\mathbb{N}$) is a predicate over $R(p)$-valuations, q is the destination state, and $\vartheta : (R(p) \to \mathbb{N}) \to (R(q) \to \mathbb{N})$,
- I is the *initialization function*, a partial function defined on the subset $\mathrm{dom}(I) \subseteq Q$ of *initial states* that specifies an *initial valuation* $I(q) : R(q) \to \mathbb{N}$ for each initial state q, and
- F is the *finalization function*, a partial function defined on the subset $\mathrm{dom}(F) \subseteq Q$ of *final states* that specifies a predicate $F(q) \subseteq R(q) \to \mathbb{N}$ for each final state q.

We say that a state $q \in Q$ is *pure* if $R(q) = \emptyset$, that is, it has no counter associated with it.

We remark that the states in an NCA of Definition 2.1 do not necessarily have the same counters. In fact, some states may not have any counter at all. In a transition $(p, \sigma, \varphi, q, \vartheta)$, we will call the predicate φ a *guard* because it may restrict a transition based on the values of the counters, and we will call the function ϑ an *action*, because it describes how to assign counter values in the destination state given the counter values in the source state.

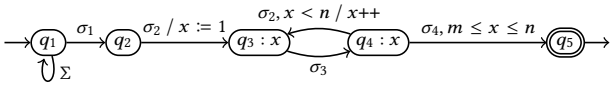
We convert regexes (with counting) to NCAs that recognize the same language using a variant of the Glushkov construction [20, 21]. In contrast to Thompson's construction [57], Glushkov's construction results in ε-free automata that are also *homogeneous*, i.e., all incoming transitions of a state are labeled with the same predicate over the alphabet. We present below several examples of NCAs.

Example 2.2. Consider the regex $r_1 = \Sigma^*\sigma_1\sigma_2\{n\}$ with $n \geq 1$, where σ_1, σ_2 are predicates over the alphabet[3]. The following automaton recognizes the language of r_1:

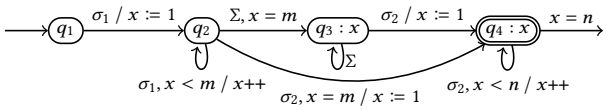


The automaton above has three states: q_1, q_2, and q_3. We write $q_3 : x$ to indicate that $R(q_3) = \{x\}$. Notice that q_1 has no annotation with counters, which means that $R(q_1) = \emptyset$ (i.e., q_1 is pure). We annotate each edge $p \to q$ with an expression of the form $\sigma, \varphi \,/\, \vartheta$, where σ is a predicate over Σ, φ is a guard over the counters of p, and ϑ is an assignment for the counters of q using the counters of p. If the guard φ is omitted, then it is always true. The action ϑ is omitted only when $R(q) \subseteq R(p)$, and the omission indicates that the counters $R(q)$ retain the values from the previous state. We can also indicate this explicitly by writing "$x := x$". We write "$x = n$" for the guard that checks whether the value of counter x is equal to n, and we write "$x := n$" to denote the assignment (action) of the value n to the counter x. We use double circle notation to indicate that a state is final (see state q_3 above). An arrow emanating from a final state q is annotated with the predicate $F(q)$ over counter valuations (recall that F is the finalization function).

The regex $r_2 = \Sigma^*\sigma_1(\sigma_2\sigma_3)\{m, n\}\sigma_4$ with $1 \leq m \leq n$ is recognized by the following automaton:



The regex $r_3 = \sigma_1\{m\}\Sigma^*\sigma_2\{n\}$ with $m, n \geq 1$ is recognized by the following automaton:



All automata so far use one counter. For the regex $r_4 = \Sigma^*\sigma_1(\sigma_2(\sigma_3\sigma_4)\{m, n\}\sigma_5)\{k\}\sigma_6$ with $1 \leq m \leq n$ and $k \geq 1$ we need two counters. See Fig. 1.

Nondeterministic semantics. Let $\mathcal{A}$ be an NCA. A *token* for $\mathcal{A}$ is a pair (q, β), where q is a state and $\beta : R(q) \to \mathbb{N}$ is a counter valuation for q. The set of all tokens for $\mathcal{A}$ is denoted by $\mathbf{Tk}(\mathcal{A})$. For a letter $a \in \Sigma$, we define the *token transition relation* $\to^a$ on $\mathbf{Tk}(\mathcal{A})$ as follows: $(p, \beta) \to^a (q, \gamma)$ if there is a transition $(p, \sigma, \varphi, q, \vartheta) \in \Delta$ with $a \in \sigma$ such that $\beta \in \varphi$ and $\gamma = \vartheta(\beta)$. A token (q, β) is *initial* if the state q is initial. A token (q, β) is *final* if the state q is final and $\beta \in F(q)$. A *run* of $\mathcal{A}$ on a string $a_1 a_2 \ldots a_n \in \Sigma^*$ is a sequence

$$(q_0, \beta_0) \xrightarrow{a_1} (q_1, \beta_1) \xrightarrow{a_2} (q_2, \beta_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} (q_n, \beta_n),$$

where each (q_i, β_i) is a token, q_0 is an initial state and $\beta_0 = I(q_0)$, and $(q_{i-1}, \beta_{i-1}) \to^a (q_i, \beta_i)$ for every $i = 1, \ldots, n$. A run is *accepting* if it ends with a final token. The NCA $\mathcal{A}$ *accepts* a string if there is an accepting run on it. We write $[\![\mathcal{A}]\!] \subseteq \Sigma^*$ for the set of strings that $\mathcal{A}$ accepts.

Notice that, for a NCA $\mathcal{A}$, the set of tokens $\mathbf{Tk}(\mathcal{A})$ together with the transition relations $\to^a$ forms a labeled transition system. The family of transition relations $(\to^a)_{a\in\Sigma}$ can be represented as a ternary relation $\to \subseteq \mathbf{Tk}(\mathcal{A}) \times \Sigma \times \mathbf{Tk}(\mathcal{A})$.

Notation for tokens: For a pure state q (i.e., a state with no counter, see Definition 2.1), there is only one valuation, denoted $0_{\mathbb{N}} : \emptyset \to \mathbb{N}$, which carries no information. So, we will often abuse notation and simply write q for the token $(q, 0_{\mathbb{N}})$. Similarly, for a state q with one counter, i.e., $R(q) = \{x\}$ for some $x \in \mathit{CReg}$, a valuation β (of type $\{x\} \to \mathbb{N}$) for q specifies only one value $c = \beta(x)$ for the unique variable x for q. For this reason, we will sometimes write (q, c) for a token for the state q.

Semantics using configurations. Let $\mathcal{A}$ be an NCA. A *configuration* for $\mathcal{A}$ is a set of tokens for $\mathcal{A}$. We write $\mathbf{C}(\mathcal{A})$ for the set of all configurations for $\mathcal{A}$. Define the configuration transition function $\delta : \mathbf{C}(\mathcal{A}) \times \Sigma \to \mathbf{C}(\mathcal{A})$ as follows:

$$\delta(S, a) = \{(q, \gamma) \mid (p, \beta) \to^a (q, \gamma) \text{ for some } (p, \beta) \in S\}.$$

[3] In order to make the example more concrete, suppose that $\sigma_1 = [ab]$ and $\sigma_2 = [\hat{}a]$. So, the regular expression r_1 is the same as .*[ab][^a]{n} using POSIX notation [38]. Note that Σ^* is the same as .* in POSIX notation.

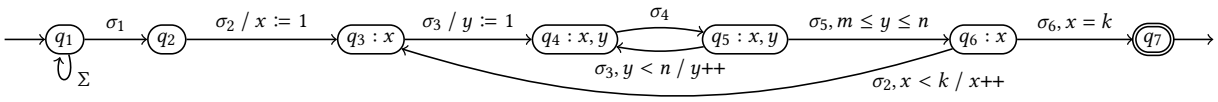


Figure 1. NCA with two counters (x and y) for the regex $\Sigma^*\sigma_1(\sigma_2(\sigma_3\sigma_4)\{m,n\}\sigma_5)\{k\}\sigma_6$ with $1 \le m \le n$ and $k \ge 1$.

We extend the transition function to $\delta : \mathbf{C}(\mathcal{A}) \times \Sigma^* \to \mathbf{C}(\mathcal{A})$ by $\delta(S, \varepsilon) = S$ and $\delta(S, xa) = \delta(\delta(S, x), a)$ for every $x \in \Sigma^*$ and $a \in \Sigma$. Let S_0 be the set of all initial tokens, which we call the *initial configuration*, and define $[\![\mathcal{A}]\!] : \Sigma^* \to \mathbf{C}(\mathcal{A})$ by $[\![\mathcal{A}]\!](x) = \delta(S_0, x)$. This semantics coincides with $[\![\mathcal{A}]\!]$ in the following sense: for every $x \in \Sigma^*$, $x \in [\![\mathcal{A}]\!]$ iff $[\![\mathcal{A}]\!](x)$ contains some final token.

Bounded counters. Let $\mathcal{A}$ be a NCA, and $n \in \mathbb{N}$ be a constant. We say that a token (q, β) is *n-bounded* if $\beta(x) \le n$ for every counter $x \in R(q)$. We also say that $\mathcal{A}$ (resp., a state q) is *n-bounded* if every token (resp., token on state q) reachable from some initial token is n-bounded. Finally, the NCA $\mathcal{A}$ is said to *have bounded counters* if there exists some constant $n \in \mathbb{N}$ such that $\mathcal{A}$ is n-bounded. Notice that NCAs with bounded counters have the same expressiveness as finite-state automata (i.e., DFAs and NFAs), but they are potentially more succinct [53].

As mentioned earlier, the automata that we consider here are obtained from regexes with counting using the Glushkov construction. A consequence of this is that every counter incrementation action of the form x++ is guarded by some test $x < n$ because it corresponds to a subexpression of the form $r\{m, n\}$. It follows that an automaton thus constructed has bounded counters. Moreover, for every control state and every counter, we can read an upper bound from the automaton. For example, in Figure 1, the counter x is bounded above by k (at all states q_3, q_4, q_5, q_6) because $(q_6, \sigma_2, \text{“}x < k\text{”}, q_3, \text{“}x\text{++”})$ is the only transition that increments x. Similarly, the counter y is bounded above by n (at all states q_4, q_5) because $(q_5, \sigma_3, \text{“}y < n\text{”}, q_4, \text{“}y\text{++”})$ is the only transition that increments y.

3 Static Analysis

In this section, we will see how to perform a static analysis over regexes to check counter-(un)ambiguity. It is well-known that the presence of counting in regexes can cause a blow-up in the amount of memory that is needed for the streaming membership problem (checking if a string matches the regex in a single left-to-right pass) [34] (more results about regexes with counting are given in [35, 53]). There are, however, many cases that do not exhibit this worst-case behavior. In this section, we will describe a static analysis for identifying occurrences of bounded repetition $\{m, n\}$ which can be implemented using memory that is logarithmic in n. This enables a significant reduction in the memory that needs to be reserved for the membership problem. In order to identify the easier cases of bounded repetition, we use the concept of counter-unambiguity, which informally says that the nondeterminism of the automaton is constrained. We then develop two algorithms for deciding counter-unambiguity (one exact and one approximate), and we provide experimental results showing that they are effective in practice.

Let $\mathcal{A} = (Q, R, \Delta, I, F)$ be an NCA. For a state $q \in Q$ and a subset $T \subseteq \mathbf{Tk}(\mathcal{A})$ of tokens for the automaton, define $T|_q = T \cap (\{q\} \times (R(q) \to \mathbb{N}))$. That is, $T|_q$ contains exactly those tokens of T whose first component is the state q. The operational intuition is that $[\![\mathcal{A}]\!](x)|_q$ is the set of tokens that we get at state q when we execute the automaton $\mathcal{A}$ on input x. When it is possible to have more than two tokens on the same state q after consuming an input string, we say that the state exhibits *counter-ambiguity*. We will now define this concept and other related notions more formally.

Definition 3.1 (Degree of Counter-Ambiguity). Let $\mathcal{A}$ be an NCA with bounded counters and q be a state. The *(counter-ambiguity) degree* (which we will also call *degree of counter-ambiguity*) of q is defined as

$$\text{degree}(q) = \sup_{x \in \Sigma^*} \big(\text{size of } [\![\mathcal{A}]\!](x)|_q\big).$$

We say that q is *counter-unambiguous* when $\text{degree}(q) \le 1$, and that q is *counter-ambiguous* when $\text{degree}(q) \ge 2$.

Notice that if the degree of a state q is equal to zero, then the state q is unreachable.

3.1 Deciding Counter-Ambiguity

According to Definition 3.1, the degree of counter-ambiguity of a state q is the maximum number of different tokens that can end up at q during a computation. A state q is counter-ambiguous iff there is a string $a_1 a_2 \dots a_n \in \Sigma^*$ and two different runs on $a_1 a_2 \dots a_n$

$$(q_0, \beta_0) \xrightarrow{a_1} (q_1, \beta_1) \xrightarrow{a_2} (q_2, \beta_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} (q_n, \beta_n)$$
$$(q'_0, \beta'_0) \xrightarrow{a_1} (q'_1, \beta'_1) \xrightarrow{a_2} (q'_2, \beta'_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} (q'_n, \beta'_n),$$

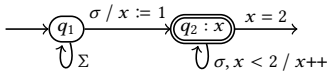
such that $q = q_n = q'_n$ and $\beta_n \ne \beta'_n$.

Let G be the labeled transition system of tokens $\mathbf{Tk}(\mathcal{A})$ and token transitions of the form $t_1 \to^a t_2$, where t_1, t_2 are tokens and $a \in \Sigma$. Define $G^2 = G \times G$ to be the *product* transition system with states $\mathbf{Tk}(\mathcal{A}) \times \mathbf{Tk}(\mathcal{A})$, which contains a transition $\langle t_1, t_2 \rangle \to^a \langle t'_1, t'_2 \rangle$ iff $t_1 \to^a t'_1$ and $t_2 \to^a t'_2$. A pair $\langle t_1, t_2 \rangle$ is initial if both t_1 and t_2 are initial tokens. According to the characterization of the previous paragraph, a state q of $\mathcal{A}$ is counter-ambiguous iff there exists a path in G^2

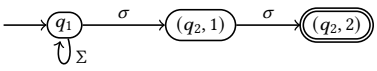
that ends with some pair $\langle (q, \beta), (q, \beta') \rangle$, where $\beta \neq \beta'$. This idea can be extended to characterize the situation where a state q has degree at least $d \geq 2$: there exists a path in the d-fold Cartesian product G^d that ends with some tuple $\langle (q, \beta_1), \ldots, (q, \beta_d) \rangle$, where $\beta_1, \ldots, \beta_d$ are all distinct.

Algorithm for Counter-Ambiguity: When the product transition system G^d is finite, we can decide whether the counter-ambiguity degree of a state is $\geq d$ with a straightforward reachability algorithm. For deciding counter-ambiguity, we check whether the degree is ≥ 2, and therefore it suffices to consider only G^2. Notice that for the bounded counter automata that we consider, G^d is always finite. We just need to exercise care to avoid a blowup in the number of transitions. In our automata, the transitions are annotated with predicates over the alphabet, not symbols of the alphabet. This is a succinct way to represent transitions, and we want to maintain such a representation in the graphs G^d (assuming that we also use such a representation for G). This can be done by considering the intersections of predicates and checking whether they are empty. More specifically, for every pair of transitions $t_1 \to^{\sigma_1} t_1'$ and $t_2 \to^{\sigma_2} t_2'$, we add the transition $\langle t_1, t_2 \rangle \to^{\sigma_1 \cap \sigma_2} \langle t_1', t_2' \rangle$ in G^2 when $\sigma_1 \cap \sigma_2$ is nonempty.

Example 3.2. We will discuss here how to check counter-(un)ambiguity for the regex $\Sigma^* \sigma\{2\}$. First, we construct the NCA for this regex, which is seen below:

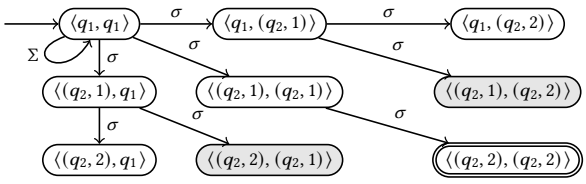


Based on this NCA, we construct the transition system of tokens seen below, where q_1 is abbreviation for the token $(q_1, 0_{\mathbb{N}})$ (q_1 is a pure state), and (q_2, n) is abbreviation for the token $(q_2, x \mapsto n)$ (the counter assignment maps x to n).



The token transition system is essentially an NFA, where the final state (token) is indicated with a double circle.

To check the counter-ambiguity of a state q, we build the product transition system and check whether there exists a path that ends in a pair of tokens $\langle (q, \beta), (q, \beta') \rangle$ with $\beta \neq \beta'$. The figure below shows the product transition system where the presence of the pair $\langle (q_2, 1), (q_2, 2) \rangle$ or $\langle (q_2, 2), (q_2, 1) \rangle$ (colored in gray) witnesses the counter-ambiguity.



Because of symmetry, some states and transitions can be safely removed from the product automaton. Notice, for example, that we do not need to explore both $\langle (q_2, 1), q_1 \rangle$ and $\langle q_1, (q_2, 1) \rangle$. Therefore, in future examples, we will omit part of the product automaton.

The exact analysis halts as soon as it finds a token pair that witnesses counter-ambiguity. So, not all pairs are generated during the static analysis, unless the regex is counter-unambiguous.

Consider a regex r that contains an occurrence of counting of the form $(abcd)\{m, n\}$. When the repetition bounds are sufficiently large, in the automaton $\mathcal{A}$ for r, the four states that correspond to $abcd$ are either all counter-unambiguous or they are all counter-ambiguous. For this reason, the notion of counter-(un)ambiguity can be defined with respect to instances of bounded repetition in regexes. We will also call a regex counter-ambiguous if it contains at least one occurrence of bounded repetition that is counter-ambiguous (equivalently, the NCA for the expression has at least one counter-ambiguous state).

Lemma 3.3 (Checking Counter-Ambiguity Is Hard). Let CAmbiguity be the following problem: Given a regex r as input, is r counter-ambiguous? CAmbiguity is NP-hard.

Proof. Consider the alphabet $\Sigma = \{a, b, \#\}$. We will give a polynomial-time reduction from the subset sum problem to CAmbiguity. Let $S = \{n_1, n_2, \ldots n_m\}$ be a set of natural numbers and T be a natural number. Recall that the subset sum problem asks whether there is a subset $S' \subseteq S$ of numbers whose sum is equal to T. Consider the regex

$$(((a\{n_1\} + \varepsilon) \cdots (a\{n_m\} + \varepsilon)\#b) + (a\{T\}\#bb))b\{2\}.$$

We focus on the rightmost occurrence of bounded repetition (i.e., $b\{2\}$). We claim that this occurrence is counter-ambiguous if and only if there is a subset $S' \subseteq S$ whose sum is T. Consider the corresponding Glushkov automaton and the state q which leads to the final state at the end that recognizes the $b\{2\}$. A word witnessing a path to q would have to be of the form $a^x\#b^y$ for some natural numbers x, y. If $x \neq T$, then the word has no path through the branch $(a\{T\}\#bb)$. So, the only value it can induce on the counter at the end is $(y - 2)$. If $x = T$, and there exists a subset S' of S such that $\sum S' = T$, then $a\{T\}\#bbb$ could either take the path $(a\{T\}\#bb)$ and set the counter to 1, or it could take the other path and set the counter to 2. If $x = T$ and there is no such subset S', then the only path the word can take is through the branch $(a\{T\}\#bb)$ which would set the counter to $(y - 2)$. □

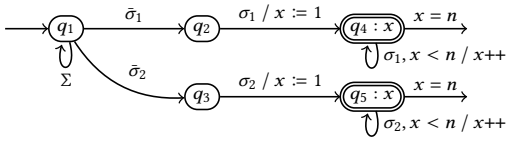
3.2 Over-Approximate Analysis

In §3.1, we presented an (exact) algorithm for deciding the counter-(un)ambiguity of regexes and NCAs. The algorithm operates on the transition system of tokens of an NCA, whose size can be exponential in the size of the regex, because of the counter valuations. For example, the regex $\Sigma^* \cdot a \cdot \Sigma\{n\}$ has size $\Theta(\log n)$ (because the repetition bound n is represented succinctly in binary or decimal notation) and the

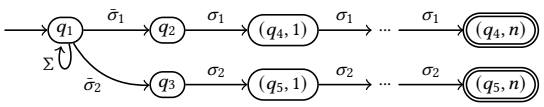
corresponding token transition system has size $\Theta(n)$. From this it follows that the exact algorithm may need exponential time in the worst case. Unfortunately, this worst-case behavior is not easy to avoid given the NP-hardness of the problem (Lemma 3.3). For this reason, we propose here a heuristic algorithm that performs an "over-approximate" analysis, which can give two outputs: it either declares that a state is counter-unambiguous, or it says that the analysis is inconclusive. In other words, there are cases where the algorithm may suspect that a state is counter-ambiguous, but it cannot conclusively declare it so.

The idea is to over-approximate all occurrences of $\{m, n\}$ (constrained repetition) with * (unconstrained repetition), except for the one that we are analyzing. If we think of this transformation in terms of NCAs, we see that it adds more paths to the token transition graph, because more transitions are now enabled. A consequence of this is that if the over-approximate automaton is counter-unambiguous, then surely the original automaton (which has less paths) is also counter-unambiguous. On the other hand, if the over-approximate automaton is counter-ambiguous, then we cannot infer that the original automaton is counter-ambiguous.

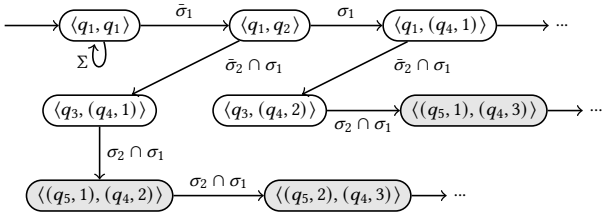
Example 3.4. We show the static analysis for a counter-unambiguous regex $r = \Sigma^*(\bar{\sigma}_1\sigma_1\{n\} + \bar{\sigma}_2\sigma_2\{n\})$, where n is a constant. For this regex, the over-approximate analysis is more efficient than the exact analysis. To illustrate this, we first construct the NCA:



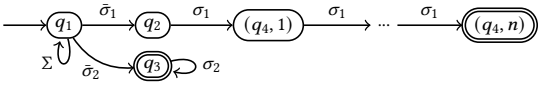
The exact analysis constructs the token transition system:



To determine whether the regex is counter-unambiguous, the exact analysis explores all possible token pairs in the product transition system. In this example, the number of explored pairs is $\Theta(n^2)$. Below is a part of the product transition system, in which all token pairs $\langle (q_5, i), (q_4, j)\rangle$ with $1 \le i < j \le n$ (colored in gray) will be explored.



We have observed that regexes of the form $r = \Sigma^*(\bar{\sigma}_1\sigma_1\{n\} + \bar{\sigma}_2\sigma_2\{n\})$, where n is a large number, can be found in the Snort and Suricata benchmarks. For these regexes, the exact analysis may require a long computation. Fortunately, the over-approximate analysis is substantially faster. We approximate the regex as $r' = \Sigma^*(\bar{\sigma}_1\sigma_1\{n\} + \bar{\sigma}_2\sigma_2^*)$ and $r'' = \Sigma^*(\bar{\sigma}_1\sigma_1^* + \bar{\sigma}_2\sigma_2\{n\})$ and check the counter-ambiguity of r' and r'' using the exact analysis. The regex r is determined to be counter-unambiguous if both r' and r'' are counter-unambiguous. Below, we construct the token transition system G for r'. Only $\Theta(n)$ token pairs are explored in the product transition system G^2.



The over-approximate analysis checks the counter-ambiguity of r', r''. So, it reduces the complexity from $\Theta(n^2)$ to $\Theta(n)$.

3.2.1 NCA Execution with Bit Vectors. If the static analysis determines that an NCA state q is counter-ambiguous, then this implies that the execution of the automaton may require several memory locations to store tokens of the form (q, β). Assuming that q has only one counter register x (i.e., $R(q) = \{x\}$) and that q is n-bounded, we know that there are at most n different possible tokens. In order to compactly represent a set of tokens, the idea is to use a bit vector that indicates the presence or the absence of a specific token on q. So, a bit vector v encodes a set of tokens on q as follows: $v[i] = 1$ iff the token (q, i) is active. We can also think of a bit vector as a representation for part of the automaton configuration (recall the configuration semantics from §2).

It remains to see how the execution of the automaton can be described using these bit vectors to represent the configuration. Example 2.2 shows the NCA for the regex $\Sigma^*\sigma_1(\sigma_2\sigma_3)\{m, n\}\sigma_4$. This NCA is general enough to illustrate the main ways in which we manipulate bit vectors:

(1) Consider a transition $p \to q$, annotated with "σ / $x := c$", where p is pure and $R(q) = \{x\}$. A token on p is transformed into a bit vector v for q that is everywhere 0 except that $v[c] = 1$.
(2) Let $p \to q$ be a transition, annotated with σ, where $R(p) = R(q) = \{x\}$. Since the transition does not change the counter valuations, a bit vector v on p is passed along unchanged to q.
(3) We will deal now with a transition $p \to q$, annotated with "$\sigma, x < n$ / $x{+}{+}$", where $R(p) = R(q) = \{x\}$. Assume further that both p and q are n-bounded, which means that each state carries a bit vector of size n. This transition corresponds to performing a *shift operation* to the bit vector v of p, resulting in a new bit vector v' for q. We have: $v'[1] = 0$ and $v'[i{+}1] = v[i]$ for ever $i = 2, \ldots, n-1$.
(4) Finally, let us consider a transition $p \to q$, annotated with "$\sigma, m \le x \le n$", where $R(p) = \{x\}$ and q is pure. If v is the current bit vector for p, then taking this transition produces a token for q if and only if one of $v[m], v[m+$

Table 1. Analysis of regexes in the benchmarks.

Benchmark	# total	# supported	# counting	# c-ambiguous
Protomata	2338	2338	1675	1675
Snort	5839	5315	1934	282
Suricata	4480	3728	1510	246
SpamAssassin	3786	3690	459	279
ClamAV	100472	100472	4823	3626

$1], \ldots, v[n-1], v[n]$ is equal to 1. In other words, we have to compute the disjunction $v[m] \vee \cdots \vee v[n]$.

The above cases involve the main operations that we use for bit vectors: setting the least significant bit (case 1), shifting left by one position (case 3), and computing the disjunction of some of the most significant bits (case 4).

The way bit vectors are used (setting the lowest-order bit, shifting, and reading high-order bits) is similar to how queues and sliding windows are used for runtime verification with metric temporal logic (MTL) [7, 15, 32, 33]. We note that MTL involves constructs that specify time durations with intervals of the form $[m, n]$, which are akin to the bounded repetition operators $\{m, n\}$ of regexes. This explains the similarity in the implementation.

3.3 Implementation and Experiments

We have implemented a Java program that statically analyzes regexes to determine if they are counter-(un)ambiguous. We will call this program the *counter-ambiguity checker*. The implementation includes both the exact and the over-approximate analyses. As the approximate analysis may be unable to verify the counter-ambiguity of some instances, our checker implements a ***hybrid analysis***. First, it checks the counter-(un)ambiguity of each instance of bounded repetition in the regex using the over-approximate analysis. If it finds a potentially counter-ambiguous instance, then it halts the over-approximate analysis and uses the exact algorithm to check the regex. Otherwise, it determines that the regex is counter-unambiguous.

The checker not only determines if a regex is counter-ambiguous but also provides a *counter-ambiguity witness*, which is a string over the alphabet. If the NCA is executed on the witness, then at least two tokens with different counter valuations will end up on some state of the NCA. The checker supports the analysis of counter-ambiguity for each instance of bounded repetition inside a regex. For example, given a regex $\sigma_1\{m\}\Sigma^*\sigma_2\{n\}$, it can check the first instance (i.e., $\{m\}$), which is counter-unambiguous, and the second instance (i.e., $\{n\}$), which is counter-ambiguous.

We evaluate the performance of our counter-ambiguity checker using five benchmarks, which contain regexes collected from real applications. These benchmarks are: (1) the **Snort** [50] and (2) **Suricata** benchmarks [55] that contain patterns for network traffic, (3) the **Protomata** benchmark that includes 1309 protein motifs from the PROSITE database [39, 42], (4) the **ClamAV** benchmark [16] that contains patterns that indicate the presence of viruses, and (5) the **SpamAssassin** benchmark [3] that includes patterns for detecting spam email.

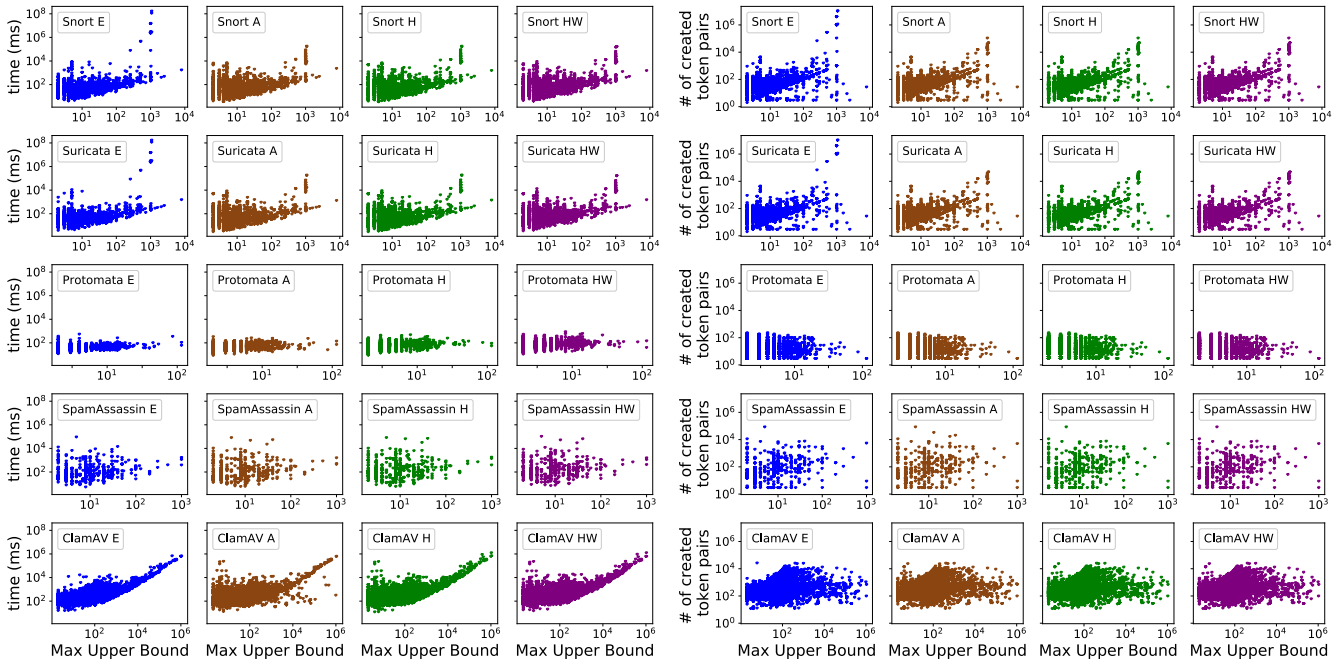
Table 1 shows some statistics for the regexes included in the benchmarks. In the Snort, Suricata, and SpamAssassin benchmarks, some of the collected regexes may contain backreferences [38], which is not a regular operator (i.e., it can give rise to non-regular languages). We filter out regexes with backreferences from the datasets and perform the static analysis on the remaining regexes (which contain the supported regular operators). Table 1 provides the following information: the total number of regexes for each benchmark, the number of regexes with supported (regular) operators, the number of regexes with at least one occurrence of constrained repetition (counting), and the number of counter-ambiguous regexes.

Experimental setup. The experiments were executed in Ubuntu 20.04 on a desktop computer equipped with an Intel Xeon(R) E3-1241 v3 CPU (4 cores) with 16 GB of memory (DDR3 at 1600 MHz). We used OpenJDK 17 and set the maximum heap size to 4 GB. For each regex, we executed 20 trials and selected the mean runtime as the value used the reported results (excluding the first 10 "warm-up" trials).

Performance: Running Time. We evaluate the performance of the static analysis over regexes that have non-nested instances of constrained repetition. We report the running time of the static analysis and we consider its dependence on the following "measure of complexity" for a regex r: the maximum repetition upper bound over all occurrences of $\{m, n\}$ in a regex, which we denote by $\mu(r)$. For example, the regex $r = \sigma_1\{1, 5\}\sigma_2\sigma_3\{4\}$ has two occurrences of constrained repetition, and the maximum repetition upper bound is $\mu(r) = \max(5, 4) = 5$. In general, we expect the running time for the analysis of a regex r to depend on $\mu(r)$, since checking counter-ambiguity involves the generation of token pairs whose number increases as $\mu(r)$ increases.

Figure 2(a) shows the running time of the static analysis indexed by the measure μ. The results are shown in 20 plots, which are organized in a 5×4 grid. There are 5 rows, one for each benchmark: Snort, Suricata, Protomata, SpamAssassin, ClamAV. There are 4 columns, one for each variant of the static analyzer: exact, approximate, hybrid, and hybrid with witness reporting. Each of these 20 plots contains multiple points, one for each regex of the benchmark. For every regex r, the corresponding point has horizontal coordinate equal to $\mu(r)$ and vertical coordinate equal to the running time of the analysis (in milliseconds). We observe that the running time for analyzing a regex r generally increases as $\mu(r)$ increases.

In the Snort and Suricata benchmarks, the checker takes more than 100 seconds to perform the exact analysis for



(a) running time **(b)** # of created token pairs

Figure 2. The (a) running time and the (b) # of created token pairs of static analysis for regexes with different maximum upper bounds of repetitions. E means exact analysis, A means approximate analysis, H means hybrid analysis, HW means hybrid analysis with reporting inputs that witness the ambiguity. E.g., "Snort E" means the exact analysis in Snort benchmark.

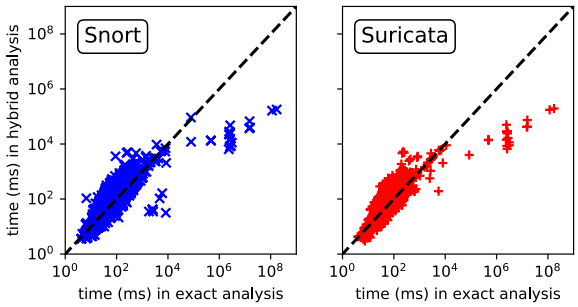


Figure 3. Running time (ms) comparison of exact and hybrid analyses on the Snort and Suricata benchmarks.

several counter-unambiguous regexes. See the top-right outliers in the plots labeled "Snort E" and "Suricata E" in Figure 2(a). This information is seen more promimently in Figure 3, where the exact and hybrid analyses are compared on the Snort and Suricata benchmarks. The points with horizontal coordinate $>10^5$ (msec) are noteworthy. They are substantially below the diagonal, which means that the hybrid analysis offers significant improvement in terms of running time. Some of these regexes are of the form $\Sigma^*(\bar{\sigma}_1\sigma_1\{m\} + \bar{\sigma}_2\sigma_2\{n\} + \cdots)$, where $m, n, \ldots$ are large numbers. When performing exact analysis on these regexes, the checker needs to explore a large number of token pairs, which makes the analysis time-consuming. However, as discussed in Example 3.4, the over-approximate analysis can greatly reduce the cost of the computation. We observe that the over-approximate analysis reduces the running time of expensive regexes by over 100 times in both the Snort and Suricata benchmarks. Moreover, as these regexes are counter-unambiguous, the result of their over-approximate analysis is accurate. This explains why the hybrid analysis also reduces the running time of these challenging regexes.

The fourth column in Figure 2(a) shows the performance (in terms of running time) of a variant of the static analyzer that reports a witness (input string) when a regex is counter-ambiguous. We observe that finding and reporting a counter-ambiguity witness add a very small overhead to the static analysis. This is because recording the witness amounts to simply storing a transition symbol whenever the analysis moves from one token pair to another.

Performance: Memory Footprint. The checker analyzes the counter-ambiguity of a regex by exploring token pairs in a product transition system. These token pairs are created on the fly, as the transition system is being explored. We estimate the memory footprint of the static analysis by measuring the number of token pairs that the checker creates. Figure 2(b) shows the results for five benchmarks and

four different variants of the static analysis. Similarly to the case of running time, the over-approximate analysis greatly reduces the worst-case cost of analyzing several counter-unambiguous regexes in the Snort and Suricata benchmarks.

4 Hardware Implementation and Experiments

In this section, we present our hardware design for efficiently executing NCAs. We augment a state-of-the-art in-memory NFA acceleration architecture called CAMA [26] with counter and bit vector modules. We report hardware simulation results in both microbenchmarks and application benchmarks.

4.1 Hardware Design

Existing in-memory automata accelerators adopt a two-phase architecture: a state matching phase that finds the current active states, and a state transition phase that calculates the available states in the next cycle. AP-style accelerators, such as AP [19], CA [54], and eAP[4] [47], perform state matching by reading from read-access memories (RAMs) that store bit vector representations of states in memory columns. Each column in the RAM represents one state, which is called a State Transition Element (STE). Using 8-bit symbols as an example, each RAM entry is 256-bit and the i-th position has value 1 iff the symbol i is associated with the state[5]. Additionally, the connections between states are programmed into a switch network where existing state transitions are realized as physical connections.

Each processing cycle begins in the state matching phase, where an input symbol is encoded as a one-hot representation[6] and used as the address to read from the state matching memory. The columns that read out '1's indicate successful matches between the input symbol and the STEs. With a logical AND operation between the available states reported from the last cycle and the matched states reported by the memory in the current cycle, matching results of the active states in the current cycle are determined. Next, in the state transition phase, the current active states pass through the programmed switch network to create the next vector which stores available states for the next cycle.

However, AP-style accelerators severely under-utilize the state matching memories in realistic NFAs across common benchmarks, because this approach is optimal only for the worst case of purely random NFAs. Impala [46] and CAMA [26] made critical improvements by proposing special encoding schemes to reduce the state matching memory requirements. CAMA further employs specialized content-addressable memories (CAM) to perform state matching with lower energy and memory footprints than all other designs using RAM. As a result, the memory requirement for 256 STEs is reduced from one 256×256 6-transistor SRAM in AP and CA, to two 16×256 6-transistor SRAMs in Impala and approximately one 16×256 8-transistor CAM in CAMA. Moreover, CAMA optimizes a reduced-crossbar switch network that was first proposed by eAP, which largely reduces the area and energy costs of state transitions. Compared with prior NFA in-memory architectures, CAMA achieves leading throughput, energy, and area efficiency. CAMA's throughput is 2.14GBps, 1.18x better than CA, 9.5x better than FPGA-based Grapefruit [40], and 2-4 orders better than CPU/GPU solutions. CAMA's energy efficiency is 4.91nJ/Byte, over 10x better than most efficient alternatives, i.e. Grapefruit (FPGA) and AP. This paper uses the latest memory- and energy-efficient CAMA architecture as the baseline and augments it with our proposed counter and bit vector modules.

Figure 4(a) shows the Glushkov NCA for the counter-unambiguous regex $a(bc)\{1,3\}c$. The Glushkov construction ensures that the NCA is homogeneous (all transitions entering a state are labeled with the same predicate over the alphabet). This property allows us to convert the NCA to a hardware-friendly representation by omitting the initial state and pushing the predicates from the edges to the states, thus transforming NCA states into STEs. For example, we push the predicate a into state q_a so that in Figure 4(b) we have a state labeled with the predicate a, which becomes an STE that is activated to fire signals only when the input satisfies the predicate a. The original CAMA design, as shown in Figure 4(c), only supports NCAs by fully unfolding bounded repetitions. In our augmented CAMA, two types of hardware modules, counters and bit vectors, are added to accelerate the execution of NCAs. As shown in Figure 4(d), both modules take input from STEs related to counting and produce output signals to the switch network. Counters are inserted to support counter-unambiguous repetitions, while bit vectors are reserved for counter-ambiguous repetitions (recall §3.2.1). Compared to CAMA, the additional counters and bit vectors retain all necessary processing information while avoiding the cost of unfolding (which results in additional STEs). In Section 4.2, we will further explain the design and the input/output ports of the counter and bit vector modules.

Figure 5 shows the structure of an augmented CAMA bank. The overall architecture of CAMA is preserved, and the functionalities of existing components remain the same. Each bank consists of an input/output buffer and 16 processing arrays. Each array has a global switch and 8 processing elements (PEs). Each PE contains two 256-STE CAM arrays, two local switches, and 8 counters, and it may contain a bit vector depending on the configuration from users. Note that

[4] eAP stands for embedded Automata Processor.

[5] Recall from §2 that we consider homogeneous automata, which means that all transitions leading to a state q are labeled with the same predicate σ over the alphabet. The RAM entry is a representation of the predicate σ.

[6] The one-hot representation of an 8-bit symbol i consists of $2^8 = 256$ bits, where the i-th bit has value 1 and the others are 0.

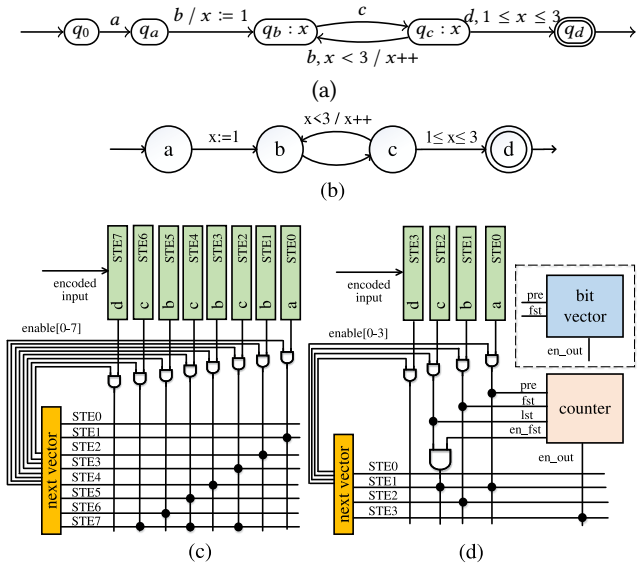


Figure 4. (a) Glushkov NCA for regex $a(bc)\{1,3\}d$. (b) Corresponding NCA with STEs. (c) Original hardware using unfolding. (d) Augmented hardware with counter or bit vector.

the input ports to the counter and bit vector modules are connected to fixed groups of STEs. For example, as shown on the right, port `pre` is connected to STEs 0 to 7, port `fst` is connected to STEs 8 to 16, and so on. When enabled, an STE within the group can pass signals to the connected port. We use an efficient mapping algorithm to build the connection between ports and STE groups so that we maintain the generality of the design but reduce the complexity of routing.

It is worth mentioning that our proposed counters and bit vectors are not only suitable for the CAMA architecture. Other in-memory automata architectures, like CA, can also be augmented for NCAs with minor hardware design changes. Specifically, these changes are: (1) counters and bit vectors need to be allowed to connect to elements that represent states, and (2) the routing network needs to be extended to store the transitions from counters and bit vectors.

Software-Hardware Codesign. The initial motivation for our hardware design came from the observation that several instances of bounded repetition require significantly less memory than what is suggested by a naive unfolding. This led to the formalization of counter-(un)ambiguity in NCAs and the corresponding static analysis. For the counter-unambiguous case, it suffices to use simple counter modules that keep track of the number of repetitions. For the counter-ambiguous case, the use of bit vectors is a very natural choice for a hardware representation of sets of tokens. These considerations led to the design of the counter and bit vector modules. Physical constraints imposed by the hardware call for minimizing the connections between STEs and the counting

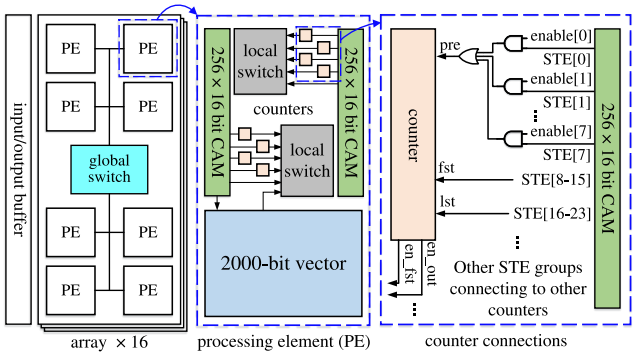


Figure 5. Abstraction of proposed augmented CAMA bank, where PE is abbreviation for Processing Element.

modules. For this reason, we have chosen to use bit vectors for counter-ambiguous repetitions of the form $\sigma\{m,n\}$ and use (partial) unfolding for other cases. The vast majority of counter-ambiguous repetitions in real-world benchmarks are of this form, so this approach offers efficiency (due to an optimized hardware implementation) without sacrificing generality (since the remaining cases can be handled at the level of the software/compiler).

4.2 Compilation from Regex to MNRL

To program the hardware, we provide a description of the automata in the MNRL language [2]. Our compiler takes a source regex and produces the MNRL file with the following steps: (1) First, the compiler parses the regex and simplifies it with certain rewrite rules, including the unfolding of repetitions with upper bound < 2 and the merging of character classes inside simple alternations (e.g., `[a]|[b]` is rewritten to `[ab]`). (2) Then, the compiler performs the static analysis of §3 and annotates the regex with the counter-(un)ambiguity result for each occurrence of repetition. (3) Finally, the compiler generates the MNRL file using these annotations, distinguishing cases where a counter suffices (counter-unambiguous) from cases where a bit vector is necessary (counter-ambiguous).

MNRL provides an element called `upCounter` for representing simple counters [2, 19]. However, there is no distinction between counter-ambiguous and counter-unambiguous repetition. We have therefore extended the MNRL format by adding syntax for counters and bit vectors.

Figure 6 presents an abstraction of the ***counter module*** (enclosed by a dashed line) by showing how it is used to implement the counter-unambiguous regex $a(bc)\{m,n\}d$ in hardware. A counter has three incoming ports `pre`, `fst`, and `lst`, and two outgoing ports `en_fst` and `en_out`, where ports are labeled with red dots in Figure 6. The input port `pre` (i.e., pre-counting) is connected to the STE (labeled with a) located right before the repetition, `fst` (i.e., first) is connected to the first STE (labeled with b) in the repetition, and

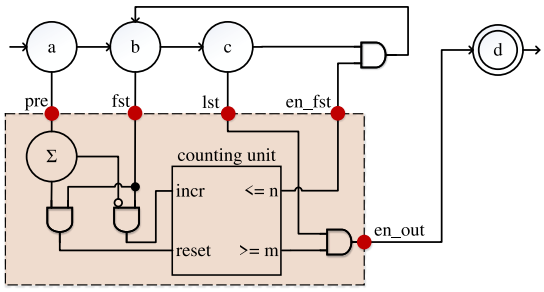


Figure 6. Use of counter module to implement $a(bc)\{m,n\}d$.

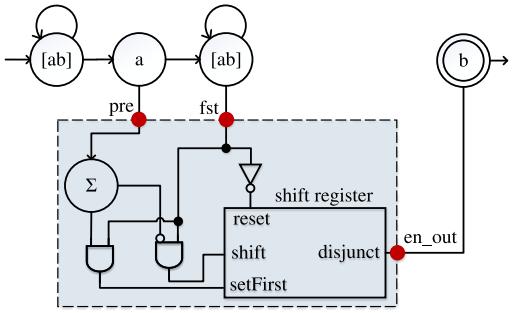


Figure 7. Use of bit vector to implement $[ab]^*a[ab]\{m,n\}b$.

`lst` (i.e., last) is linked to the last STE (labeled with c) in the repetition. The output port `en_out` (i.e., enable output STE) activates the STE (labeled with d) located right after the repetition, and `en_fst` (i.e., enable first STE) activates the first STE (labeled with b) in the repetition. The counter module consists of a synchronous counting unit using D flip-flop and two digital comparators. The module is designed to meet four constraints: (1) The counter value is reset to 0 when `pre` was active in the previous cycle and `fst` is currently active. This corresponds to the initialization of the repetition. (2) The counter value is incremented by 1 when `fst` is active but `pre` was not active in the previous cycle. This corresponds to one complete cycle. (3) `en_out` fires if `lst` is active and the counter value is within the expected range (i.e., $[m,n]$). (4) `en_fst` fires if `lst` is active and the counter value is $\leq n$.

Figure 7 presents an abstraction of the ***bit vector module*** by showing how the regex $[ab]^*a[ab]\{m,n\}b$ is implemented in hardware. The core component of the bit vector is a serial-in-parallel-out shift register. It supports four primary operations: (1) `reset`, which resets all bits in the vector to 0, (2) `setFirst`, which sets the first bit of the vector to 1, (3) `shift`, which shifts the vector by one bit, and (4) `disjunct`, which computes the disjunction of a sub-array of bits from index m to n (if one of the bits in the sub-array is 1, the output signal fires).

Table 2. Hardware component parameters

Component	Energy (fJ)	Delay (ps)	Area (μm^2)
CAMA Bank	16780	325	3919
17-bit counter	288	101	237
2000-bit vector	3340	71	6382

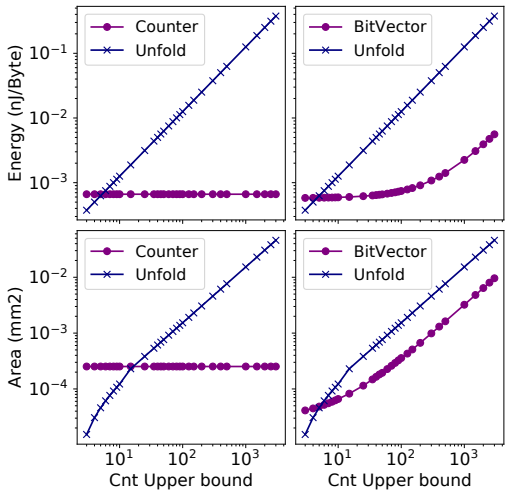


Figure 8. Energy (upper two figures) and area (bottom two) trade-off of unfolding vs using counter (left two figures) and bit vector (right two), where axis is log-scaled.

4.3 Hardware Evaluation

We modified the open-source simulator VASim [61] to simulate the hardware performance of our augmented CAMA. We include 17-bit counters for supporting unambiguous counting, and 2000-bit vectors for supporting ambiguous counting, where the bit vector can be broken down to segments and used separately for counting with small upper bounds. We use a TSMC 28nm CMOS technology and the industry-standard SPICE circuit simulator [52] to obtain the energy, delay, and area parameters of each component (Table 2). Since state transition is the critical path in CAMA, state matching and counter/bit-vector operations can be performed within a single clock cycle in the augmented CAMA, maintaining the same clock frequency of 2.14 GHz and throughput as CAMA-T (CAMA version optimized for high throughput) without performance penalties.

Micro-benchmarks. Figure 8 shows the trade-off of unfolding vs. using counter and bit vector modules. In the left two sub-figures, we consider regexes $a\{n\}$ with different values of n. These regexes are counter-unambiguous – the hardware implementation only needs a single counter module to perform the matching, while unfolding creates n STEs. The upper-left (resp., bottom-left) sub-figure shows the energy (resp., area) cost of using a counter module compared with unfolding, where we always use a 17-bit counter module to represent counter values regardless of their different

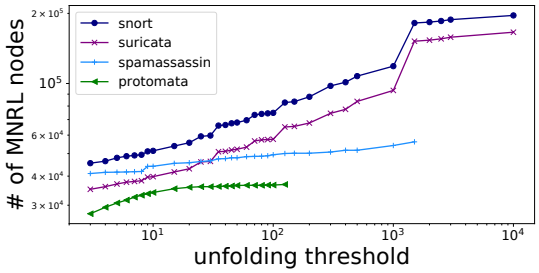


Figure 9. Total number of MNRL nodes with different unfolding thresholds (both axes are log-scaled).

repetition bounds. In the right two sub-figures, we consider regexes $\Sigma^* a\{n\}$. These regexes are counter-ambiguous, so the hardware needs to use a bit vector to perform matching, while unfolding creates n STEs. In this comparison, we set the length of the bit vector to be equal to n for each data point (this implies that bits are wasted). The upper-right (resp., bottom-right) sub-figure shows the energy (resp., area) cost of using a bit vector compared with unfolding. From the results shown in Figure 8, we observe that using a counter/bit vector provides better performance compared to unfolding even for repetitions with small upper bounds. It consistently reduces energy usage by orders of magnitude and areas by large margins.

Application benchmarks. We use the same benchmarks as described in Section 3.3 (except for ClamAV). Figure 9 shows the number of MNRL nodes (which is linear in the number of STEs) for different unfolding thresholds. For each benchmark and each point in the corresponding curve, the x coordinate is an unfolding threshold k and the y coordinate is the number of MNRL nodes that are obtained from compiling the entire benchmark after bounded repetitions up to k have been unfolded. The rightmost point on each benchmark curve shows the unfolding threshold that results in full unfolding for all regexes of the benchmark and the resulting number of MNRL nodes.

We have simulated the area and the energy consumption of our augmented CAMA by feeding compiled MNRL files with different unfolding thresholds to the modified VASim. Figure 10 shows the per-input-byte energy consumption and the total area cost of the augmented CAMA. The results show up to 76% energy reduction and 58% area reduction in benchmarks with an abundance of instances of bounded repetition with large upper bounds (i.e., Snort and Suricata). In benchmarks that generally include bounded repetitions with small upper bounds (i.e., Protomata and SpamAssassin), the augmented CAMA hardware still outperforms pure CAMA with little to no overhead. We observe that for the Protomata and SpamAssassin benchmarks, our hardware implementation provides less energy and area reduction compared with Snort and Suricata. This is because, in general, the regexes in Protomata and SpamAssassin have small repetition upper bounds. The wasted area in Figure 10 corresponds to unused bits in the bit vector modules.

5 Related Work

There is a rich set of prior works that define ***(un)ambiguity on regular expressions***. Book et al. [10] have defined unambiguous regexes using Glushkov automata [21]. Bruggemann-Klein and Wood have expressed the related notions of *deterministic* [12] and *1-unambiguous* [13] regexes. Hovland [24] has defined the class of *counter-1-unambiguous* for regexes with counting. Hovland et al. [25] have further considered a *strongly 1-unambiguous* class where the membership problem, for regexes with counting and unordered concatenations, can be solved in polynomial time. Gelade et al. [20] have defined *strong* and *weak determinism* and shown that weakly deterministic regexes are exponentially more succinct than the strongly deterministic ones. A survey of unambiguity in automata theory can be found at [17].

Several different automata models and automata-based techniques have been proposed to handle ***the matching of regexes with counting***. DFAs and NFAs have been extended by [23] and [8] respectively by introducing counting operations and guards as an alternative to unfolding for large repetition bounds. An implementation of a class of counter automata, proposed in [59], is based on queues for representing sets of counter values. A variety of software regex matchers, including RE2 [18, 41], Rust's Regex [44], PCRE [37], SRM [45], and Hyperscan [65] support the matching of regexes with counting. These matchers are typically based on the execution of DFAs or NFAs. Matchers like RE2 and SRM unfold constrained repetitions when performing on-the-fly determinization or computing derivatives.

A series of ***ASIC hardware architectures*** [11, 58] have been designed to reach high throughput for network applications relying on pattern matching algorithms. The IBM regX [31] accelerator extends the idea of representing regexes with compressed DFAs [8, 36, 68], which are hybrids between DFAs and NFAs, and its parallelized architecture improves performance on large workloads. Dlugosch et al. [19] designed the Automata Processor (AP), a reconfigurable ASIC hardware based on bit-parallelism [4] that simulates NFAs in parallel. Liu et al. [28] developed SparseAP to provide support for AP to efficiently execute large-scale applications. AP can support many regexes found in real-life applications [61, 62]. However, it provides restricted support for regexes with counting (when upper bounds are larger than 512 they are considered unbounded [43]). Other major ASIC works are based on the Aho-Corasick algorithm [1] including [58], HAWK [56], and HARE [22]. They compute partial matches for all possible alignments and merge them to find a global match. HARE achieves a 32Gbps throughput but has limited support for Kleene operators (which only allow single

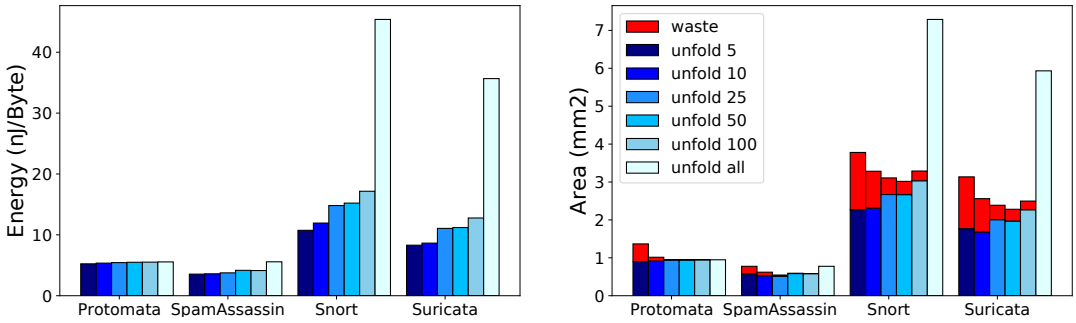


Figure 10. Per-input-byte energy consumption (left) and total area cost (right) of the augmented CAMA hardware

character class repetition), and it provides no support for unbounded counting.

Many prior works [5, 48] focus on ***FPGA and GPU hardware architectures*** to take advantage of their configurability and parallelism. [67] and [51] provide support for regexes with counting on FPGA hardware. [63] extends the DFA ambiguity expressed in [49] to NFA with counters by defining the *character class ambiguity*, a problem that arises when the intersection between two adjacent character class with constraint repetitions (CCR) is non-empty. A min-max algorithm with two counters for every CCR keeps track of all possible matches. Our notion of counter-ambiguity is formulated more generally, and our simulation based on bit vectors handles character class ambiguity. Finally, there are several works that implement regex matching algorithms on GPUs [14, 29, 60, 70].

6 Conclusion

We have investigated hardware acceleration for regular pattern matching, where the patterns are specified by regexes with an extended syntax that involves bounded repetitions of the form $r\{m, n\}$. We have developed a design that integrates counter and bit vector modules into an in-memory NFA-based hardware architecture. This design is inspired from the theoretical model of nondeterministic counter automata (NCAs) and the observation that some instances of bounded repetitions require only a small amount of memory. We formalize this idea using the notion of counter-unambiguity. We have implemented a regex-to-hardware compiler that performs a static analysis for counter-(un)ambiguity over a regex and then creates a representation of an automaton with counters and bit vectors that can be deployed on the hardware. Our experiments show that using counters and bit vectors outperforms unfolding solutions by orders of magnitude. Moreover, in experiments with realistic workloads, we have observed that our design can provide up to 76% energy reduction and 58% area reduction in comparison to CAMA [26], a state-of-the-art in-memory NFA processor.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. This research was supported in part by the US National Science Foundation award CCF 2008096 and the Rice University Faculty Initiative Fund.

References

[1] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (jun 1975), 333–340. https://doi.org/10.1145/360825.360855

[2] Kevin Angstadt, Jack Wadden, Westley Weimer, and Kevin Skadron. 2017. *MNRL and MNCaRT: An Open-Source, Multi-Architecture State Machine Research and Execution Ecosystem.* Technical Report CS2017-01. University of Virginia. https://doi.org/10.18130/V3FN18

[3] Apache SpamAssassin 2022. Apache SpamAssassin: An Open Source Anti-spam Platform. http://spamassassin.apache.org/.

[4] Ricardo A. Baeza-Yates and Gaston H. Gonnet. 1989. Efficient Text Searching of Regular Expressions. In *Automata, Languages and Programming*, Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca (Eds.). Springer, Heidelberg, 46–62. https://doi.org/10.1007/BFb0035751

[5] Zachary K. Baker and Viktor K. Prasanna. 2004. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA '04)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/968280.968312

[6] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-Based Runtime Verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS, Vol. 2937)*. Springer, Heidelberg, 44–57. https://doi.org/10.1007/978-3-540-24622-0_5

[7] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. 2018. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). LNCS, Vol. 10457. Springer, Cham, 135–175. https://doi.org/10.1007/978-3-319-75632-5_5

[8] Michela Becchi and Patrick Crowley. 2008. Extending Finite Automata to Efficiently Match Perl-Compatible Regular Expressions. In *Proceedings of the 2008 ACM CoNEXT Conference (CoNEXT '08)*. ACM, New York, NY, USA, Article 25, 12 pages. https://doi.org/10.1145/1544012.1544037

[9] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 Using Automata Processing Across Different Platforms. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 737–748. https://doi.org/10.1109/HPCA.2018.00068

[10] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. 1971. Ambiguity in Graphs and Expressions. *IEEE Trans. Computers* C-20, 2 (1971), 149–153. https://doi.org/10.1109/T-C.1971.223204

[11] Benjamin C Brodie, David E Taylor, and Ron K Cytron. 2006. A Scalable Architecture for High-throughput Regular-expression Pattern Matching. *ACM SIGARCH computer architecture news* 34, 2 (2006), 191–202.

[12] Anne Brüggemann-Klein and Derick Wood. 1992. Deterministic Regular Languages. In *STACS 92*. Springer, Heidelberg, 173–184.

[13] Anne Brüggemann-Klein and Derick Wood. 1998. One-Unambiguous Regular Languages. *Inf. Comput.* 140, 2 (1998), 229–253. https://doi.org/10.1006/inco.1997.2688

[14] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA Pattern Matching on GPGPU Devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010), 20–26.

[15] Agnishom Chattopadhyay and Konstantinos Mamouras. 2020. A Verified Online Monitor for Metric Temporal Logic with Quantitative Semantics. In *Runtime Verification (RV) (LNCS, Vol. 12399)*, Jyotirmoy Deshmukh and Dejan Ničković (Eds.). Springer, Cham, 383–403. https://doi.org/10.1007/978-3-030-60508-7_21

[16] ClamAV 2022. ClamAV®: An Open-source Antivirus Engine for Detecting Trojans, Viruses, Malware & Other Malicious Threats. https://www.clamav.net/.

[17] Thomas Colcombet. 2015. Unambiguity in Automata Theory. In *Descriptional Complexity of Formal Systems*. Springer, Cham, 3–18. https://doi.org/10.1007/978-3-319-19225-3_1

[18] Russ Cox. 2010. Regular Expression Matching in the Wild. https://swtch.com/~rsc/regexp/regexp3.html.

[19] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098. https://doi.org/10.1109/TPDS.2014.8

[20] Wouter Gelade, Marc Gyssens, and Wim Martens. 2009. Regular Expressions with Counting: Weak versus Strong Determinism. In *Mathematical Foundations of Computer Science 2009*. Springer, Heidelberg, 369–381.

[21] Victor Mikhaylovich Glushkov. 1961. The Abstract Theory of Automata. *Russian Math. Surveys* 16, 5 (1961), 1–53. https://doi.org/10.1070/RM1961v016n05ABEH004112

[22] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware Accelerator for Regular Expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12. https://doi.org/10.1109/MICRO.2016.7783747

[23] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. 2019. Succinct Determinisation of Counting Automata via Sphere Construction. In *Programming Languages and Systems*, Anthony Widjaja Lin (Ed.). Springer, Cham, 468–489. https://doi.org/10.1007/978-3-030-34175-6_24

[24] Dag Hovland. 2009. Regular Expressions with Numerical Constraints and Automata with Counters. In *Theoretical Aspects of Computing - ICTAC 2009*. Springer, Heidelberg, 231–245. https://doi.org/10.1007/978-3-642-03466-4_15

[25] Dag Hovland. 2012. The Membership Problem for Regular Expressions with Unordered Concatenation and Numerical Constraints. In *Language and Automata Theory and Applications*. Springer, Heidelberg, 313–324.

[26] Yi Huang, Zhiyu Chen, Dai Li, and Kaiyuan Yang. 2021. CAMA: Energy and Memory Efficient Automata Processing in Content-Addressable Memories. https://doi.org/10.48550/arXiv.2112.00267 arXiv:2112.00267 [cs.AR]

[27] Marzieh Lenjani and Mahmoud Reza Hashemi. 2014. Tree-based Scheme for Reducing Shared Cache Miss Rate Leveraging Regional, Statistical and Temporal Similarities. *IET Computers & Digital Techniques* 8, 1 (2014), 30–48. https://doi.org/10.1049/iet-cdt.2011.0066

[28] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. 2018. Architectural Support for Efficient Large-Scale Automata Processing. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, New York, NY, USA, 908–920. https://doi.org/10.1109/MICRO.2018.00078

[29] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. 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 (ASPLOS '20)*. ACM, New York, NY, USA, 251–265. https://doi.org/10.1145/3373376.3378471

[30] T. Liu, Y. Yang, Y. Liu, Y. Sun, and Li Guo. 2011. An Efficient Regular Expressions Compression Algorithm from a New Perspective. In *2011 Proceedings IEEE INFOCOM*. IEEE, New York, NY, USA, 2129–2137. https://doi.org/10.1109/INFCOM.2011.5935024

[31] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. 2012. Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, New York, NY, USA, 461–472. https://doi.org/10.1109/MICRO.2012.49

[32] Konstantinos Mamouras, Agnishom Chattopadhyay, and Zhifu Wang. 2021. Algebraic Quantitative Semantics for Efficient Online Temporal Monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 12651)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, Cham, 330–348. https://doi.org/10.1007/978-3-030-72016-2_18

[33] Konstantinos Mamouras and Zhifu Wang. 2020. Online Signal Monitoring with Bounded Lag. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3868–3880. https://doi.org/10.1109/TCAD.2020.3013053

[34] Albert R. Meyer and Michael J. Fischer. 1971. Economy of Description by Automata, Grammars, and Formal Systems. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 188–191. https://doi.org/10.1109/SWAT.1971.11

[35] Albert R. Meyer and Larry J. Stockmeyer. 1972. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *13th Annual Symposium on Switching and Automata Theory (SWAT 1972)*. IEEE Computer Society, Los Alamitos, CA, USA, 125–129. https://doi.org/10.1109/SWAT.1972.29

[36] Hiroki Nakahara, Tsutomu Sasao, and Munehiro Matsuura. 2011. A Regular Expression Matching Circuit Based on a Decomposed Automaton. In *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, Heidelberg, 16–28. https://doi.org/10.1007/978-3-642-19475-7_4

[37] PCRE 2021. PCRE - Perl Compatible Regular Expressions. https://www.pcre.org/.

[38] Posix Syntax in PCRE 2022. Posix Syntax in PCRE. https://www.pcre.org/original/doc/html/pcrepattern.html.

[39] PROSITE 2022. PROSITE: Database of Protein Domains, Families and Functional Sites. https://prosite.expasy.org/.

[40] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. 2020. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, New York, NY, USA, 138–147. https://doi.org/10.1109/FCCM48280.2020.00027

[41] RE2 2021. RE2: Google's regular expression library. https://github.com/google/re2.

[42] Indranil Roy and Srinivas Aluru. 2016. Discovering Motifs in Biological Sequences Using the Micron Automata Processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 13, 1 (2016), 99–111. https://doi.org/10.1109/TCBB.2015.2430313

[43] Indranil Roy, Ankit Srivastava, Matt Grimm, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. 2019. Evaluating High Performance Pattern Matching on the Automata Processor. *IEEE Trans. Comput.* 68, 8 (2019), 1201–1212. https://doi.org/10.1109/TC.2019.2901466

[44] RustRegex 2021. Regex: A Rust Library for Parsing, Compiling, and Executing Regular Expressions. https://github.com/rust-lang/regex.

[45] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS, Vol. 11427)*. Springer, Cham, 372–378. https://doi.org/10.1007/978-3-030-17462-0_24

[46] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron. 2020. Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, New York, NY, USA, 86–98. https://doi.org/10.1109/HPCA47549.2020.00017

[47] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A Scalable and Efficient In-Memory Accelerator for Automata Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 87—-99. https://doi.org/10.1145/3352460.3358324

[48] Reetinder Sidhu and Viktor K Prasanna. 2001. Fast Regular Expression Matching Using FPGAs. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*. IEEE, New York, NY, USA, 227–238.

[49] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. 2008. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/1402958.1402983

[50] Snort 2022. Snort Intrusion Detection System. https://www.snort.org/.

[51] Ioannis Sourdis, Joao Bispo, Joao MP Cardoso, and Stamatis Vassiliadis. 2008. Regular Expression Matching in Reconfigurable Hardware. *Journal of Signal Processing Systems* 51, 1 (2008), 99–121. https://doi.org/10.1007/s11265-007-0131-0

[52] SPICE 2022. SPICE: A General-purpose Circuit Simulation Program for Nonlinear DC, Nonlinear Transient, and Linear AC Analyses. http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE.

[53] Larry J. Stockmeyer and Albert R. Meyer. 1973. Word Problems Requiring Exponential Time (Preliminary Report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC '73)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/800125.804029

[54] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache Automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 259–272. https://doi.org/10.1145/3123939.3123986

[55] Suricata 2022. Suricata Threat Detection Engine. https://suricata.io/.

[56] Prateek Tandon, Faissal M Sleiman, Michael J Cafarella, and Thomas F Wenisch. 2016. Hawk: Hardware Support for Unstructured Log Processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, New York, NY, USA, 469–480. https://doi.org/10.1109/ICDE.2016.7498263

[57] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

[58] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. 2004. Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection. In *IEEE INFOCOM 2004*, Vol. 4. IEEE, New York, NY, USA, 2628–2639 vol.4. https://doi.org/10.1109/INFCOM.2004.1354682

[59] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex Matching with Counting-Set Automata. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 218 (2020), 30 pages. https://doi.org/10.1145/3428286

[60] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. 2009. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Recent Advances in Intrusion Detection*, Engin Kirda, Somesh Jha, and Davide Balzarotti (Eds.). Springer, Heidelberg, 265–283. https://doi.org/10.1007/978-3-642-04342-0_14

[61] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–12. https://doi.org/10.1109/IISWC.2016.7581271

[62] Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, and Kevin Skadron. 2018. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, New York, NY, USA, 13–24. https://doi.org/10.1109/IISWC.2018.8573482

[63] Hao Wang, Shi Pu, Gabriel Knezek, and Jyh-Charn Liu. 2010. A Modular NFA Architecture for Regular Expression Matching. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '10)*. ACM, New York, NY, USA, 209–218. https://doi.org/10.1145/1723112.1723149

[64] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An Overview of Micron's Automata Processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES '16)*. ACM, New York, NY, USA, Article 14, 3 pages. https://doi.org/10.1145/2968456.2976763

[65] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-Pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. USENIX Association, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[66] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. 2017. REAPR: Reconfigurable Engine for Automata Processing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, New York, NY, USA, 1–8. https://doi.org/10.23919/FPL.2017.8056759

[67] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. 2008. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '08)*. ACM, New York, NY, USA, 30–39. https://doi.org/10.1145/1477942.1477948

[68] Yi-Hua E. Yang and Viktor K. Prasanna. 2011. Space-time Tradeoff in Regular Expression Matching with Semi-deterministic Finite Automata. In *2011 Proceedings IEEE INFOCOM*. IEEE, New York, NY, USA, 1853–1861. https://doi.org/10.1109/INFCOM.2011.5934986

[69] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. 2006. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '06)*. ACM, New York, NY, USA, 93–102. https://doi.org/10.1145/1185347.1185360

[70] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. 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 (PPoPP '12)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/2145816.2145833