BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions with Bounded Repetitions

Ziyuan Wen*
zw75@rice.edu
Rice University
Houston, Texas, USA

Lingkun Kong*
klk@rice.edu
Rice University
Houston, Texas, USA

Alexis Le Glaunec
afl5@rice.edu
Rice University
Houston, Texas, USA

Konstantinos Mamouras
mamouras@rice.edu
Rice University
Houston, Texas, USA

Kaiyuan Yang
kyang@rice.edu
Rice University
Houston, Texas, USA

Abstract

Regular pattern matching is pervasive in applications such as text processing, malware detection, network security, and bioinformatics. Recent studies have demonstrated specialized in-memory automata processors with superior energy and memory efficiencies than existing computing platforms. Yet, they lack efficient support for the construct of bounded repetition that is widely used in regular expressions (regexes). This paper presents BVAP, a software-hardware co-designed in-memory Bit Vector Automata Processor. It is enabled by a novel theoretical model called Action-Homogeneous Nondeterministic Bit Vector Automata (AH-NBVA), its efficient hardware implementation, and a compiler that translates regexes into hardware configurations. BVAP is evaluated with a cycle-accurate simulator in a 28nm CMOS process, achieving 67-95% higher energy efficiency and 42-68% lower area, compared to state-of-the-art automata processors (CA, eAP, and CAMA), across a set of real-world benchmarks.

CCS Concepts: • **Theory of computation → Regular languages**; • **Hardware → Application specific processors**; • **Computer systems organization → Architectures**.

Keywords: Action-Homogeneous Nondeterministic Bit Vector Automata, Automata Processor, energy efficiency

* These authors contributed equally to this work.

ACM Reference Format:
Ziyuan Wen*, Lingkun Kong*, Alexis Le Glaunec, Konstantinos Mamouras, and Kaiyuan Yang. 2024. BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions with Bounded Repetitions . In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620665.3640412

1 Introduction

Regular pattern matching is useful in many application domains, including network security [49] and bioinformatics [4, 29]. Various algorithms have been developed based on nondeterministic finite automata (NFAs) or deterministic finite automata (DFAs). NFA-based regular pattern matching is widely adopted for hardware acceleration [8, 10, 16, 26, 30, 31, 37], as it combines the memory efficiency of NFAs (over DFAs) and leverages the inherent parallelism of hardware to efficiently simulate NFA execution.

Classical regular expressions can be translated into NFAs whose state space is linear in the size of the expression. However, regexes that arise in practice use the construct $r\{m, n\}$, called *bounded repetition*, which describes the repetition of the pattern r from m to n times. The naïve approach to deal with bounded repetition involves unfolding it. For example, the regex $r\{n, n\}$ is rewritten into the n-fold concatenation $r^n = r \cdot r \cdots r$. More generally, the bounded repetition $r\{m, n\}$ is equivalent to $r\{m, m\} \cdot r\{0, n-m\}$ and can therefore be unfolded into $r^m \cdot (r^?)^{n-m}$, where $r^?$ either matches the empty string or the pattern r. Unfolding $r\{m, n\}$ increases the size of the pattern by a factor of $\Theta(n)$. This increase can be substantial when the repetition bound n grows large. Due to the succinct encoding of repetition bounds (using decimal notation), the size of the representation of the counting operator $\{m, n\}$ is $\Theta(\log m + \log n)$, which is the same as $\Theta(\log n)$ because $m \leq n$. Therefore, regexes with bounded repetition can be exponentially more succinct than classical regexes. Bounded repetition is ubiquitous in practical use cases of regexes. Over the diverse collection of datasets that we consider, bounded repetition is found in 37% of the regexes and they account for 85% of all NFA states (after unfolding and regex-to-NFA translation).

To address the challenges raised by the widespread use of bounded repetition, we propose a novel approach that is not based on NFAs, but rather on automata models that represent counting in a more succinct way. In NFAs, each transition $q \rightarrow^{\sigma} q'$ is annotated with a character class σ (a subset of letters) and is enabled if the source state q is active and the current letter belongs to σ. We draw inspiration from the classical model of nondeterministic counter automata (NCAs) [11], where the states are extended with counters that can keep track of the number of repetitions. The regex `.*a.{100}` (in PCRE notation) can be represented as an NFA with 102 states by unfolding the repetition `.{100}`. Alternatively, it can be encoded with the following NCA that has three states:

NCA for `.*a.{100}`

$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[loop below, "."] \arrow[r, "a"] & q_1 \arrow[r, ". / x := 1"] & \boxed{q_2 : x} \arrow[loop below, ". , x < n / x{+}{+}"] \arrow[r, "x = n"] & {} \end{tikzcd}$$

The NCA has as many states as the number of character classes that appear in the regex. It uses the counter register x to keep track of the number of repetitions, so there is only one control state for the bounded repetition `.{100}`. In NCAs, a transition has the form $q \rightarrow^{\sigma,\varphi/\vartheta} q'$, where q is the source state, q' is the destination state, σ is a character class, φ is a predicate over the counter, and ϑ describes how to update the counter. NCA execution can be highly nondeterministic. Generally, it may require several different counter values for the same control state. This means that NCA simulation requires maintaining sets of counter values (not just a single value) at each counting control state.

The main idea of this work is the use of the model called ***Nondeterministic Bit Vector Automata*** (NBVAs) [18]. These automata enable efficient support for bounded repetition because they can encode the computation of NCAs in a way that is amenable to efficient execution on hardware. As mentioned earlier, NCA execution requires maintaining a set S of active counter values at a control state q. If q corresponds to a bounded repetition $\{m, n\}$, then S is a subset of the bounded set $\{1, 2, \ldots, n\}$ of possible counter values. A bit vector v with indexes $1, 2, \ldots, n$ can represent S in the following way: $v[i] = 1$ if $i \in S$ and $v[i] = 0$ if $i \notin S$. One can also think of v as the characteristic function of S. The following automaton corresponds to the NBVA for regex `.*.{100}`:

NBVA for `.*a.{100}`

$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[loop below, "."] \arrow[r, "a"] & q_1 \arrow[r, ". / v \cdot [1, 0, \ldots, 0]"] & \boxed{q_2 : x} \arrow[loop below, ". / \mathrm{shft}(v)"] \arrow[r, "v[n] = 1"] & {} \end{tikzcd}$$

We note that the topology of the NBVA is the same as that of the NCA shown earlier. So, the size of the NBVA's state space is *linear* in the size of the regex (because there is one state for each character class of the regex). In NBVAs, each control state carries a *bit vector* (BV) of some fixed size, and each transition $q \rightarrow^{\sigma/\vartheta} q'$ is annotated with a character class σ and a bit vector operation ϑ. The operation ϑ transforms the BV of q into a BV for q'. If many transitions with destination q' are enabled, the BV for q' is the bitwise OR of all BVs generated by the individual transitions. The use of bitwise OR in the NBVA corresponds to the union of sets of counter values in the corresponding NCA: if v_1 and v_2 are the bit vectors for the sets S_1 and S_2 respectively, then $v_1|v_2$ is the bit vector for $S_1 \cup S_2$ (we use | as notation for bitwise OR).

According to the previous discussion, the NBVA for a bounded repetition $r\{m, n\}$ is similar to the NFA for r, but each control state has a BV of size n. So, the main succinctness property is that a regex with bounded repetitions can be translated into an NBVA whose state space is *linear* in the size of the regex. To enable an efficient hardware implementation, we show that each NBVA can be equivalently transformed so that all incoming transitions for a state are annotated with the same BV operation ("action"). We call such automata ***Action-Homogeneous NBVAs*** or ***AH-NBVAs***. We establish that a small set of simple BV operations, all of which admit an efficient hardware implementation, suffice for representing regexes. This gives rise to a method for pattern matching that retains all practical benefits of NFAs while avoiding a blowup of the state space due to bounded repetition.

Co-designed with AH-NBVA, BVAP extends a state-of-the-art, in-memory NFA accelerator with bit vector modules (BVM) to execute bit vector processing at greatly reduced area and energy overheads. We adopt one of the latest designs, CAMA [16], as the baseline, but the proposed approach can be applied to other alternatives (see §2). BVMs support the bit-vector-processing phase of AH-NBVA with high programmability using a custom instruction set. BVM is custom-designed from the transistor level to achieve the desired functionalities using SRAMs and tiny peripheral circuits. On top of BVM, we build and optimize the entire architecture of BVAP, including the scheduling, control, I/O, and various reconfigurability to offer application-specific optimizations.

Our ***main contributions*** are the following:

(1) We have designed BVAP, the first energy and area efficient automata processor for regexes with bounded repetitions. BVAP is optimized across the circuit, architecture, and algorithms, leading to high energy efficiency and low memory usage while maintaining high programmability.

(2) We propose a novel method to transform NBVA into Action-Homogeneous NBVA, a more convenient representation for designing and programming hardware.

(3) We have a custom-designed Bit Vector Module (BVM) for efficient yet programmable bit vector processing and its integration with in-memory automata processors.

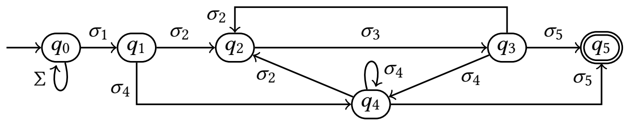
(4) We have developed a regex-to-hardware compiler for high-level programming of the hardware. This compiler implements the action-homogeneous transformation of NBVAs and translates the source regexes into configurations used to program BVAP.

(5) BVAP is evaluated with the 28nm CMOS process across seven real-world benchmarks. Compared with the state-of-the-art designs, BVAP reduces energy by 67%, 95%, and 94% over CAMA, CA, and eAP, while saving more than 30% of area. BVAP also supports a streaming input mode, BVAP-S, which achieves a constant but 67% less throughput and consumes 39% less energy due to lower voltage.

2 Preliminaries

Regular expressions or *regexes* are a widely used formalism for describing regular patterns. For a finite alphabet Σ, classical regexes over Σ are given by the grammar $r ::= \varepsilon \mid \sigma \mid (r\,|\,r) \mid r \cdot r \mid r^*$, where $\sigma \subseteq \Sigma$ is a predicate over the alphabet called a *character class*. The predicate Σ contains all symbols in the alphabet, which is similar to the notation `.` in PCRE-style syntax [25]. We write a for the singleton predicate $\{a\} \subseteq \Sigma$ and $[a_1 \dots a_n]$ for the predicate $\{a_1, \dots, a_n\} \subseteq \Sigma$. The grammar of regexes is often extended with more features for convenience and succinctness: $r^?$ indicates that the pattern r is optional and r^+ describes the repetition of r at least once. *Bounded repetition* (also called counting), written as $r\{m, n\}$, describes the repetition of r from m to n times. We say that m and n are the lower and upper bounds of the bounded repetition. The pattern $r\{m, n\}$ can be translated using concatenation and ? but is exponentially more succinct. The notation $r\{n\}$ is an abbreviation for $r\{n, n\}$. The abbreviation $r\{n,\} = r\{n\}r^*$ describes the repetition of r at least n times. The naïve approach for dealing with bounded repetition is to *unfold* it. For example, $r\{n\}$ 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). A regex can be converted to an NFA that recognizes the same language using the construction of Thompson [40] or Glushkov [13, 14]. We adopt the latter because it results in ε-free automata that are also *homogeneous*, i.e., all incoming transitions of a state are labeled with the same character class. Let Σ be a finite alphabet. A ***Glushkov NFA*** or GNFA with input alphabet Σ is a tuple $\mathcal{A} = (Q, L, \Delta, I, F)$, where Q is a finite set of *(control) states*, $L : Q \to \mathcal{P}(\Sigma)$ is a function that maps each state to a character class, $\Delta : Q \to \mathcal{P}(Q)$ is the *transition relation*, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*.

Example 2.1. Consider the regex $\Sigma^*\sigma_1(\sigma_2\sigma_3|\sigma_4)^*\sigma_5$, where $\sigma_1, \sigma_2, \dots, \sigma_5$ are character classes. The following GNFA recognizes the language of this regex:



This homogeneous GNFA has six control states; q_0 is the initial state and q_5 is the final state. Edges that lead to the same state are labeled with the same character class.

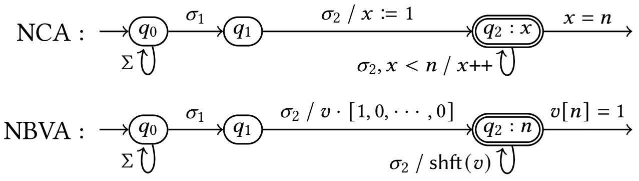
Nondeterministic counter automata (NCAs) [17, 42] extend NFAs with counter registers. In an NCA, a computation involves not only transitions between control states but also the use of a finite number of registers that hold nonnegative integers. NCA is a natural execution model for regexes with bounded repetitions. *Nondeterministic bit vector automata* (NBVAs) [18] are expressively equivalent to NCAs if the counters are bounded. The configuration of the NBVA specifies for each control state q a bit vector to represent the set of counter values that are on the control state q.

We fix an infinite set *CReg* of counter registers or, simply, *counters*. 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 *(control) states* and $R : Q \to \mathcal{P}(CReg)$ is a function that maps each state to a finite set of counters. The *transition relation* Δ 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 $\vartheta : (R(p) \to \mathbb{N}) \to (R(q) \to \mathbb{N})$. The *initialization function* I specifies the set $I(q) \subseteq R(q) \to \mathbb{N}$ of initial valuations for each state q. The *finalization function* F specifies the set $F(q) \subseteq R(q) \to \mathbb{N}$ of final valuations for each state q.

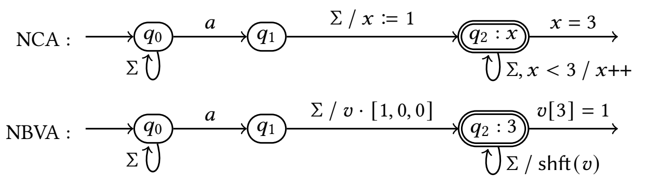
Some states in an NCA may not have any counter at all. In a transition $(p, \sigma, \varphi, q, \vartheta)$, we will call the predicate φ a *guard* because it may restrict a transition based on the values of the counters, and we will call the function ϑ an *assignment* as it describes how to assign counter values to the destination state given the counter values in the source state.

A ***nondeterministic bit vector automaton*** (NBVA) is a tuple (Q, w, Δ, I, F), where Q is a finite set of *(control) states*, and $w : Q \to \{1, 2, \dots\}$ is a function that maps each state to a strictly positive integer. The *transition relation* Δ contains finitely many transitions of the form $(p, \sigma, q, \vartheta)$, where p is the source state, $\sigma \subseteq \Sigma$ is a predicate over the alphabet, q is the destination state, and $\vartheta : \mathbb{B}^{w(p)} \to \mathbb{B}^{w(q)}$. The *initialization function I* specifies an *initial vector* $I(q) : \mathbb{B}^{w(q)}$ for each state q, and the *finalization function F* specifies a function $F(q) : \mathbb{B}^{w(q)} \to \mathbb{B}$ for each state q. A state q is *initial* if $I(q) \neq \mathbf{0}_{w(q)}$, where $\mathbf{0}_{w(q)}$ is the zero vector of length $w(q)$. A state q is *final* if $F(q)(v) = 1$ for some $v \in \mathbb{B}^{w(q)}$.

Example 2.2. Consider the regex $r = \Sigma^*\sigma_1\sigma_2\{n\}$ with $n \geq 1$, where σ_1 and σ_2 are character classes. The following NCA and NBVA recognize the language of r:



The NCA has three states. We write $q_2 : x$ to indicate that q_2 has a counter register x. Notice that q_0 and q_1 have no annotation with counters, which means they have no counter.



input	NCA configuration	NBVA configuration			output
	q_0	$q_0 : 1,$	$q_1 : 0,$	$q_2 : [0,0,0]$	0
b	q_0	$q_0 : 1,$	$q_1 : 0,$	$q_2 : [0,0,0]$	0
a	q_0, q_1	$q_0 : 1,$	$q_1 : 1,$	$q_2 : [0,0,0]$	0
b	$q_0, (q_2, 1)$	$q_0 : 1,$	$q_1 : 0,$	$q_2 : [1,0,0]$	0
a	$q_0, q_1, (q_2, 2)$	$q_0 : 1,$	$q_1 : 1,$	$q_2 : [0,1,0]$	0
a	$q_0, q_1, (q_2, 1), (q_2, 3)$	$q_0 : 1,$	$q_1 : 1,$	$q_2 : [1,0,1]$	1
b	$q_0, (q_2, 1), (q_2, 2)$	$q_0 : 1,$	$q_1 : 0,$	$q_2 : [1,1,0]$	0
a	$q_0, q_1, (q_2, 2), (q_2, 3)$	$q_0 : 1,$	$q_1 : 1,$	$q_2 : [0,1,1]$	1
a	$q_0, q_1, (q_2, 1), (q_2, 3)$	$q_0 : 1,$	$q_1 : 1,$	$q_2 : [1,0,1]$	1
a	$q_0, q_1, (q_2, 1), (q_2, 2)$	$q_0 : 1,$	$q_1 : 1,$	$q_2 : [1,1,0]$	0

Figure 1. Execution of NCA and NBVA for regex $\Sigma^* a \Sigma\{3\}$.

We annotate each edge $p \longrightarrow q$ with an expression of the form $\sigma, \varphi \,/\, \vartheta$, where σ is a character class, φ is a guard over p's counters, and ϑ is an assignment for the counters of q using p's counters. If the guard φ is omitted, then it is always true. The assignment ϑ is omitted only when the counters retain the values from the previous state (i.e., "$x := x$"). 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 of the value n to x. We use double circle notation to indicate that a state is final (see q_2). An arrow emanating from a final state is annotated with a predicate (see $x = n$ for q_2) over counter valuations. A match is reported if the final state is active and the counter value satisfies the predicate.

For NBVA, we write "$q_2 : n$" to indicate that $w(q_2) = n$, i.e., q_2 carries a bit vector of size n. States q_0 and q_1 have no annotation, i.e., they have no bit vector. We annotate each edge $p \longrightarrow q$ with an expression of the form $\sigma \,/\, \vartheta$, where σ is a predicate over Σ, and ϑ is a function for computing the bit vector of q using the bit vector of p. We use v as a symbol that represents the bit vector of p. As further explanation: (1) We write $[1, 0, \ldots, 0]$ to denote the bit vector that is zero everywhere, except for position 1, where it is equal to 1. (2) We write "$v[n] = 1$" to denote the function of type $\mathbb{B}^n \to \mathbb{B}$ that checks whether the value of v at the n-th position is equal to 1. (3) We write "$\mathsf{shft}(v)$" to denote the function of type $\mathbb{B}^n \to \mathbb{B}^n$ that shifts a bit vector by one position. More formally, the shift operation is defined as: $\mathsf{shft}(v)[1] = 0$ and $\mathsf{shft}(v)[i] = v[i-1]$ for every $i = 2, \ldots, n$. (4) We use double circle notation to indicate that a state is final (see state q_2). An arrow emanating from a final state q is annotated with a description of the function $F(q) : \mathbb{B}^{w(q)} \to \mathbb{B}$.

Figure 1 shows the execution of the NCA and NBVA for $\Sigma^* a \Sigma\{3\}$. With the provided input sequence, the control state q_2 may carry several counter values, i.e., its bit vector has several bits set to '1'. In NBVAs, when a state q has several incoming transitions that produce several bit vectors for q, the bit vector for q becomes the bitwise OR over them.

During NCA simulation, an NCA operation f (e.g., $x < n \,/\, x{+}{+}$ on a loop backedge) is applied to all values of S to obtain $f(S) = \{f(c) \mid c \in S\}$. A consequence of this is that f (on sets) commutes with union: $f(S_1 \cup S_2) = f(S_1) \cup f(S_2)$. Suppose that g is (e.g., "shift") the corresponding operation on bit vectors. Then, it should hold that $g(v_1|v_2) = g(v_1)|g(v_2)$, where v_1, v_2 are the bit vectors for S_1, S_2 respectively. This equation says that the operation g on bit vectors is *linear* (with respect to bitwise OR). NBVAs with non-linear operations do not have an immediate correspondence to NCAs.

In-memory Automata Processor. A plethora of recent automata processors based on in-memory computing have shown significant promise for efficient regex matching. AP [10] achieves more than 10× better performance than existing CPU [19] and GPU [5] architecture and accelerator XeonPhi. CA [37] proposes an 8-transistor (8T) Fully connected CrossBar (FCB) for routing and aggregation (logic OR) of active states. eAP [31] creates Reduced CrossBar (RCB) exploiting the sparsity of the *switch network* to decrease the area. Impala [30] and CAMA [16] made critical improvements to AP by proposing efficient encoding schemes to reduce memory usage. CAMA further uses content addressable memory (CAM) to reduce energy and memory usage. These designs typically adopt a two-phase architecture: a ***state-matching phase*** for identifying the currently activated states and a ***state-transition phase*** for detecting available states in the next cycle. They use STEs to represent states, where each STE carries a predicate such that it will fire a signal '1' (i.e., it is matched) if an input symbol satisfies this predicate. The exact operation of AP-style processors is detailed in §3 and Figure 3(a), using CAMA as an example. AP-style processors support regexes found in many real-world applications [44, 45] but provide limited and inefficient support for regexes with bounded repetitions.

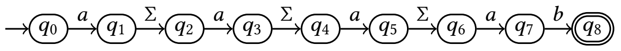
3 Design Principle of BVAP

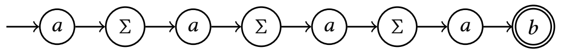
In this section, we present the design principle of BVAP. We first discuss how current ASIC architectures such as AP and CAMA struggle to simulate regexes with bounded repetition efficiently. Next, we propose a naïve solution that integrates bit vectors into existing frameworks to minimize the number of STEs required for regex matching. We further introduce our BVAP solution, which substantially reduces the hardware resources compared to the naïve solution.

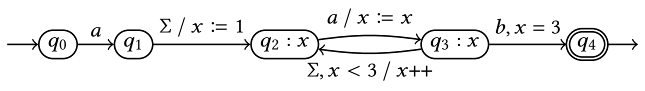
Existing Solution with Unfolding. As a motivating example, let us consider the matching of regex $a(\Sigma a)\{3\}b$. Recall a and b are predicates that match input symbols a and b respectively, while Σ matches any symbol. Fig. 2(a) shows

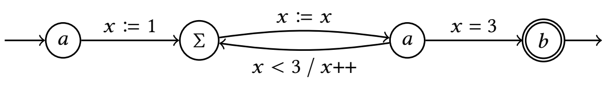
Table 1. Sample execution of the naïve design in Fig. 3(b), where (1) the column STEi contains a bit indicating whether STEi is active ('1' means active), (2) bv$i$→ is the initial value of bit vector in PEs associated to STEi (it will be all '0's if STEi is inactive), (2) fields `set1`, `shift`, `copy`, and `r(3)` show the value of bit vectors on the corresponding crossing points after the operation, (3) →bv$i$ is the updated value of bit vector in bv$i$.

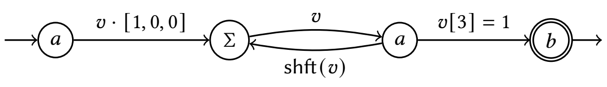
input	STE1	STE2	STE3	STE4	bv1→	bv2→	bv3→	bv4→	`set1`	`shift`	`copy`	`r(3)`	→bv2	→bv3	→bv4
a	1	0	0	0	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]	[0, 0, 0]
b	0	1	0	0	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]
a	1	0	1	0	[0, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 1, 0]	[0, 0, 0]	[0, 0, 0]	[1, 1, 0]	[0, 0, 0]	[0, 0, 0]
a	1	1	0	1	[0, 0, 0]	[1, 1, 0]	[0, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]	[1, 1, 0]	[0, 0, 0]	[1, 0, 0]	[1, 1, 0]	[0, 0, 0]
a	1	1	1	0	[0, 0, 0]	[1, 0, 0]	[1, 1, 0]	[0, 0, 0]	[1, 0, 0]	[0, 1, 1]	[1, 0, 0]	[0, 0, 0]	[1, 1, 1]	[1, 0, 0]	[0, 0, 0]
b	0	1	0	1	[0, 0, 0]	[1, 1, 1]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[1, 1, 1]	[0, 0, 0]	[0, 0, 0]	[1, 1, 1]	[0, 0, 0]
a	1	0	1	0	[0, 0, 0]	[0, 0, 0]	[1, 1, 1]	[0, 0, 0]	[1, 0, 0]	[0, 1, 1]	[0, 0, 0]	[1, 1, 1]	[1, 1, 1]	[0, 0, 0]	[1, 1, 1]
b	-	-	-	1	-	-	-	[_, _, 1]	-	-	-	-	-	-	-

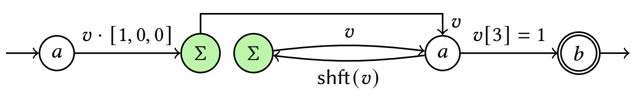

(a) Glushkov NFA for regex $a(\Sigma a)\{3\}b$ constructed by unfolding


(b) Simplified diagram for Glushkov NFA for regex $a(\Sigma a)\{3\}b$


(c) NCA for regex $a(\Sigma a)\{3\}b$


(d) Simplified NCA for regex $a(\Sigma a)\{3\}b$


(e) Simplified NBVA for regex $a(\Sigma a)\{3\}b$


(f) Action-homogeneous NBVA for regex $a(\Sigma a)\{3\}b$

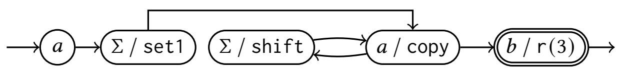

(g) Simplified action-homogeneous NBVA for regex $a(\Sigma a)\{3\}b$

Figure 2. Automata constructed from the regex $a(\Sigma a)\{3\}b$.

the NFA constructed from this regex, where $a(\Sigma a)\{3\}b$ is unfolded into $a\Sigma a\Sigma a\Sigma ab$. The Glushkov construction ensures that all transitions entering a state are labeled with the same predicate. This property allows us to simplify the NFA diagram to a hardware-friendly representation by omitting the initial state and pushing the predicates from the edges to the states. For example, we push the predicate a into state q_2 so that in Fig. 2(b) we have a state labeled with the predicate a.

Existing ASIC designs for processing automata, such as AP [10], CA [37], eAP [31] and CAMA [16], support the matching of $a(\Sigma a)\{3\}b$ by unfolding and executing the corresponding NFA. Fig. 3(a) illustrates how CAMA, a state-of-the-art ASIC automata processor, executes this task. CAMA uses STEs to represent states, where each STE corresponds to a state in Fig. 2(b). In the state-matching phase of each processing cycle, if the STE is active, it will send '1' to a buffer called *active buffer* in Fig. 3(a). We call an STE ***active*** iff it is both available and matched by an input element. The execution of this design involves two steps: routing and aggregation. In the first step, the signal stored in the active buffer will be routed to another STE by a switch network. If an STE (e.g., STE1 in Fig. 3(a)) connects to another STE (e.g., STE2), a dot is put on the crossing point of the switch network. The value of the dot is '1' iff the STE placed on the corresponding column is active. In the second step, a logic OR aggregation is performed on each row of the switch network. If the aggregation result is '1', the STE on the corresponding row will become available in the next cycle. STE1 in Fig. 3(a) is an initial STE. Users can search for a *partial match* of regexes over an input sequence by setting STE1 as available for each input. STE8 in Fig. 3(a) is a reporting STE, which will report a match if it is active.

However, unfolding is resource-intensive because the increasing number of STEs and the size of the switch network lead to significant growth of memory and energy requirements. For instance, the matching of the regex $a(\Sigma a)\{n\}b$ requires $\Theta(n)$ STEs and a switch network of size $\Theta(n^2)$.

Naïve Solution with Bit Vectors. A design leveraging NBVAs can address the inefficiencies of the unfolding solution. Fig. 2(c) shows the NCA for $a(\Sigma a)\{3\}b$ and Fig. 2(d) simplifies the diagram by exploiting homogeneity. Fig. 2(e) shows the corresponding NBVA, where a match is reported if the state labeled with b is active.

Fig. 3(b) shows this BV-based design. Execution in this design involves three steps: routing, PE computation, and aggregation. In the first step, instead of the switch network, a processing element (PE) array is used to route the bit vectors.

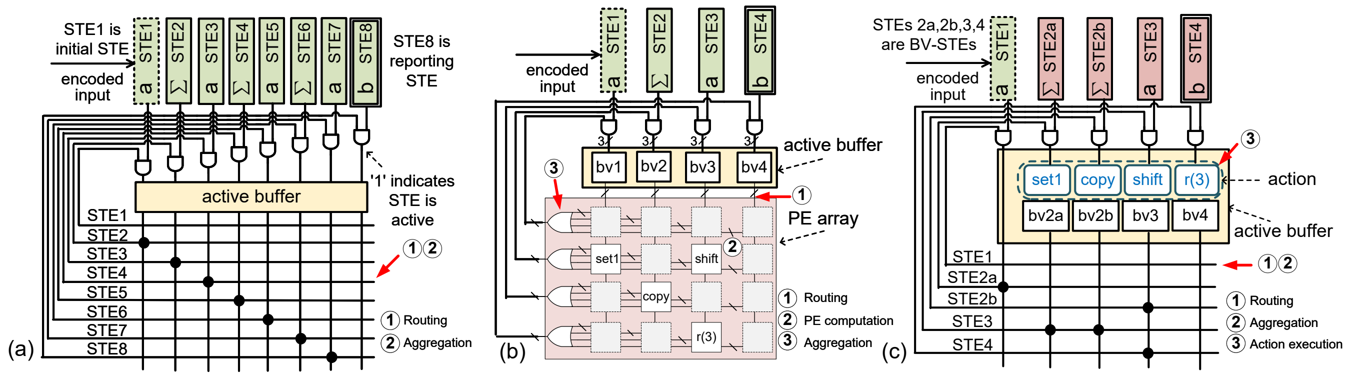


Figure 3. Simplified diagrams of (a) CAMA design with unfolding, (b) the naïve design using bit vectors, and (c) BVAP based on action-homogeneous automata, and their configuration to match regex $a(\Sigma a)\{3\}b$.

The bottom of Fig. 3(b) shows such a PE array, which routes the initial value of bit vectors stored in the active buffer to the corresponding PE for computation. Notice if STEi is not matched, the initial value of bv$i$ will be set to all '0's. Each PE supports a set of operations, including `set1` that sets the first bit to 1 in the bit vector (i.e., $v \cdot [1, 0, 0]$ in Fig. 2(e)), `copy` that copies the value of bit vector (i.e., v), and `shift` that shifts the bit vector (i.e., $\mathrm{shft}(v)$). Moreover, $\mathtt{r}(n)$ reads the nth bit from the bit vector, and it will copy the bit vector if this bit is '1' (i.e., $v[3] = 1$), otherwise it will set the bit vector to be all '0's. In the PE computation step, each PE will perform its operation over the received bit vector. Finally, in the third step, the PE array will aggregate the results of PEs on the same row with a bitwise-OR computation. This aggregating result will be used as the initial value of the corresponding bit vector in the active buffer in the next cycle.

Table 1 illustrates the execution of the naïve design in Fig. 3(b) for detecting the partial match of the regex over an input sequence "*abaaabab*", i.e., STE1 is active for each input element that is *a*. If an STE is not matched by the current input element, it will set the value of its associated bit vectors to be all '0's. STE4 is a reporting STE, which will report a match if it is active and the corresponding bit vector (i.e., bv4) has '1' on the third bit at the beginning of a cycle.

With this design, the number of STEs becomes independent of the repetition bound. For example, regex $a(\Sigma a)\{n\}b$ only requires four STEs here, instead of $2n$ STEs when unfolded. However, the hardware resources required by the PE array in this naïve Bit Vector design grow quadratically with the number of STEs supported in one tile, because each node in the routing switch needs one PE. Considering the common choice of 256 STEs per tile in state-of-the-art automata processors [16, 31, 37], which is determined by the tradeoff between the desired STE connectivity and the quadratically scaling crossbar size, the maïve NBVA approach suffers from significant area and power overheads in the PE array, which in turn reduces the benefits from STE reduction.

BVAP Solution. We propose the BVAP design to further reduce memory and energy costs. BVAP is based on the action-homogeneous NBVA (i.e., AH-NBVA) model, where the term ***action*** refers to an operation over bit vectors. This design is inspired by the homogeneity property for NFAs, where states (instead of transitions) are annotated with character classes. Our insight is that we can transform an NBVA so that for each control state q, all transitions entering q are labeled with the same action. We call an NBVA that satisfies this property *action-homogeneous*. Fig. 2(f) shows the AH-NBVA obtained from the NBVA of Fig. 2(e). The key idea of the transformation is to split a state with several different incoming actions into multiple copies. Each of these copies has the same action over all its incoming transitions and inherits the outgoing transitions from the original state. For instance, the state labeled with Σ in Fig. 2(e) has two actions, $v \cdot [1, 0, 0]$ and $\mathrm{shft}(v)$, on its input edges. This state is split into the two green STEs shown in Fig. 2(f). Fig. 2(g) shows a simpler AH-NBVA diagram, where `set1`, `shift`, `copy`, `r(3)` correspond to $v \cdot [1, 0, 0]$, $\mathrm{shft}(v)$, $v := v$, and $v[3] = 1$ respectively.

Fig. 3(c) shows the conceptual diagram of BVAP, where STE2a and STE2b are obtained by splitting STE2 with the AH transformation. In AH-NBVAs, we assign BV actions to STEs (instead of transitions), and we call an STE carrying a bit vector a ***BV-STE***. STEs 2a, 2b, 3, and 4 are BV-STEs, which are activated by signals stored in the active buffer such that '1' allows BV-STEs to perform corresponding actions, and '0' sets the value of the bit vector to '0's. STE4 is the reporting STE, which reports a match if both the STE is active and the third bit of its associated bit vector (i.e., bv4) is '1'. BVAP performs state-matching and state-transition in each cycle, just like AP-style designs, but with an additional ***bit-vector-processing phase*** including three operations - routing, aggregation, and action execution. First, the value of the bit vector stored in the active buffer is routed to its corresponding BV-STEs. Next, a bitwise-OR aggregation is performed on all bit vectors that are routed to the same BV-STE. Finally, each BV-STE executes its corresponding

Table 2. Sample execution of the BVAP design in Fig. 3(c), where (1) STE*i* shows whether STE*i* is active, (2) bv*i* → is the initial value of bit vector, which is set to '0's if STE*i* is not active, and (3) →bv*i* is the updated bit vector values for the next cycle.

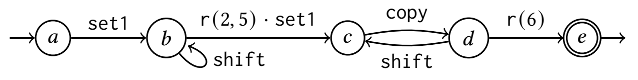
input	STE1	STE2a	STE2b	STE3	STE4	bv2a→	bv2b→	bv3→	bv4→	→bv2b	→bv3	→bv4
a	1	0	0	0	0	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]
b	0	1	0	0	0	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]
a	1	0	0	1	0	[0, 0, 0]	[0, 0, 0]	[1, 0, 0]	[0, 0, 0]	[0, 1, 0]	[0, 0, 0]	[0, 1, 0]
a	1	1	1	0	0	[0, 0, 0]	[0, 1, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[1, 1, 0]	[0, 0, 0]
a	1	1	0	1	0	[0, 0, 0]	[0, 0, 0]	[1, 1, 0]	[0, 0, 0]	[0, 1, 1]	[1, 0, 0]	[0, 1, 1]
b	0	1	1	1	1	[0, 0, 0]	[0, 1, 1]	[1, 0, 0]	[0, 1, 1]	[0, 1, 0]	[1, 1, 1]	[0, 1, 0]
a	1	0	1	1	0	[0, 0, 0]	[0, 1, 0]	[1, 1, 1]	[0, 0, 0]	[0, 1, 1]	[0, 1, 0]	[0, 1, 1]
b	-	-	-	-	1	-	-	-	[_,_,1]	-	-	-

action to update its bit vector. In contrast to the naïve design (Fig. 3(b)), BVAP executes the action after the aggregation of bit vectors. If we view a BVAP action as a function f applied to the bit vector, all BVAP actions satisfy the following *linearity* property: for bit vectors v_1, v_2 of the same size, $f(v_1)|f(v_2) = f(v_1|v_2)$. This property ensures the naïve design and BVAP generate consistent results. BVAP uses Bit Vector Module (BVM) to perform computations during the bit-vector-processing phase (see §5).

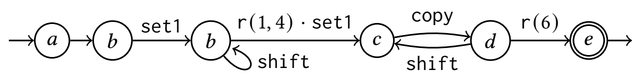
BVAP needs $O(1)$ STEs for $a(\Sigma a)\{n\}b$ since the AH transformation only adds a constant number of STEs. By using BV-STEs instead of a PE array, BVAP greatly reduces the memory requirements. Table 2 illustrates the execution of BVAP, which corresponds to the design in Fig.3(c). The action assigned to each BV-STE is performed on the initial value of bit vectors at the beginning of each processing cycle. Compared to existing automata processors, BVAP efficiently supports regexes with large bounds, e.g. `url=.{8000}` in Snort requires 8004 STEs when unfolded and only 270 STEs in BVAP. Previous AP-style hardware is limited to at most 4096 STEs per regex. Another example can be found in the ClamAV dataset, where `\x43\x30\x30\x30.{9139}` `\x65\x6e\x75\x00` represents two sequences of characters interleaved by 9139 characters and cannot be supported by state-of-the-art designs.

4 Action-Homogeneous Transformation

The following set of operations is sufficient for NBVAs that are constructed from regexes: (1) `set1`, (2) `shift`, (3) copy, (4) $\mathtt{r}(n)$, (5) $\mathtt{r}(m, n)$ that returns 1 if any of the bits $v[m]$, $v[m+1]$, ... $v[n]$ is equal to 1, (6) $\mathtt{r}(n)\cdot$`set1`, and (7) $\mathtt{r}(m, n)\cdot$`set1`. E.g., the NBVA for $ab\{2, 5\}(cd)\{6\}e$ is shown below:

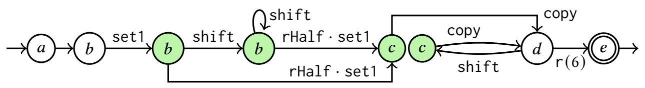


Supporting the operation $\mathtt{r}(m, n)$ (for all possible choices of m and n) in hardware is not practical, because this would require a fully configurable readout circuit. By rewriting $r\{m, n\}$ into the equivalent $r\{m-1\}r\{1, n-m+1\}$, we restrict the set of operations that need to be supported. For example, $ab\{2, 5\}(cd)\{6\}e$ is rewritten into $abb\{1, 4\}(cd)\{6\}e$.



So, it suffices to support read operations $\mathtt{r}(n)$ and $\mathtt{r}(1, n)$.

The theoretical NBVA model allows control states to have bit vectors of different sizes. For the hardware implementation, however, it is desirable that all bit vectors are of the same size K. Bounded repetitions can be partially unfolded so that bit vectors of size K suffice (see §7). In order to optimize the read operations from the SRAM (see §5), the operation $\mathtt{r}(1, n)$ for all possible values $n \leq K$ is undesirable. We choose to support the operations $\mathtt{r}(1, K)$, $\mathtt{r}(1, K/2)$ and $\mathtt{r}(1, K/4)$, which we also denote with `rAll`, `rHalf` and `rQuarter` respectively. The figure below shows the AH-NBVA for the regex $abb\{1, 4\}(cd)\{6\}e$ with $K = 8$.



The operations needed for AH-NBVA are: `set1` which creates a bit vector which is '0' everywhere except for the lowest position; `copy` which copies a bit vector from one state to another; `shift` which shifts a bit vector and fills in a zero at the lowest position, and $\mathtt{r}(n)$ which reads the nth bit of the bit vector (1-based indexing). Given a k-bit vector, `rAll` (resp., `rHalf`, `rQuarter`) returns '1' if any of the first k (resp., $k/2$, $k/4$) bits is '1'. We also support combination operations that apply `set1` if the first operation returns '1', including $\mathtt{r}(n)\cdot$`set1`, `rAll` · `set1`, `rHalf` · `set1`, and `rQuarter` · `set1`. These operations create a bit vector that is '0' everywhere except for the lowest position if the first operation returns true, otherwise, they create a bit vector with all '0's.

The transformation from an NBVA $\mathcal{A}$ to an equivalent AH-NBVA $\mathcal{B}$, which we call ***AH transformation***, identifies states that violate the AH property and splits them in several copies, one for each kind of incoming action. Let us focus on a state q that has distinct incoming actions $\vartheta_1, \ldots, \vartheta_k$. We create k copies $q_1, \ldots, q_k$ of the state. If there is a transition $p \to^{\sigma/\vartheta_i} q$ in $\mathcal{A}$, we add the transition $p \to^{\sigma/\vartheta_i} q_i$ to $\mathcal{B}$. If there is a transition $q \to^{\sigma/\vartheta} q'$ in $\mathcal{A}$, then we add in $\mathcal{B}$ a transition $q_i \to^{\sigma/\vartheta} q'$ for every $i = 1, \ldots, k$.

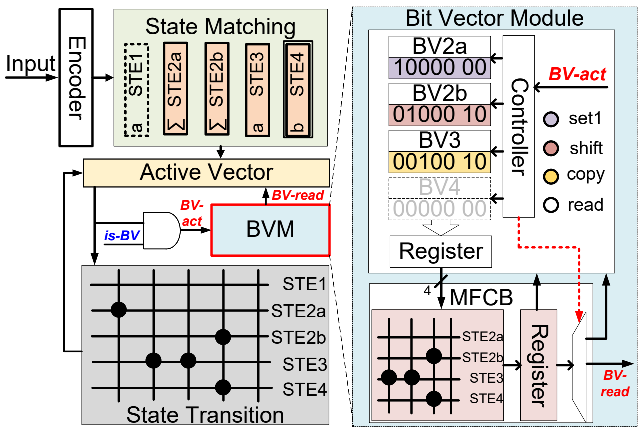


Figure 4. Diagrams of the proposed Bit Vector Module (BVM) and its integration with CAMA. A toy example of matching regex $a(\Sigma a)\{3\}b$ is illustrated.

5 Hardware Primitives of BVAP

This section covers the key hardware primitives to enable AH-NBVA processing in a state-of-the-art in-memory automata processor. Since we adopt the exact hardware designs for ***state matching*** and ***state transition*** as CAMA [16] (see §3 and Fig. 3 (a)), this section focuses on the Bit Vector Module (BVM) to enable ***bit-vector-processing***.

Bit Vector Module (BVM). Our BVM contains a cluster of *Bit Vectors* (BV), a *Multi-bit Fully-connected CrossBar* (MFCB), and a local controller (see Fig. 4). Each BV stores a bit vector corresponding to a BV-STE in an AH-NBVA model and executes BV operations based on a small custom instruction set defined in Table 3. We further equip each BV with an instruction buffer to individually program its action, since our AH-NBVA model demands each BV-STE executing its own action. During the bit-vector-processing phase, BVM updates the value of BVs attached to activated BV-STEs. As detailed in §3, bit-vector-processing involves three operations: aggregation, routing, and action execution. The aggregation (bitwise OR) and routing of BVs are performed inside the MFCB, which works similarly to the state transition crossbar first introduced in CA [37]. The routing of MFCB is pre-programmed based on compiled AH-NBVAs. We introduce three key optimizations to BVM beyond the conceptual diagram shown in Fig. 3(c).

First, we separate routing switches for *state transition* and *bit-vector-processing* to enhance the bandwidth of BV routing, because each bit vector contains many more bits than a normal STE. As such, BVM can be employed as an add-on module to state-of-the-art in-memory automata processors, which only needs to communicate with the Active Vector block that stores and controls the active STEs. The input to BVM is *BV-act*, signaling the activated BV-STEs. It is a masked subset of the *active buffer* in our baseline (see Fig. 3(a)). A `is-BV` mask is added to allow reconfiguring a BV-STE into a normal STE and gate the corresponding BV when there are more BV-STEs than what a regex requires. The output of BVM is *BV-read*, which is the result of $\texttt{r}(n)$ or $\texttt{r}(1,n)$. It is returned to the Active Vector to deactivate BV-STEs with read failures.

Second, all read actions are performed at the source BV-STEs, so that only the output bits of read actions (*BV-read*) need to route, rather than the entire bit vector, saving significant routing energy. To realize such an optimization, BVM executes in two sequential steps, Read and Swap, as illustrated in Fig. 5. In the Read step, each BV performs the read actions, and the results (*BV-read*) are routed through MFCB and stored in MFCB's output buffer. The default *BV-read* value for a BV-STE is '1' if its instruction is no-read. In addition to routing, the MFCB aggregates (bitwise OR) all read outputs with the same destination. Lastly, based on aggregated read outputs, certain BV-STEs are deactivated and the corresponding BVs are reset. The Swap step reads bit vector values from BVs, routes and aggregates them before writing back to the destination BVs. Swap step is used to execute `copy`, `shift`, and `set1` actions (see Table 3).

Table 3. Instruction set of BVAP.

	Index	Value	Action	Step
Instruction Code	0	0	No `set1` action	Swap
		1	`set1`	
	1	0	`copy`	Swap
		1	`shift`	
	2-4	0xx	no-read action	Read
		100	`r` (i.e., read)	
		101	`rQuarter`	
		110	`rHalf`	
		111	`rAll`	
Pointer	5-10	0-63	Pointer to n in $\texttt{r}(n)$	Read

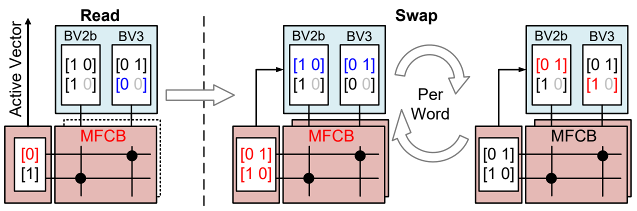


Figure 5. Read and Swap steps in bit-vector-processing phase. The STE2b and STE3 for regex $a(\Sigma a)\{3\}b$ are used for illustration, where blue/red texts indicate values being read and written.

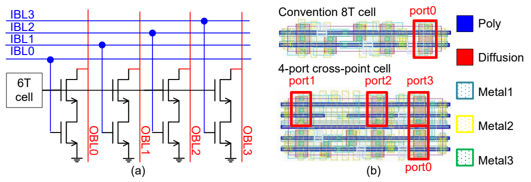


Figure 6. The 4-port cross-point design and layout.

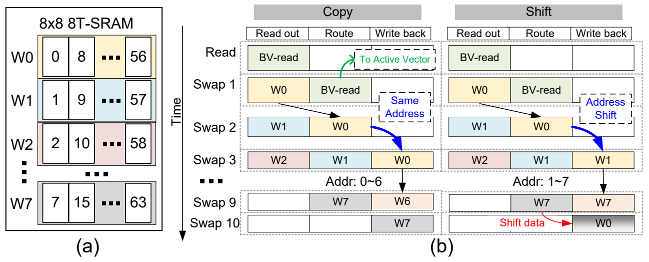


Figure 7. (a) Bit vector storage map; (b) Pipelining of bit-vector-processing phase for `copy` and `shift` actions.

Third, we designed a semi-parallel routing strategy for implementing the aforementioned Swap step. Parallel routing of n-bit bit vectors using n FCBs takes 1 cycle but requires a large area, while serial routing incurs n times longer latency and n times system throughput reduction than the parallel alternative. Our strategy is to split each bit vector into words of the same bit-width as the MFCB, and execute the Swap step word by word until the entire BV is updated, to balance area and throughput overheads.

Circuit Design of MFCB. Implementing MFCB with multiple FCBs is straightforward but inefficient because all bits share identical routes and routing memories are duplicated. Here, we employ a 4-port cross-point circuit for MFCB (see Fig. 6(a)). Each cross-point comprises a 6T SRAM cell storing routing information and four independent 2T ports. Each port operates with its own input bitline (IBL) and output bitline (OBL). In the 28nm layout, we find having four ports in each cell is optimal as it fully utilizes the metal routing tracks. Adding more ports will lead to wasted area and reduce overall density. Fig. 6(b) shows the layout of our 4-port cross-point using the logic design rule. Compared to using four conventional 8T cells, the 4-port cell saves 30% area without compromising functionality. In our final design, each MFCB contains two 4-port cross-points to process 8 bits per cycle.

Circuit Design of Bit Vector (BV). While bit vectors are common in hardware designs [6, 36], conventional register-based designs exhibit critical drawbacks for usage in BVAP. First, since each BV-STE requires a BV, register-based logic takes significant area and energy. Second, the latency of BVM is predominantly constrained by the bandwidth of routing switches, rendering the high bandwidth of register-based BVs unnecessary. Third, supporting the read action $\mathtt{r}(1, n)$ requires an additional reconfigurable OR tree.

The size of bit vectors is an important design parameter that affects the trade-off between area and throughput. Our design space exploration (see §8) suggests that the optimal bit vector size is less than 64 across all benchmarks. Hence, we custom-designed a compact 64-bit Bit Vector (BV) circuitry consisting of one tiny 8T-SRAM array, instruction latches, and control logic. Similar to routing switches, we employ 8T-SRAM to efficiently perform OR logic on all cells sharing the same bitline, which is essential to realize $\mathtt{r}(1, n)$ action for BV. 8T-SRAM-based BV achieves more than 50% area reduction over a register-based alternative and supports simultaneous reading and writing of two words in a single cycle, which is essential for the proposed semi-parallel implementation of BV actions. The dimension of the 8T-SRAM array is made 8×8 to match the maximum bandwidth of MFCB (8 bits per cycle). Here, we map the 64-bit bit vector along the bitlines of the 8T-SRAM, as shown in Fig. 7(a), in order to simplify address-controlled action `shift` and support virtual BV sizes. *Virtual BV* is designed to efficiently support scenarios that benefit from shorter bit vectors even when the hardware BV size is fixed to 64 because it reduces cycles and energy to process a BV in our semi-parallel scheme. The virtual BV is configured by the compiler (see §7) and realized by simply adjusting the number of Swap steps in the BVM controller.

In the Read step, BV executes read actions $\mathtt{r}(n)$ and $\mathtt{r}(1, n)$ as described in § 4. For $\mathtt{r}(n)$, the bit vector value at bit position n is read from the SRAM and sent to MFCB. For $\mathtt{r}(1, n)$, because bitwise OR logic within the 8T-SRAM can only be performed along the eight bitlines, the hardware restricts the choice of n. Given a K-bit (virtual) BV, our design supports $\mathtt{r}(1, K/4)$, $\mathtt{r}(1, K/2)$, and $\mathtt{r}(1, K)$ by combining the OR results of 2, 4, and 8 bitlines. These three actions are programmed by `rQuarter`, `rHalf`, and `rAll` instructions. AH-NBVA can be compiled with only these three $\mathtt{r}(1, n)$ actions (see §4) Meanwhile, all inactive BVs are reset by raising all RWLs and writing '0' to all cells in one cycle.

In the Swap step, actions `copy`, `shift`, and `set1` are executed as shown in Fig. 7(b). A BV first reads and routes one word through the MFCB to update the word. Depending on its own instruction, each BV selects from two possible writing addresses generated by the local controller of BVM. For `copy`, the writing address will be the same as the word reading address. For `shift`, the writing address is the reading address plus one. The Swap step works in a pipeline manner with a 3-cycle latency, in order to avoid data hazards caused by `shift` without performance compromise. During `shift`, when writing the last word to the first address, the word must be right-shifted by one bit, and padded with '0'. When a BV takes the `set1` instruction, it is power-gated except for a simple logic that sends the stored constant to the MFCB.

Working Example. Fig. 4 shows a working instance of BVAP for matching $a(\Sigma a)3b$ with the AH-NBVA shown in Fig. 2(g). There are five STEs in total, STE1 is a standard STE while the other four STEs are BV-STEs. The action of each BV-STE is programmed with the instruction set. STE1 is activated upon matching an input character a. If the following character is Σ, STE2a will be activated. When STE2a is activated, BV2a will send the constant set value (2'b1000) to STE3 through MFCB. For every cycle that BVM is active, the aggregation of BV2a and BV2b values is sent to BV3 via MFCB. When STE3 is activated, BV3 sends its bit vector to BV2b and BV4 in the Swap step. Meanwhile, BV3 reads the third bit of its bit vector as BV-read of action r(3) in the Read step and sends it to the Active Vector. When BV2b receives the updated bit vector from BV3, it always executes the `shift` action before writing back. Finally, the activation of reporting state STE4 requires the read result of STE3 to be '1' and the incoming character is b. Here, BV4 is deactivated because it has r(3) instruction, while BV2a is partially power-gated and only sends a constant because of its action `set1`. The state transition among all STEs is implemented with the standard state transition in SOTA designs, while the routing among BVs is achieved in MFCB. For illustration purposes, this example reduces the pointer in the instruction set (Table 3) from the actual 6 bits to 2 bits.

6 System Architecture of BVAP

The hierarchical architecture of BVAP, shown in Fig. 8, comprises three levels: bank, array, and tile. Each bank includes *four* arrays and I/O, while each array consists of *sixteen* tiles, which is limited by the size of the global switch used for state transitions beyond a single tile. Because BVs cannot communicate across tiles by design, the maximum upper bound of repetition is limited by the number of BVs in a BVM. Based on the observation that the ratio of BV-STEs is typically below 18% across our benchmarks, we designed 48 BVs for each 256-STE tile, which covers over 99% of regexes in our datasets. The unsupported regexes can be executed via partial unfolding. As such, each BVAP bank supports up to 16,384 STEs, with 3,072 of them being BV-STEs. Each BVM contains 48 BVs and two 48×48 4-port FCBs, functioning as an MFCB. The state matching and state transition modules are adopted from CAMA [16], including a 256×32 8T-SRAM-based CAM and a 128×128 Reduced CrossBar (RCB). Tiles are grouped in pairs to further support reconfiguration between RCB and FCB modes for regexes with different state transition sparsities. In the 32-bit FCB mode, one CAM subarray and one BVM are power-gated, while the two crossbars function as one 128×128 FCB.

To handle varying bit-vector-processing latency across tiles, BVAP has a Global Controller to coordinate behavior throughout the array (see Fig. 9). When BVM of any tile is activated, the Global Controller stalls other tiles within the same array, because the Array Input Buffer is designed to broadcast to all tiles with low bandwidth. To reduce the throughput penalty incurred by the stall, we designed two levels of buffering to partially hide the latency across the array. When multiple BVMs are activated, the Global Controller finds the one with the longest bit-vector-processing latency and sends its tile ID to the input buffer to properly stall input broadcasting. An 8-entry look-up table in the Array Input Buffer stores the maximum bit-vector-processing latency of each tile to assist this process. This dynamic stall

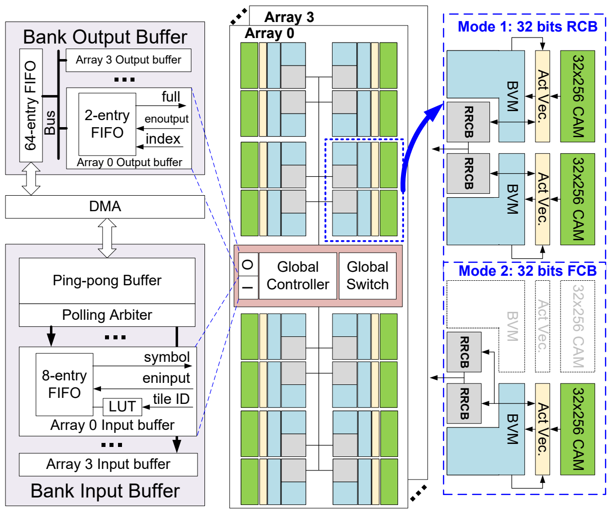


Figure 8. Overview of a hierarchical BVAP bank.

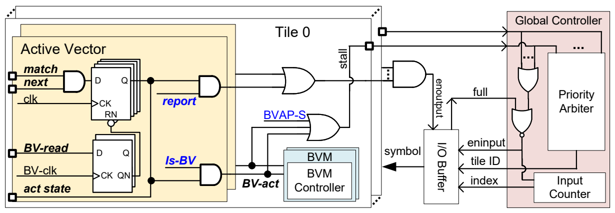


Figure 9. The control logic of a BVAP array.

scheme optimizes the delay penalty to support tiles with varying bit-vector-processing delays while ensuring correctness. The additional control logic and look-up table that enable this scheme take negligible area and energy (< 1%) of a BVAP array, according to our synthesized design.

Input/Output Streaming. The I/O interfaces BVAP and the host system. The hardware configuration is first loaded to BVAP. Then the system transmits streaming data through DMA to the Bank Input Buffer of BVAP for processing. When the host receives matching results from the Bank Output Buffer, it will take further analysis and actions.

To efficiently support asynchronous data processing across multiple arrays, we have designed an I/O buffer hierarchy (Fig. 8). The two-level input buffer accommodates scenarios where BVM in different arrays are activated by different symbols, and the delay of the bit-vector-processing phase is different. The Bank Input Buffer is a 128-entry ping-pong buffer to hide the latency of loading data through DMA. The Bank Input Buffer employs a polling arbiter to process the data requests by the Array Input Buffers. To ensure that Array Input Buffers are never empty, the bandwidth of the Bank Input Buffer and the capacity of the Array Input Buffer must scale linearly with the number of BVAP arrays in a bank. Considering the tradeoff between DMA sharing and

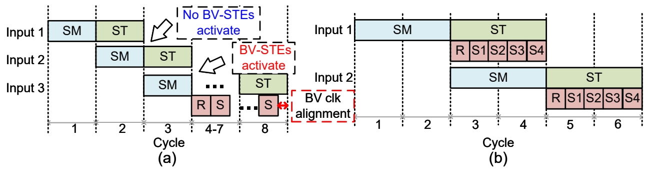


Figure 10. Scheduling of (a) BVAP and (b) BVAP-S. (Abbreviations: SM: state matching, ST: state transition, R: Read step, S: Swap step)

the Array Input Buffer overheads, we chose four arrays in a bank. Under this configuration, each array contains an 8-entry FIFO as its input buffer, which broadcasts one 8-bit input symbol to all tiles when the array is not stalled by bit-vector processing. When the number of symbols in a FIFO is less than four, the FIFO will request new data from the Bank Input Buffer. Upon request, the Bank Input Buffer transmits four symbols at a time to the array initiating the request.

The output buffer is simpler since the match rate is typically lower than 10%. Once a `report` flag is issued by a tile, the counter that monitors the input sequence (in Fig. 9) sends the index to the output 64-entry FIFO. To alleviate congestion when multiple arrays request the bus simultaneously, each array has its own 2-entry FIFO to store report results. In the unlikely event that the output buffer of an array is full, a `full` alert is sent to the Global Controller to stall the array. The final outputs are sent out through DMA when the Bank Output Buffer is full.

Pipeline in BVAP. Fig. 10(a) shows the time schedule of BVAP. Due to the reuse of the Active Vector for state matching, state transition, and BVM, all parts of the BVAP tile can work in parallel. In addition, as the bit-vector-processing phase has smaller step delays compared to state matching and state transition phases, a faster clock, referred to as the Bit Vector clock (BV clk), is assigned to BVM. The result of BVM is aligned with the clock of the system at the end of the bit-vector-processing phase. BVM is activated only when BV-STEs are active, and then the latency is determined by the bit-vector-processing phase. If all BV-STEs remain inactive, BVM is disabled, and the latency is determined by state matching and state transition phase. This event-driven scheme increases the throughput of BVAP.

To support continuous streaming input from sensors, a constant processing throughput is necessary to avoid a huge input buffer. We devise a BVAP-S mode (see Fig. 10(b)), where BVM is activated for every input symbol. As such, the bit-vector-processing phase becomes the critical path. With BVM maintaining the same clock frequency as in standard BVAP mode, the system clock is slowed down. As a result, state matching and state transition modules can save energy by running slower at a lower supply voltage (reduction from 0.9V to 0.65V in our experiment in §8).

7 Compiler Implementation

We offer a regex-to-hardware compiler that enables the high-level programming of the hardware. Our compiler rewrites regexes taking into account two key parameters: the unfolding threshold of bounded repetitions and the virtual bv size (defined in §5).

The rewriting with the use of unfolding is straightforward. The compiler unfolds bounded repetitions when the upper bound is below the unfolding threshold.

Example 7.1. If the unfolding threshold is set to 4, the compiler will rewrite the regex $a(bc)\{2\}d\{1,3\}ef\{2,\}g\{7\}$ to $abcbcdd^?d^?efff^*g\{7\}$, where $bcbc$, $dd^?d^?$, and fff^* are unfolded from $(bc)\{2\}$, $d\{1,3\}$, and $f\{2,\}$ respectively. Notably, $g\{7\}$ is not unfolded as 7 exceeds the threshold.

Since the bit vectors we use have a fixed size, the compiler splits each large bounded repetition into smaller pieces to ensure each piece fits within the bit vector. We illustrate this with examples for a virtual BV size of 64.

Example 7.2. The compiler rewrites the regex $ab\{147\}c$ into $ab\{64\}b\{64\}b\{19\}c$. The sub-regex $b\{147\}$ is equivalent to $b\{64\}b\{64\}b\{19\}$ because $147 = 64 + 64 + 19$.

The regex $ab\{2,114\}c$ is rewritten by the compiler into $ab\{1,64\}b\{1,32\}b\{0,16\}b\{0,2\}c$. The occurrences of bounded repetitions $b\{1,64\}$, $b\{1,32\}$, and $b\{0,16\}$ are supported by the hardware since they can be transformed into states with actions `rAll`, `rHalf` and `rQuarter` respectively. If the unfolding threshold is 4, this regex will be further rewritten as $ab\{1,64\}b\{1,32\}b\{0,16\}b^?b^?c$.

We rewrite $a\{1,100\}$ as $a\{1,64\}a\{0,32\}a^?a^?a^?a^?$, where $a\{1,64\}$ and $a\{0,32\}$ are supported by actions `rAll` and `rHalf` respectively.

Our compiler aims to minimize the number of occurrences of bounded repetitions when rewriting regexes given an upper bound of the size of the bit vector. This strategy reduces the memory needed for mapping bit vectors on hardware.

Compilation Procedure. Our compiler follows a sequence of steps to convert a set of regexes into JSON configuration files for programming the hardware: (1) The compiler first parses the regex and unfolds bounded repetitions whose upper bound is ≤ 2. (2) The compiler analyzes input symbols that occur in regexes and generates an encoding schema for every input symbol. We use a similar encoding algorithm as presented in [16]. (3) Using the user-specified unfolding threshold and the virtual BV size, the compiler applies the previously mentioned rewriting rules. It unfolds regexes and splits large bounded repetitions. (4) After that, the compiler constructs corresponding NBVAs for regexes and transforms them into AH-NBVAs. (5) Finally, the compiler produces a JSON file to describe the generated automata. This file serves as the configuration file for programming the hardware.

Table 4. Circuit models in 28nm.

Type	Size	Energy (pJ)	Delay (ps)	Area (μm^2)	Leakage (μA)
8T SRAM routing switch	128×128	1-14.2	298	5655	57
	256×256	2-55	410	18153	228
8T CAM	32×256	33.56	336	7838	28.5
4-port SRAM routing switch	48×48	0.76-3.25	173	1818	25
Bit Vector	64	1.37	178	17.7	0.56
Global wire	1 mm	0.07	66	50	N/A

8 Experimental Evaluation

In this section, we evaluate the performance of BVAP. We first introduce the datasets we used for benchmarking and explain the setup of our experiments. We then use a micro-benchmark to evaluate the performance of BVAP with different upper bounds of bounded repetition and different match rates. We also perform a parameter sweep to select the best bit vector size and the unfolding threshold for each dataset. Finally, we fix the value of the bit vector size based on the result of the parameter sweep and evaluate the performance of BVAP against CA, eAP, and CAMA.

Datasets. ANMLZoo [44] and AutomataZoo [45] are popular benchmarks for evaluating regex matching and automata processors, including datasets like Snort, ClamAV, and YARA. However, bounded repetitions are unfolded in ANMLZoo and AutomataZoo for simple execution and thus are not suitable for evaluating our work on efficient processing of regexes with bounded repetitions. Therefore, we collected seven datasets from multiple real-world applications, including around 11,000 regexes with non-trivial (maximum upper bound > 4) bounded repetitions with upper repetition bounds exceeding 10,000[1]. These datasets are: (1) the **Snort** [28, 34] and (2) **Suricata** [38] datasets which contain patterns for network traffic, (3) the **Prosite** dataset [29, 32] which includes patterns for detecting protein motifs, (4) the **ClamAV** [9] and (5) **YARA** [43] datasets which contains patterns for identifying the presence of viruses, (6) the **SpamAssassin** dataset [12] which includes patterns for detecting spam email, and (7) the **RegexLib** dataset [27] which is a collection of regexes for describing email addresses, phone numbers, URL, etc.

Experiment Setup. We developed a custom cycle-accurate simulator for BVAP, which can also simulate existing in-memory automata accelerators like CA, eAP, and CAMA. The simulator reads a configuration file generated by the compiler and performs the matching over a sequence of input symbols. The simulator emulates hardware behavior cycle by cycle with the actual dataflow. Meanwhile, we performed consistency checks to verify the functionality of BVAP and the correctness of the hardware simulator by comparing its matching results against a reliable software matcher. Table 4 lists circuit models used in our evaluations, including access

[1]Available at https://kmamouras.github.io/projects/regexes/datasets

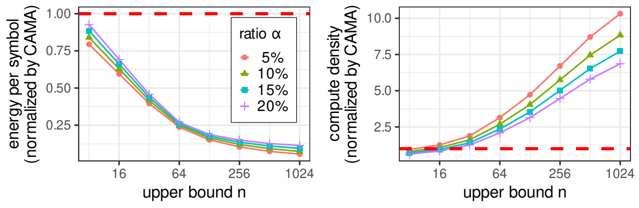


Figure 11. Energy per symbol and compute density of BVAP (normalized by CAMA) across bit-vector activation ratio α and n in $ra\{n\}$.

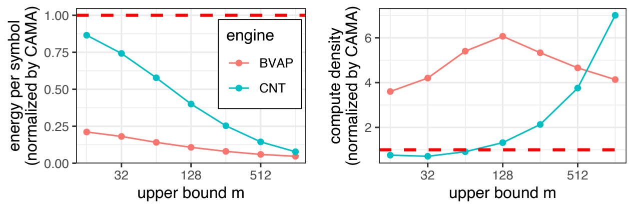


Figure 12. Energy per symbol and compute density of BVAP and CNT (normalized by CAMA) across m in $ra\{64\}b\{m\}$.

energy, delay, and area. These values are derived from SPICE simulations on custom-designed SRAM and CAM arrays in TSMC 28nm CMOS. The values of global wire come from CA [37]. The BVM occupies 4490 μm^2, which is 20% smaller than RRCB. The energy in Table 4 is simulated under various input vector and output vector conditions. The energy of routing switches scales up with both the number of activated wordlines and the number of "1" on OBLs. The global wire estimation is based on the data provided in CA [37]. For a fair comparison, all other automata processor architectures reported in this paper are simulated with the same circuit model and simulator. Since a BVAP tile is 1.5× larger than a CAMA tile, BVAP's global wire delay is estimated to be 39.1ps, which is 50% larger than the 26.1ps global wire delay reported by CAMA [16]. The largest delay of the BVAP pipeline stage is 449.1 ps, which sets its clock frequency to 2 GHz. As explained in §6, BVM runs at a faster clock for higher throughput, which is 5GHz based on Table 4. Note all chosen clock frequencies include a 10% safety margin.

Two key performance metrics we focus on are *energy per symbol* and *compute density*. Energy per symbol measures energy efficiency by dividing the total energy cost by the number of processed input symbols. Compute density is calculated by dividing the processing throughput by the total area, representing the area efficiency of the processor.

Micro-benchmarks. With crafted micro-benchmarks, we illustrate the merits of BVAP without considering mapping multiple regexes to fixed-sized tiles (i.e. we customize the memory size for a single regex). The BV size is fixed to 64.

We first consider the regex $ra\{n\}$, where r is $a \cdot a \cdots a$ (16-fold concatenation of a), to evaluate the impacts of the bit-vector-activation ratio α (representing the proportion of input elements that activate BV-STEs) and the upper bound of the repetition n. We set the length of r to be 16 because, in the analysis of RegexLib, a comprehensive collection of regexes used in various domains, the average number of normal STEs is 16. We build the input sequence by selecting each element from symbols a and b with a certain probability distribution, thereby controlling α. Our evaluation considers α values of 5%, 10%, 15%, and 20% given α is rarely above 10% in real-world applications. Fig. 11 shows that BVAP offers consistently lower energy per symbol (i.e., better energy efficiency) and higher compute density for $n \geq 16$ compared to CAMA. Increasing n improves both metrics because each BV-STE in BVAP replaces more normal STEs, saving both area and state transition energy. Higher α values, on the other hand, worsen the compute density due to more frequent BV-STE activations, reducing throughput and negatively affecting compute density. The energy per symbol also gets slightly worse with higher α because of energy overhead in Bit Vector processing but is consistently better than CAMA.

One may wonder about the effectiveness of only adding counters as an alternative to matching regexes with bounded repetition [10]. In principle, this method faces difficulties in dealing with counter-ambiguous regexes [17], where a counting state of the NCA constructed from the regex carries different counter values (see Fig. 1). Nonetheless, we implemented this solution by adding counters to the CAMA architecture, which we call CNT, and consider a second regex $ra\{64\}b\{m\}$ for varying integer values of m. While the counter element can process $b\{m\}$, $a\{64\}$, being counter-ambiguous, necessitates unfolding to be executed by CNT. In comparing BVAP with CNT and CAMA, Fig. 12 shows that BVAP consistently consumes less energy per symbol than CNT, and achieves higher compute density against CNT for $m \leq 512$.

Design Space Exploration. The bit vector size (`bv_size`) and unfolding threshold (`unfold_th`) are two user-controlled parameters that offer application-specific optimization of BVAP's area, energy, and throughput performance. Larger `bv_size` enables higher regex compression rates, saving area and energy, but at the cost of increased delay during the bit-vector-processing phase. Smaller `unfold_th` allows more BV-STEs with small upper bounds of bounded repetitions to be compiled, potentially leading to underutilization of bit vectors during matching. In this experiment, we use all seven datasets and consider the mapping of NBVAs to the actual hardware implementation. To expedite the simulation, we selectively sampled >300 regexes from each dataset, while keeping a similar distribution of the number of STEs in the subset. The regexes are mapped to BVAP arrays with a greedy mapping algorithm similar to that in [16].

Fig. 13 depicts BVAP compute density and energy delay product (EDP) across `bv_size` and `unfold_th` combinations. We find that the best points of compute density and EDP are not always aligned with the same workload. Therefore, we further define a figure of merit (FoM) for optimization purposes, where FoM = total energy × area/throughput, to evaluate the BVAP with the trade-off between energy-efficiency and compute density. Fig. 13 also depicts BVAP FoM across `bv_size` and `unfold_th` combinations, while Table 5 presents parameters yielding the best FoM.

Table 5. Parameters achieving the best FoM in each dataset. (RL. is the abbreviation of RegexLib. SpamA. is the abbreviation of SpamAssassin.)

dataset	ClamAV	Prosite	RL.	Snort	SpamA.	Suricata	YARA
`bv_size`	64	16	16	64	16	64	64
`unfold_th`	8	4	4	12	12	12	8

Real-World Benchmarks. We evaluated our proposed architecture using real-world inputs collected from applications linked to each dataset, compiling all regexes using optimal parameters from the design space exploration. Given that the optimal BV size found in design space exploration is 64 or less, we set the physical bit vector size to 64, which leads to an upper bound of 3072 for bounded repetitions in regexes, because a tile contains 48 BVs. The virtual bit vector size for each dataset is chosen based on Table 5. The wasted BVM area due to the partial use of BVs was considered in the benchmark evaluation. Fig. 14 summarizes the metrics (area, energy per symbol, power, compute density, throughput, and FoM) of BVAP, BVAP-S, CAMA [16], eAP [31], and CA [37]. Averaging across all benchmarks, BVAP achieves notable reductions in area, power, and energy per symbol over CAMA, CA, and eAP. The average compute density of BVAP is on par with CAMA but is remarkably higher than CA by 134% and eAP by 62%. On Prosite and SpamAssassin datasets, BVAP has lower compute density than CAMA, because most bounded repetitions in Prosite have small upper bounds, and the proportion of BV-STEs in SpamAssassin is only ~ 5%. On other datasets, BVAP's compute density surpasses CAMA by 64% (Snort), 65% (Suricata), 18% (YARA), and 34% (ClamAV). Meanwhile, BVAP achieves similar throughput as CA and eAP but is slower than CAMA by 11.2%. BVAP improves FoM 4.3×, 50×, and 33× compared to CAMA, CA, and eAP.

We also evaluated the BVAP-S mode for streaming input without buffering. Compared to BVAP, BVAP-S saves energy and power by 39% and 79%, thanks to the lower supply voltage for state matching and state transition hardware. But it should also be noted that the throughput and compute density of BVAP-S are lower than BVAP due to its reduced clock frequency. These properties make BVAP-S more appropriate for direct sensor connection in edge computing scenarios.

9 Related Work

Several studies have extended traditional ***automata*** with counters to handle regexes with bounded repetitions. XFA [33]

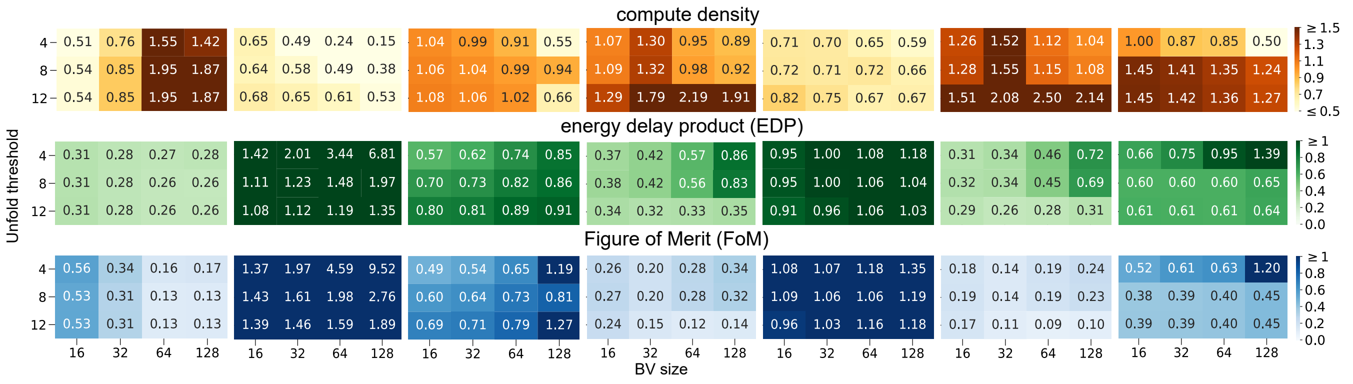


Figure 13. Compute density, EDP, and FoM results of design space exploration in ClamAV, Prosite, RegexLib, Snort, SpamAssasssin, Suricata, and YARA datasets. All values are normalized to that of CAMA.

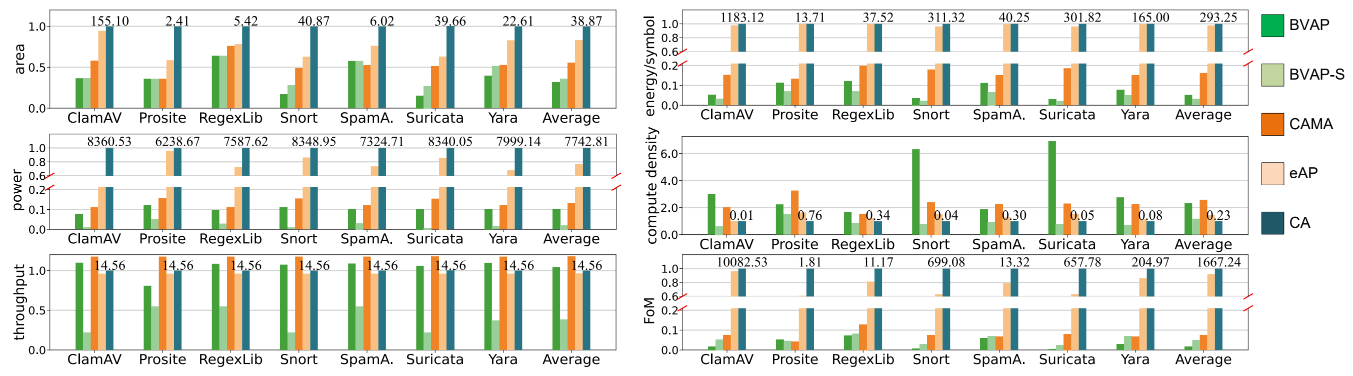


Figure 14. Comparison of the area (mm^2), energy per symbol (nJ/byte), power (Watt), compute density (Gbps/mm^2), throughput (Gbps), and FoM (mJ×mm^2/Gbps) among BVAP, BVAP-S, CAMA, eAP, and CA, all normalized to CA values. The absolute values of CA metrics are annotated in the figures. Lower values denote better performance for area, energy per symbol, power, and FoM, while higher values are better for compute density and throughput. (SpamA. is the abbreviation of SpamAssassin.)

and counting-NFA [2] reduce memory requirements by DFAs and NFAs, respectively. [42] uses counting automata to handle counting, which is then converted into deterministic counting-set automata. [17] employs nondeterministic counter automata (NCAs), using counter ambiguity to separate simple and complex cases of counting. [18] proposes nondeterministic bit vector automata (NBVAs), a convenient alternative to NCAs for specifying regex matching algorithms. This work builds upon NBVAs, presenting a novel algorithm to transform NBVAs into AH-NBVAs for efficient hardware implementations. Lookaround assertions, which can succinctly encode certain kinds of counting, are considered in [24].

Meanwhile, various ***hardware platforms*** have been explored for regular pattern matching. Several works implement efficient regex matching algorithms on GPUs [5, 21, 22, 46, 51]. However, irregular and unpredictable memory access on GPU restricts the achievable energy efficiency and throughput. Various ASIC accelerators have been proposed for regex matching [1, 10, 15–17, 20, 23, 30, 31, 37, 39, 41], but with limited support for bounded repetitions. Moreover, FPGAs have also been studied for regular pattern matching [3, 7, 8, 26, 35, 47, 48, 50], whose performance and scalability are commonly limited by routing congestion.

10 Conclusion

We present BVAP, a software-hardware co-designed Bit Vector Automata Processor for efficient regular pattern matching. It is based on a novel hardware-friendly model AH-NBVA, which extends NFAs with bit vectors. AH-NBVAs facilitate energy- and memory- efficient matching of regexes with the challenging construct of bounded repetition. The BVAP hardware employs a specialized Bit Vector Module to support bit vector processing efficiently. Through cross-stack co-design, BVAP achieves higher efficiency, smaller area, and higher compute density across real-world benchmarks, over state-of-the-art automata accelerators.

Acknowledgments

This work is supported by the National Science Foundation (NSF) under grants No.2313062 and No.2146476.

References

[1] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Efficient text searching of regular expressions. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca, editors, *Automata, Languages and Programming*, pages 46–62, Heidelberg, 1989. Springer.

[2] Michela Becchi and Patrick Crowley. Extending finite automata to efficiently match Perl-compatible regular expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, New York, NY, USA, 2008. ACM.

[3] Joao Bispo, Ioannis Sourdis, Joao M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *2006 IEEE International Conference on Field Programmable Technology*, pages 119–126, USA, 2006. IEEE.

[4] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 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)*, pages 737–748. IEEE, 2018.

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

[6] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):530–543, 2020.

[7] Milan Češka, Vojtech Havlena, Lukáš Holík, Jan Korenek, Ondrej Lengál, Denis Matoušek, Jirí Matoušek, Jakub Semric, and Tomáš Vojnar. Deep packet inspection in FPGAs via approximate nondeterministic automata. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 109–117. IEEE, 2019.

[8] Jian Chen, Xiaoyu Zhang, Tao Wang, Ying Zhang, Tao Chen, Jiajun Chen, Mingxu Xie, and Qiang Liu. Fidas: Fortifying the cloud via comprehensive FPGA-based offloading for intrusion detection: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1029–1041, New York, NY, USA, 2022. Association for Computing Machinery.

[9] ClamAV. ClamAV - open source antivirus engine. Available at https://www.clamav.net/, 2023. [Online; Accessed 17 July, 2023].

[10] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[11] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Counter machines and counter languages. *Mathematical Systems Theory*, 2(3):265–283, 1968.

[12] Apache Software Foundation. Apache Spamassassin. Available at https://spamassassin.apache.org/, 2022. [Online; Accessed 17 July, 2023].

[13] Wouter Gelade, Marc Gyssens, and Wim Martens. Regular expressions with counting: Weak versus strong determinism. In *Mathematical Foundations of Computer Science 2009*, pages 369–381, Heidelberg, 2009. Springer.

[14] Victor Mikhaylovich Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16(5):1–53, 1961.

[15] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. HARE: Hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[16] Yi Huang, Zhiyu Chen, Dai Li, and Kaiyuan Yang. CAMA: Energy and memory efficient automata processing in content-addressable memories. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 25–37, New York, NY, USA, 2022. IEEE.

[17] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan 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*, PLDI 2022, pages 733–748, New York, NY, USA, 2022. ACM.

[18] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. Regular expression matching using bit vector automata. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 2023.

[19] Marzieh Lenjani and Mahmoud Reza Hashemi. Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities. *IET Computers & Digital Techniques*, 8(1):30–48, 2014.

[20] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. Architectural support for efficient large-scale automata processing. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 908–920, New York, NY, USA, 2018. IEEE.

[21] Hongyuan Liu, Sreepathi Pai, and Adwait 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*, ASPLOS '20, pages 251–265, New York, NY, USA, 2020. ACM.

[22] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. Asynchronous automata processing on GPUs. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(1), mar 2023.

[23] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 461–472, New York, NY, USA, 2012. IEEE.

[24] Konstantinos Mamouras and Agnishom Chattopadhyay. Efficient matching of regular expressions with lookaround assertions. *Proceedings of the ACM on Programming Languages*, 8(POPL), 2024.

[25] Pcre syntax. Available at https://www.pcre.org/original/doc/html/pcrepattern.html, 2023. [Online; Accessed 18 July, 2023].

[26] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. 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)*, pages 138–147, New York, NY, USA, May 2020. IEEE.

[27] RegexLib. Regular expression Library. Available at https://regexlib.com/, 2023. [Online; Accessed 17 July, 2023].

[28] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, page 229–238, USA, 1999. USENIX Association.

[29] Indranil Roy and Srinivas Aluru. Discovering motifs in biological sequences using the Micron Automata Processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 13(1):99–111, 2016.

[30] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 86–98, 2020.

[31] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 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, pages 87—-99, New York, NY, USA, 2019. ACM.

[32] Christian J. A. Sigrist, Lorenzo Cerutti, Edouard de Castro, Petra S. Langendijk-Genevaux, Virginie Bulliard, Amos Bairoch, and Nicolas Hulo. PROSITE, a protein domain database for functional characterization and annotation. *Nucleic Acids Research*, 38(suppl_1):D161–D166,

2009.

[33] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster signature matching with extended automata. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, page 187–201, USA, 2008. IEEE Computer Society.

[34] Snort. Snort - network intrusion detection & prevention system. Available at https://www.snort.org/, 2023. [Online; Accessed 17 July, 2023].

[35] Ioannis Sourdis, Joao Bispo, Joao MP Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2008.

[36] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[37] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 259–272, New York, NY, USA, 2017. ACM.

[38] Suricata. Suricata - open source intrusion detection and prevention engine. Available at https://suricata.io/, 2023. [Online; Accessed 17 July, 2023].

[39] Prateek Tandon, Faissal M Sleiman, Michael J Cafarella, and Thomas F Wenisch. HAWK: Hardware support for unstructured log processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 469–480, New York, NY, USA, 2016. IEEE.

[40] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[41] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM 2004*, volume 4, pages 2628–2639 vol.4, New York, NY, USA, 2004. IEEE.

[42] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. Regex matching with counting-set automata. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[43] VirusTotal. YARA: The pattern matching swiss knife for malware researchers. Available at https://virustotal.github.io/yara/, 2023. [Online; Accessed 17 July, 2023].

[44] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2016.

[45] Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, and Kevin Skadron. AutomataZoo: A modern automata processing benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 13–24, New York, NY, USA, 2018. IEEE.

[46] Yuguang Wang, Robbie Watling, Junqiao Qiu, and Zhenlin Wang. GSpecPal: Speculation-centric finite state machine parallelization on GPUs. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 481–491, New York, NY, USA, 2022. IEEE.

[47] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. REAPR: Reconfigurable engine for automata processing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, New York, NY, USA, 2017. IEEE.

[48] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. 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, page 30–39, New York, NY, USA, 2008. ACM.

[49] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. 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, pages 93–102, New York, NY, USA, 2006. ACM.

[50] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, November 2020.

[51] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng 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*, PPoPP '12, page 129–140, New York, NY, USA, 2012. ACM.