RAP: Reconfigurable Automata Processor

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

Alexis Le Glaunec*
Rice University
Houston, Texas, USA
alexis.leglaunec@rice.edu

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

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

Abstract

Regular pattern matching is essential for applications such as text processing, malware detection, network security, and bioinformatics. Recent in-memory automata processors have significantly advanced the energy and memory efficiency over conventional computing platforms. However, these processors are typically optimized only for one type of automata, limiting their capability to efficiently support regex processing under diverse real-world workloads. This paper presents RAP, the first reconfigurable in-memory automata processor for efficient regular pattern matching across diverse workloads. It supports Nondeterministic Finite Automata (NFA), Nondeterministic Bit Vector Automata (NBVA), and Linear NFA (LNFA) through reconfigurable architecture and circuit designs, and a compiler for translation. RAP is evaluated in 28nm CMOS PDK, achieving 1.2-1.5× higher energy efficiency and 1.3-2.5× higher compute density compared to SotA automata processors for NFA (CA and CAMA) over diverse real-world benchmarks. It also achieves 1.6× higher compute density and similar energy efficiency as BVAP, a SotA optimized for bounded repetitions. Finally, RAP is >100× and >1000× more energy efficient than SotA GPU and CPU solutions.

CCS Concepts

• **Theory of computation** → **Regular languages**; • **Hardware** → **Hardware accelerators**; • **Computer systems organization**;

Keywords

Automata Processor, Nondeterministic Bit Vector Automata, Linear Nondeterministic Finite Automata, Reconfigurability

ACM Reference Format:
Ziyuan Wen, Alexis Le Glaunec, Konstantinos Mamouras, and Kaiyuan Yang. 2025. RAP: Reconfigurable Automata Processor. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731106

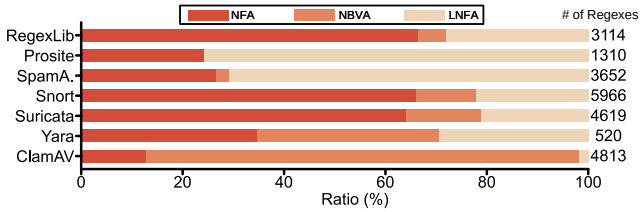


Figure 1: The proportions of regexes that can be represented by NFA, NBVA, and LNFA models in seven benchmarks.

1 Introduction

Regex matching is a core computational primitive in the application domains of network security [54], data mining [38], and bioinformatics [4, 34]. For example, to monitor the traffic on a 10 Gb/s high-speed network with a single flow, Snort network intrusion detection systems used more than 57% of CPU resources [17] while regex matching takes up to 90% of the time in network monitoring [54]. To enable network monitoring on edge devices, higher energy and area efficiency are demanded due to their restricted power budgets. However, existing solutions on edge require over 2 mW when using a transceiver with a data rate of 250 kbps [1]. For the emerging bioinformatic applications, the latest sequencing machine produces results at over 10GB/s, the speed and cost of which are exponentially improving [11]. To meet the growing demand for regex matching in clouds and edges, it is essential to improve the energy/area efficiency and throughput of the processing system.

Most specialized hardware accelerators [10, 18, 36, 37, 41] perform regex matching with nondeterministic finite automata (NFAs) to take advantage of the inherent parallelism of hardware and the memory efficiency of NFAs. They encode the NFA character classes (CCs) inside the memory and use a crossbar switch to realize the transfer function. A downside of this approach is that it does not take advantage of the redundancy of CCs or the high sparsity in the switch, thus wasting energy and memory resources. Some accelerators support approximate matching for specific pattern matching applications, such as DNA classification [13, 14, 16] and local sensitive hashing [27, 32]. However, these methods lack the expressiveness of regex. Most FPGA implementations are also based on NFA simulation [5, 7, 8, 30, 35, 49, 53, 56], but their throughput is limited by routing congestion. GPU-based accelerators [2, 6, 25, 26, 44, 48, 55, 57] are also based on NFA simulation algorithms, which can give rise to irregular memory accesses and thus lower throughput.

Classical regular expressions can be translated into NFAs with a number of states linear in the size of the expression. A common feature in regexes is the $r\{m,n\}$ construct, called bounded repetition, that is used to describe the repetition between m and n times of the pattern r. To execute such regexes using an NFA, we need to unfold the bounded repetition. For instance, $r\{n,n\}$ is unfolded into $r^n = r \ldots r$ where r is repeated n times. The unfolding of $r\{m,n\}$ increases the size of the pattern by a factor $\Theta(n)$. Instead of unfolding, previous work on CPU [22] and in ASIC [20, 52] use an automaton called nondeterministic bit vector automaton (NBVA)

*These authors contributed equally to this research.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731106

to support regexes with bounded repetitions efficiently. In NBVAs, control states can have bit vectors to keep track of the number of repetitions of the bounded repetitions. These ASICs use additional counter or bit vector modules to efficiently support regexes with bounded repetitions, whereas a significant number of regexes do not contain bounded repetitions. They also do not take advantage of the high sparsity of the routing switch for NFAs that have a linear structure $q_0, q_1, \ldots, q_{n-1}$ where each transition is from a state q_i to a neighboring state q_{i+1}. We call them Linear NFAs (LNFA). It is known that those regexes can be executed efficiently with the Shift-And [3] algorithm, which is a bit-parallel algorithm with very high throughput that is used in the software matcher Hyperscan [51], as well as in the GPU matcher of HybridSA [23].

By analyzing the compositions of seven benchmarks from real-world applications in Fig. 1, we have found that, for different benchmarks, the proportion of regexes that can be simulated with NBVAs, LNFAs, and NFAs varies tremendously. In the RegexLib dataset, most regexes cannot be simulated with NBVAs or LNFAs and are simulated with NFAs instead. In contrast, more than 80% regexes in the ClamAV dataset have bounded repetitions and can be efficiently supported with NBVAs. For the Prosite and SpamAssassin datasets, the majority of regexes can be translated into LNFAs. These observations motivate a reconfigurable hardware solution that efficiently supports different automata models for diverse workloads.

To efficiently support various types of regexes for different applications, we present the first-of-its-kind Reconfigurable Automata Processor (RAP), which effectively processes various regex classes while maintaining minimal controller overhead compared to hardware specialized for a single regex class. While add-on modules to support bit vector and/or counter in regexes with large bounded repetitions could drastically reduce area and energy [20, 52], these dedicated add-ons suffer from low flexibility and underutilization facing diverse workloads. Since all components needed to simulate NFA, NBVA, and LNFA can be stored or encoded within 8T-SRAMs that dominate the chip area (76% in), we reuse the 8T-SRAM with different control flows to efficiently support all three modes. Specifically, RAP (1) unifies the storage of CCs and bit vectors, (2) dynamically allocates hardware resources based on workload demands, and (3) integrates the transfer function encoding scheme and bit vectors processing schemes in local switches. With these optimizations, NBVA processing in RAP takes 73% lower energy and 75% smaller area compared to baseline NFA processing.

To support the other common regex model, LNFA, we exploit the Shift-And algorithm [3] well-suited for LNFA's linear transfer functions, and its efficient implementation using the same in-memory fabric. The transfer function is implemented by a repurposed vector rather than a crossbar to reduce area and energy. We fully utilize the CAM and local switches to store CCs and suggested a binning method to process multiple LNFA together to further reduce energy. As such, LNFA processing in RAP achieves 79% lower energy consumption compared to processing as NFA.

In summary, our ***main contributions*** are the following:

- We designed RAP, the first reconfigurable automata processor that supports NFA, NBVA, and LNFA models with little overhead, leading to high energy efficiency and low memory usage for all types of regex matching workloads.
- We realized unified storage for CCs and bit vectors and devised an encoding scheme to process bit vectors in local switches to handle NBVA with RAP design.
- We implemented LNFA with the Shift-And algorithm to compress its transfer function and propose a binning method to decrease the energy consumption of LNFA further.
- We developed a regex-to-hardware compiler for high-level programming of the hardware. This compiler chooses the best automata model for each regex and programs it on RAP.
- RAP is evaluated with a 28nm CMOS process across seven real-world benchmarks. RAP improves compute density, measured as throughput per unit area, by 1.6× compared to BVAP while maintaining a similar energy efficiency. It also enhances compute density by 1.3×, and 2.5× over CAMA, and CA, while offering 1.5× and 1.2× higher energy efficiency. RAP achieves >100× and >1000× greater energy efficiency compared to GPU and CPU solutions. It also improves throughput by 11× over SotA FPGA solutions.

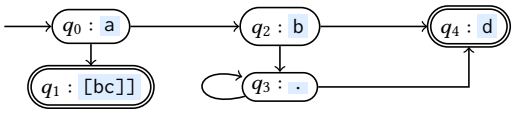
2 Preliminaries

2.1 Automata Theory

Regexes are a widely used formalism for describing regular patterns. For a finite alphabet Σ, regexes over Σ are given by the grammar $r ::= \varepsilon \mid \sigma \mid (r|r) \mid r \cdot r \mid r^* \mid r\{m,n\}$, where $\sigma \subseteq \Sigma$ is a predicate over the alphabet and m, n are natural numbers with $m \leq n$. We also use the term *character class* to refer to a predicate over Σ. The predicate Σ contains all symbols in the alphabet, which corresponds to the notation `.` in PCRE-style syntax [29]. 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. The expression $r\{m,n\}$ is called a *bounded repetition* and describes the repetition of r from m to n times. We write $r\{m\}$ for $r\{m,m\}$. The pattern $r\{m,n\}$ can be translated using concatenation and ? but is exponentially more succinct. 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 [43] or Glushkov [15]. 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 ***homogeneous NFA*** 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 labeling 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*, where $\mathcal{P}$ is the powerset operation.

Example 2.1. Consider the regex $r =$ `a([bc]|b.*d)`. The following homogeneous automaton recognizes the language of r.

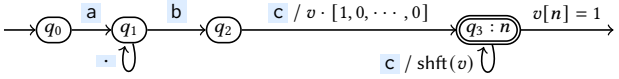


The automaton contains 5 states: q_0, q_1, q_2, q_3 and q_4, where q_0 is the initial state (represented with an incoming edge) and states q_1 and q_4 are final states (represented with a double circle). As the automaton is homogeneous, we label the states instead of the transitions. For instance, q_4 : `d` means that all incoming edges to state q_4 (from q_1 and q_3) are labeled with character `d`.

***Nondeterministic bit vector automata* (NBVAs)** [20, 22] extend NFA with bit vectors (which correspond to sets of counter values in the closely related model of nondeterministic counter automata or NCAs). In an NBVA, a computation involves not only transitions between control states but also the use of a finite number of registers that hold nonnegative integers. NBVA is a natural execution model for regexes with bounded repetitions. The configuration of the NBVA specifies for each control state q a bit vector to represent the set of counter values that are in the control state q.

An NBVA is a tuple (Q, w, Δ, I, F), where Q is a finite set of *(control) states*, and $w : Q \to \{1, 2, \ldots\}$ is a function that maps each state to a strictly positive integer, corresponding to the counter values for a given state. The *transition relation* Δ contains finitely many transitions of the form (p, σ, q, θ), where p is the source state, $\sigma \subseteq \Sigma$ is a predicate over the alphabet (i.e., a character class), q is the destination state, and $\theta : \mathbb{B}^{w(p)} \to \mathbb{B}^{w(q)}$ is the update function for the bit vector. 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 $v \in \mathbb{B}^{w(q)}$.

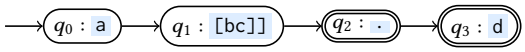
Example 2.2. Consider the regex $r =$ `a.*bc{n}` where $n > 1$. The following NBVA recognizes the language of r.



The NBVA has 4 states: q_0, q_1, q_2 and q_3. We write "$q_3 : n$" to indicate that $w(q_3) = n$, i.e., q_2 has a bit vector of size n. States q_0, q_1 and q_2 are not annotated because they do not carry a bit vector. We annotate each transition $p \to q$ with an expression of the form σ / θ, where σ is a predicate over Σ, and θ is a function used to compute the bit vector of q from the bit vector of p. We use v to represent the bit vector of p. In more details, we write $[1, 0, \ldots, 0]$ to denote the bit vector that is zero everywhere, except for index 0 that is set to 1. We write "$v[n] = 1$" to denote the function checking that the n-th bit of the bit vector v is set to 1. We denote as "shft(v)" the shift left by one operation over a bit vector v, e.g., $\text{shft}([1, 0, 1, 0]) = [0, 1, 0, 1]$.

***Linear NFA* (LNFA)** is a class of homogeneous NFAs where the states can be placed in order $q_0 \ldots q_n$ in a line and each transition is from state q_i to state q_{i+1}. We call them LNFAs because the automata are structured like a line.

Example 2.3. Consider the regex $r =$ `a[bc].d?`. The following LNFA recognizes the language of r.



The LNFA has 4 states: q_0, q_1, q_2 and q_3. Each state is labeled with a character class because an LNFA is also a homogeneous automaton. Note that all the transitions are between states q_i and their neighboring state $q_{i+1} : q_0 \to q_1, q_1 \to q_2$ and $q_2 \to q_3$.

Input		a	b	c
next	0000	0001	0011	0101
labels	0000	0001	0010	0100
states	0000	0001	0010	0100
output	0	0	0	1

Figure 2: Shift-And execution of LNFA for `a[bc].d?`

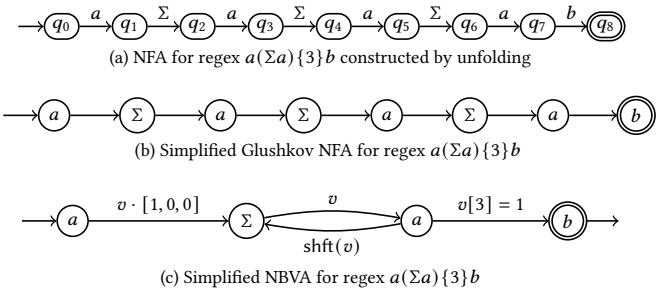


Figure 3: Automata constructed from the regex $a(\Sigma a)\{3\}b$ (`a(.a){3}b` in PCRE syntax).

This structure enables a specialized bit-parallel algorithm, called **Shift-And** [3], to accelerate the simulation of LNFA using efficient bitwise operations. It consists of two steps: (1) the preprocessing of the bit vector masks, and (2) the execution of the automaton with shift-left, bitwise AND and bitwise OR instructions. We will use the notation $x_{n-1}\, x_{n-2}\, \ldots\, x_2\, x_1\, x_0$ for an n-bit vector, where the leftmost bit is the most significant and the rightmost bit is the least significant. We often refer to bit vectors as bit masks in the context of Shift-And, because they are used as OR masks or as AND masks.

Fig. 2 shows the execution of Shift-And for the LNFA `a[bc].d?`. The initial (resp. final) mask is *maskInitial* = `0001` (resp. *maskFinal* = `1100`) and encodes the positions of the initial (resp. final) states. In Shift-And, we keep character masks *labels* where *labels*[`c`] indicates the positions of states whose character class matches letter `c`. We encode the set of active states (resp. next active states) in the *states* (resp. *next*) bit vector where the ith bit (starting from the right) set to "1" indicates that state q_i is active. Now, we go through the execution of Shift-And for LNFA `a[bc].d?` over input `abc`. After consuming the character `a`, we start by performing the transitions with *next* = (*states* $\ll$ 1) OR *maskInitial* = (`0000` $\ll$ 1)OR `0001` = `0001`. Then, we compute the active states that match letter `a` with *states* = *next* AND *labels*[`a`] = (`0001` AND `0001`) = `0001`. This means that, after consuming letter `a`, only state q_0 is active. Then, we check if there is a match with the boolean test ((*states* AND *maskFinal*) $\neq$ `0000`), which is equal to false. So, there is no match. We perform the same steps for the next letters `b` and `c`, and find a match after reading letter `c` because state q_2 is active and is a final state.

2.2 In-memory Automata Processor

NFA Execution. SotA in-memory automata processors, such as AP [10], CA [41], and CAMA [18], execute NFA through two in-memory operations: **state-matching** and **state-transition**. To

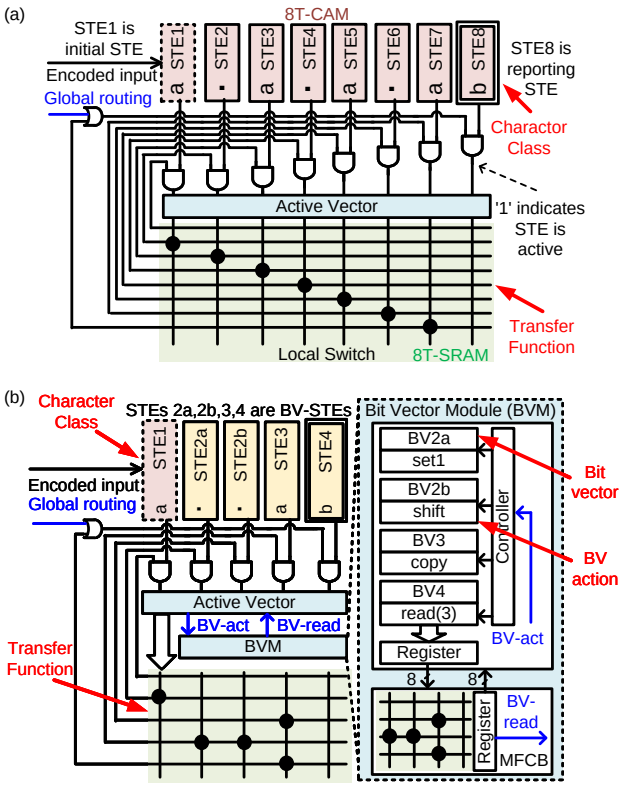


Figure 4: Conceptual diagrams of (a) CAMA and (b) BVAP; and their configurations to match `a(.a){3}b`.

explain their principles, consider matching the regex `a(.a){3}b`, which can be represented with NFA, NBVA, and LNFA models shown in Fig. 3. To construct a basic NFA for this regex, `a(.a){3}b` is expanded into `a.a.a.ab`. This NFA is also an LNFA by definition. The Glushkov construction in Fig. 3(b) ensures that all transitions entering a state are labeled with the same predicate.

Fig. 4(a) illustrates how CAMA, a recent automata processor using CAMs for efficient state matching and the baseline of this work, executes NFA matching. Same as other in-memory automata processors, CAMA uses state-transition elements (STEs) to represent states in Fig 3(b). Each STE has two components: a character class (CC) and a transfer function. CCs are stored in the CAM, and the local switches encode the transfer function. If an STE (e.g., STE1 in Fig. 4(a)) connects to another STE (e.g., STE2), a dot with the value of '1' will be programmed to the corresponding crossing point. During each processing cycle, the state-matching phase compares the input character with all CCs, producing a match result stored in the active vector. In the following state-transition phase, the active vector passes through the transfer function in the switch network, where a logic OR aggregation is naturally performed on each row. In case the number of STEs within a tile is insufficient, global routing switches are included to support larger NFAs, the values of which are merged with local routing results after aggregation. If the aggregation result is '1', the corresponding STE will become available in the next cycle. We call a STE active if it is matched by an input element and is in an available state. STE1 in Fig. 4(a) is an initial STE that is always available for all input. STE8 in Fig. 4(a) is a reporting STE, which will report a match if it is activated.

Although NFAs can support all types of regexes in theory, unfolding bounded repetition exponentially increases the number of STEs and the size of routing switches, incurring significant memory and energy costs. Meanwhile, AP-style in-memory automata processors cannot fully exploit LNFA features, e.g., the compressed routing switch RCB in [37] utilizes less than 5% of switches on LNFAs.

Efficient Support of Bounded Repetitions. BVAP [52] is a recent accelerator that efficiently supports bounded repetitions using NBVA model. To update the bit vector of a state in NBVA mode, we manipulate the bits of the bit vector with actions such as "$\mathrm{shft}(v)$" (See Section 2.1). We call it a BV action. Fig. 4(b) shows the conceptual diagram of BVAP. We refer to any STE that carries a bit vector as a BV-STE. A BV-STE extends a standard STE in AP-style processors with bit vector storage and a programmable BV action. BVAP realizes these actions within an add-on module to CAMA called Bit Vector Module (BVM). BVM includes a semi-parallel multi-bit routing switch (MFCB). In Fig. 4(b), STEs 2a, 2b, 3, and 4 are BV-STEs, which are activated by the BV-act signal from the active vector. BVAP adds an event-driven **bit-vector-processing** phase to standard state-matching and state-transition phases in CAMA or other AP-style designs. During this phase, BVM performs BV reading, routing, and action in a three-stage pipeline. The read results generated in the bit-vector-processing phase, BV-read, are fed back to the active vector, deactivating STEs with read failure.

BVAP was developed to support regexes with large bounds efficiently. In our `a(.a){3}b` example, BVAP only needs $O(1)$ STEs. In BVAP, the size of the BV and the number of BVs inside the BVM are constant, which severely limits its flexibility and utilization under diverse tasks. Besides, even though BVM is worthy of its area in dealing with regexes with large bounds, it is wasted when running basic NFAs. Thus, deciding the number of BVs and BVMs on chip faces a critical tradeoff between the maximum supported number of BV-STEs and the compute density in diverse benchmarks.

3 Hardware Design of RAP

We design RAP to flexibly support NFA, NBVA, and LNFA models by locally reconfiguring a tile of AP-style hardware with little overhead. We adopt CAMA [18] for basic NFA processing and propose new techniques to realize efficient reconfiguration for NBVA and LNFA models. To support the NBVA model, we first design a unified storage for both Bit Vectors (BVs) and CCs and a novel encoding scheme for BV actions. Second, we illustrate the LNFA implementation using the Shift-And algorithm, which is improved by multi-LNFA binning. In the end, we present the system architecture of RAP. Overall, RAP can dynamically allocate hardware resources to NFA, NBVA, and LNFA automata models through reconfiguration of CAMs and local switches, efficiently adapting to diverse workloads in real-world applications. It compresses bounded repetition and linear transfer functions to save area and energy.

3.1 NBVA Mode

Unified Memory for Reconfigurable Modes. It has been shown that 8T-SRAM can be repurposed as 8T-CAM for pattern matching

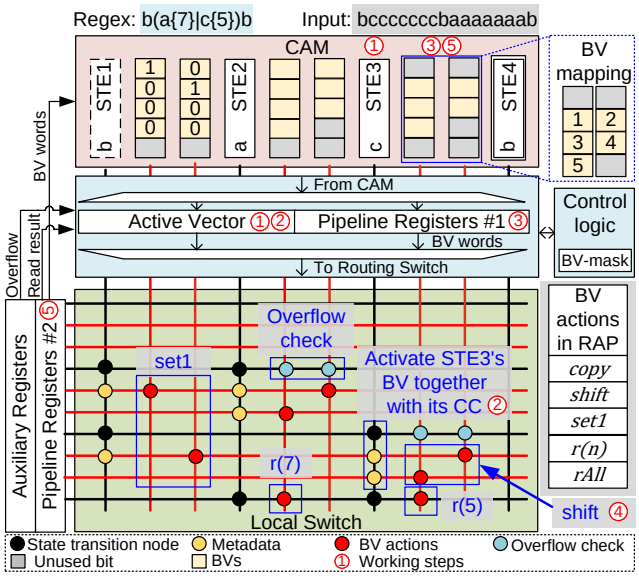


Figure 5: Illustration of NBVA mode in RAP with an example of matching regex `b(a{7}|c{5})b`. Circles in the local switch mean the bits store '1'. The black and red lines indicate transfer function encoding and BV actions encoding respectively.

through the encoding of data [24]. CAMA [18] exploits this circuit technique to utilize the repurposed 8T-CAM for state matching, where CCs of the regex are encoded and stored inside the CAM. In addition, 8T-SRAM can function as BVs and execute actions required by the NBVA model as well [52]. Therefore, both CCs and BVs can be stored within the same 8T-CAM. The 8T-CAM works as either CAM or BVs, according to the types of data being processed. Because the CAM size is typically much larger than the size of the BVs, we can dynamically allocate space to store multiple BVs in a single CAM for varying workloads.

Dynamic allocation of columns of CAM to store either CCs or BVs is shown in Fig. 5. For each BV, we choose the number of rows utilized for the storage of bit vectors, which is called the depth of the BV. The depth of BV determines the compression rate of the bounded repetitions and the latency of the bit-vector-processing phase. With a selected depth, we use minimal contiguous columns of CAM that fit the size of the bit vector, which is called the width. In Fig. 5, we assign two columns to the BVs of STE2, so its width is 2. To support various widths of BVs, we use a BV-mask, a bitmap that designates the storage type of each CAM column that is set during deployment. We use a row-first mapping method to map BVs to a CAM, as shown in Fig. 5. It divides a bit vector into multiple BV words to facilitate semi-parallel routing. If a BV does not require all rows assigned by its depth, it is aligned to the bottom of the BVs. Additionally, the last BV word can be partially used as well. This unified storage mechanism allows RAP to support BVs of varying sizes and numbers within a tile, dynamically adapting to specific workloads by reconfiguring the CAMs. By compressing bounded repetitions into bit vectors, we decrease storage for identical CCs in NFA mode and save energy during the state-matching phase.

BV actions Encoding Scheme. For RAP, BVs are stored inside the CAM, allowing the corresponding columns of the local switch to be repurposed for functions other than transfer function encoding. RAP reconfigures columns corresponding to the storage of BVs for bit vector routing and BV actions. BVs are interconnected by local switches through columns corresponding to BV storage. The number of these columns matches the width of the BV, enabling semi-parallel routing. The cross-point region in the local switch, formed by BV connections, creates a matrix where BV actions can be encoded using an alternative scheme, enabling reconfigurability. In addition, different BV actions can be encoded in distinct regions formed by BV connections to implement NBVA. The BV actions supported and an example of the encoding scheme of BV actions are shown in Fig. 5. For *copy*, the diagonal nodes of the cross-point region are set to '1' so that the BV word is routed identically to its destinations. For *shift*, the last bit of a BV word is replaced by auxiliary registers, which are then updated with the last bit of the BV word. Next, the last bit of the BV word is routed to the position of the first bit, while the remaining bits are right-shifted by one through local switches. For *set1* action, RAP needs to store an initial vector in one column for each connected BV-STE because their first-bit positions may differ due to alignment. Initial vectors are directed to the first column of connected BVs during routing, setting their first bit to '1'. For read actions, the read result is combined with the active vector to deactivate STEs with read failures.

To prevent the activation of BVs when they overflow, we incorporate an overflow checker in the BV actions encoding scheme. All BV words perform bitwise-OR operations through BV routing to detect if bit '1' exists in the bit vector. The intermediate overflow check values are stored in the auxiliary registers. After checking the last BV word, overflow is detected if RAP finds all bits in the BV are '0'. Then, we reset the corresponding bit of active vector to '0' to avoid unnecessary activation of BV-STEs and improve throughput.

Pipelining in Bit-Vector-Processing Phase. RAP performs state-matching and state-transition in each cycle, but only activates CAM columns containing CCs. When BV-STEs are triggered, RAP enters the bit-vector-processing phase, and CAM works as BVs. The pipeline of the bit-vector-processing phase is adopted from BVAP [52], as illustrated in Section 2. First, a BV-word of each BV is read and sent to the local switch. Second, the BV-word performs various BV actions along the routing through the switch, and all the BV-words forwarded to the same BV-STEs are aggregated. Finally, the BV-words are sent back to CAM to update the BVs. This process is repeated for a number of cycles equal to the depth to update all BV-words. To simplify the control and reduce overhead, RAP introduces several improvements.

To activate the bit-vector-processing phase, we need to identify BV-STEs through metadata. To reduce storage, metadata about the size of the BVs and the association between BVs and CCs is encoded in the unused bit within the local switches. If an STE connects to BV-STEs, we place '1' at the cross-points formed by the CC of the source STE and the BV of the destination STE. In this way, the BV is activated whenever a BV-STE is activated along with its CC. After generating the active vector, RAP uses the BV-mask to check whether columns storing BVs are activated. If any BV is activated, RAP starts the bit-vector-processing phase.

For the action *set1*, we only need to store and route one column of initial vectors for each BV. To reset the remaining BV columns to zero, BV-STE with action *set1* only activates the first column of the destination BVs during state transition. Then, the remaining columns are automatically reset to '0' along with inactive BVs. As such, the energy and area for the BV-STEs is reduced with the *set1* action compared to storing and routing the whole bit vector.

In RAP, the BVs of a tile are stored in one CAM, so they must execute the same action in each cycle. For this reason, the BVs in a tile must have the same read action and depth to minimize latency overhead. For $r(n)$ action, the CAM reads the bottom BV words and selects the bit for the read result via local switches. For *rAll* action, the CAM reads all words of the BVs and performs the bitwise-OR operation on these words through RBLs. The latency of the bit-vector-processing phase for BVs within the same tile is identical since the depths of BVs are uniform.

Example 3.1. Fig. 5 shows a working instance of RAP for matching `b(a{7}|c{5})b` in NBVA mode. The depth of BVs is set to 4, and the read actions are $r(7)$ and $r(5)$. There are 4 STEs in total. STE4 is a standard STE, while the other three STEs are BV-STEs. Each Bit Vector occupies two columns based on its size and the selected depth, so STE2 and STE3 each need three columns. The BV of STE3 aligns its first bit to the second row because it only occupies three rows in total. The actions of each BV-STE are programmed in the local switches. STE1 is activated upon matching an input `b`. Since STE1 triggers STE2 and STE3, it stores two initial vectors within two columns. In the bit-vector-processing phase, two initial vectors are sent to STE2 and STE3, and their first bit is set to '1'. Then five consecutive `c` characters are consumed, and each of them works in a similar behavior. In step 1, input `c` matches the CC of STE3 via CAM as state matching. In step 2, the active vector is sent to the local switch for state transition. STE3, activated by the previous `c`, transits itself along with the corresponding BV to the active states. Then, RAP performs a bitwise-AND operation on the result of state-matching and state-transition and updates the active vector. Meanwhile, the local controller detects that STE3, which is associated with a BV, is active, prompting RAP to enter the bit-vector-processing phase. In step 3, a BV-word is read from the BV of STE3 and stored in the pipeline registers. In step 4, we generate the shifted word with a *shift* action and update the word through the local switch. More precisely, the first bit of the word is sent from the 8^{th} column to the 9^{th} row, which is the second bit of the shifted word. The second bit of the word is sent from the 9^{th} column to the 8^{th} row as the first bit of the shifted word. Then, the updated word after the local switch is the input word shifted after performing the *shift* action. In step 5, we update the BV with the updated word. Then, RAP repeats steps 3 to 5 until all of the BV words are updated, before processing the next input symbol. Overall, the BV of STE3 is shifted by 5 bits. Upon receiving the sixth `c`, STE3 is shifted once more, and the BV is full of '0's. At this point, the overflow check confirms that STE3 has overflowed and deactivates it for the next input. Consequently, when the 7^{th} `c` is received, the bit-vector-processing phase has not started, as STE3 is now inactive. Then, input `baaaaaa` is fed to RAP, which fills the BV of STE2. Upon receiving the 7^{th} character *a*, RAP reads the 4^{th} word of the BV. The 7^{th} bit of STE2's BV is routed to STE4, making

Shift-And Execution

Cycle	1	2	3
Input	a	b	c
next	100	110	101
labels	110	011	011
states	100	010	001
output	0	0	1

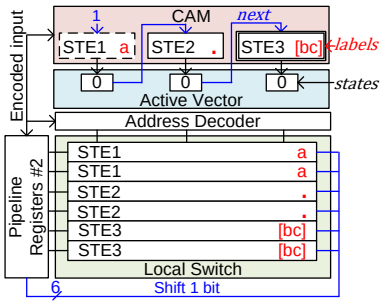


Figure 6: Example of LNFA module for `a.[bc]` in RAP using CAM and a local switch.

it available for the next input character. Finally, the reporting state STE4 is activated after reading the input character `b`, and a match report is generated. For illustration purposes, this example reduces the rows of the CAM from 32 to 5.

3.2 LNFA Mode

Shift-And Matching. The structure of LNFA only allows transition between states q_i to its neighbor state q_{i+1}, which translates into a very sparse local switch full of ones on the diagonal and zeros everywhere else. Thus, we designed a modified version of the Shift-And algorithm on the hardware to compress the transition function. An example of the Shift-And execution is presented in Fig. 6. Compared to the classical Shift-And, we keep the *states* to represent the set of active states, but we compute *labels* from the STE CC instead of storing it directly. We first match the input character to all CCs of the LNFA and then compute the *labels* from the CCs output. In practice, if the CC of STE_i matches the input character, we set '1' at position i in *labels* (starting from the left). Then, we perform the transitions with $next = (states \gg 1) \text{OR } 10\ldots0$, and update *states* with *states* = *next* AND *labels*, which keeps the states matching the current input symbol. When *states*AND $0\ldots01 \neq 0^n$, where n is the number of states in LNFA, we report a match. Compared to the classical Shift-And, this version assumes that there is a single initial state q_0 and a single final state q_{n-1}. Various encoding schemes can be applied to character classes, such as the one-hot encoding scheme or the multi-zero prefix encoding scheme [18].

Implementation of LNFA. In RAP, we implement the Shift-And matching for LNFA, which closely resembles the state-matching process, so that we can store codes of CCs in the CAM and *states* in the active vector. We encode some CCs into multiple 32-bit codes with the multi-zero prefix encoding scheme. Each of these CCs occupies multiple columns of the CAM, whose active state needs to be shifted multiple bits per character. This disrupts the uniform bitwise shift operation of the Shift-And algorithm and requires an FCB to realize irregular routing. To address this, we require all CCs in an LNFA mapped to the CAM to be encodable within a single 32-bit code using the multi-zero prefix encoding scheme, and 84% of LNFAs satisfy this requirement in practice. Under this condition, the transfer function of LNFA can be encoded into a chain that is implemented using a vector rather than a crossbar, which reduces the footprint from $\Theta(n^2)$ to $\Theta(n)$. LNFAs mapped to the CAM and active vector are shown in Fig. 6. We devise a specialized routing

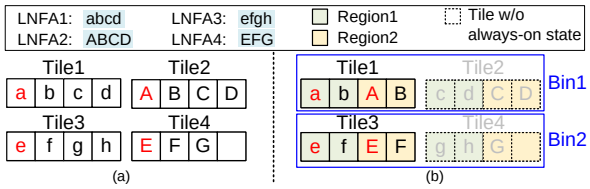


Figure 7: Mapping of 4 LNFAs (a) without and (b) with the binning scheme. STEs colored in red are the initial states.

path for the active vector to facilitate bitwise shift. The Active Vector right-shifts by one bit each cycle, controlling which columns remain active for the next input character. This approach reduces energy consumption during matching compared to activating all columns in NFA mode. In the example shown in Fig. 6, STE1 is an initial state, which keeps active for matching input characters working as *maskInitial*. In the first cycle, input `a` matches STE1 and STE2 via CAM, generating *labels* as `110`. The *next*, `100`, only activates STE1. Therefore, the *states* stored in the active vector is `100`. In the next cycle, *states* right shifts 1 bit and functions as *next*. Hence, input `b` matches active STE2 and unactive STE3, updating *states* to `010`. Similarly, when input `c` arrives, it matches the active STE3, triggering a match report.

Processing LNFA with CAMs leaves local switches unused, and CAMs cannot process some LNFAs. To address this, we reconfigure local switches for LNFAs by employing a one-hot encoding scheme, ensuring all LNFAs are mapped effectively. With this scheme, all CCs have a uniform code length of 256 bits, matching the alphabet size. Each one-hot code is stored across two local switch columns. The MSB of the 8-bit input character is used to select the local switch's output, while the remaining 7 bits are encoded into a 128-bit one-hot code that activates 1 row of the local switch. The pipeline buffer used for NBVA mode is repurposed to store *states* with a routing path that enables bitwise shift operations. Therefore, the LNFA mode of RAP decreases the memory usage compared to the NFA mode. In the meantime, RAP utilizes either CAM or a local switch for state matching and reconfigures the active vector for state transition. Therefore, RAP can process one input character per cycle in LNFA mode, matching the throughput of NFA mode.

Multi-LNFA Binning. We find that if a tile does not have any initial state, it can be power-gated when none of its states is activated by global routing. For LNFAs, only the first STE is an initial state, so we can group multiple LNFAs as a bin and map them in a regex-sliced manner, as shown in Fig. 7(b). Here, all initial states within the bin are placed in a single tile, allowing the remaining tiles storing the bin to stay power-gated when the initial states do not match the input character. Otherwise, all tiles contain an initial state and perform state-matching every cycle, as shown in Fig. 7(a). Overall, binning consolidates the initial states to a restricted region and leaves more tiles idle, leading to lower energy consumption.

Since the active vector only supports bitwise shift, LNFAs of the bin are mapped regex-sliced across tiles and in a region-based manner within the tile. As shown in Fig. 7(b), mapping Bin1, which contains LNFA1 and LNFA2, across two tiles requires each LNFA to be split into two parts. Then, we place the first two STEs of both LNFAs of the bin in Tile1, while the third and fourth STEs

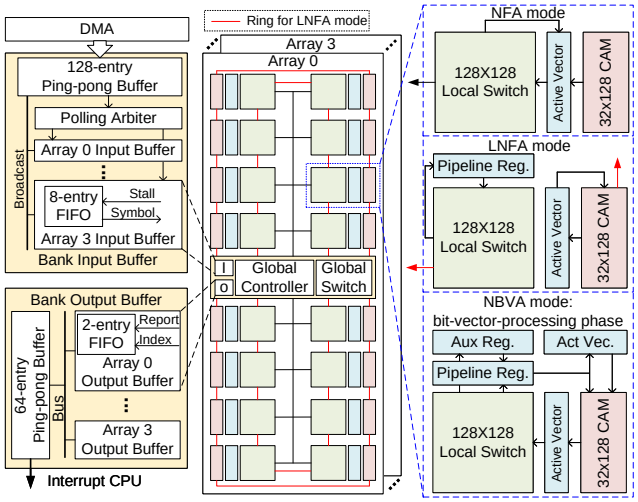


Figure 8: Overview of a hierarchical RAP bank and tile configuration of three modes.

are positioned in Tile2. To organize these LNFAs within a tile, the tile is divided into multiple regions, with the number of regions matching the number of LNFAs in the bin. Then we map LNFA1 to Region1 and LNFA2 to Region2. If the sizes of LNFAs are different within a bin, we treat them as the maximum size LNFA inside the bin, leaving partial regions unused.

Each tile can only support bins with an identical number of LNFAs because tiles need to be split into constant regions for mapping. In addition, only one STE within the LNFA will connect to a corresponding STE in another tile if it spans multiple tiles, ensuring a consistent global routing pattern. We devised a ring routing network specialized for LNFA global routing and power-gated the global switch. The width of the ring needed for global routing is determined by the number of LNFAs inside the bin. The ring connects adjacent tiles with global wires over a short distance, which introduces low area and energy overhead.

3.3 System Architecture of RAP

The overall architecture of RAP, as shown in Fig. 8, involves a three-level hierarchy: bank, array, and tile. Each bank includes four arrays and I/O, while each array consists of sixteen tiles and a 256×256 fully-connected crossbar (FCB) as a global switch. The dimension of the global switch for state transitions limits the number of tiles inside an array. Although a larger global switch can connect more tiles to support exceptionally large regexes, the quadratically increasing global switch reduces the system energy and area efficiency. Each tile includes a 32×128 8T-SRAM based CAM and a 128×128 FCB as a local switch. The capacities of the CAM and local switch are decided under a trade-off between the density of the local switch and the global routing bandwidth allocated to each tile. In the current tile design, 32 STEs can communicate with other tiles through the global switch.

RAP tiles support NFA, NBVA, and LNFA modes. Each tile in an array can be individually configured to any of the modes. To avoid global routing complexity and overheads, communication

between arrays is not supported in RAP. As such, RAP can support regexes with up to 2048 STEs in NFA and LNFA modes. Through design space exploration (See Section 5), we only support at most 32 LNFAs inside a bin for LNFA mode, so the width of the ring is set to 64 bits. In NBVA mode, the columns of the CAM can be randomly assigned to store either CCs or BVs, so the maximum size of a single BV is 4064 bits. Meanwhile, RAP does not support the transmission of bit vectors across tiles. Hence, BVs exceeding this limit must be split into multiple vectors that conform to hardware constraints. Therefore, RAP supports regexes with at most 64528 STEs after unfolding in NBVA mode.

To properly manage different operating modes across tiles, RAP has a Global Controller to coordinate behavior throughout the array and local controllers in each tile. The Global Controller decides the enabling of the CAMs and local switches. The local controllers select the data sent to CAMs and local switches and store the outputs in registers. In NFA and LNFA mode, both CAMs and local switches are always set to enable. In NBVA mode, the Global Controller stalls other tiles within the same array when any tile starts the bit-vector-processing phase. In tiles without active BV-STEs, the CAM and local switch are disabled to save energy. The depth of BVs in each array can be different, leading to different bit-vector-processing phase latencies. To reduce the throughput penalty incurred by the stalls, we designed two levels of buffering to hide the latency across arrays partially. To reduce the throughput discrepancy between NBVA mode and NFA/LNFA mode, multiple RAP banks can be configured to share the workload of low throughput banks.

Input/Output Streaming. The I/O interfaces the RAP and the host CPU. The hardware configuration is pre-loaded to RAP during deployment. The system transmits streaming data through DMA to the Bank Input Buffer of RAP for regexes matching without a cache structure. When the CPU receives matching results from the Bank Output Buffer, further analysis and actions will be taken.

The two-level input buffer structure is adopted from BVAP [52] to accommodate scenarios where NBVAs in different arrays are activated by different input characters, as shown in Fig. 8. The Bank Input Buffer is a 128-entry ping-pong buffer to hide the latency of loading data through DMA, and each array contains an 8-entry FIFO as its input buffer. Array Input Buffer broadcasts one 8-bit input symbol to all tiles when the bit-vector-processing phase does not stall the array. If the bank contains tiles in NBVA mode, the Bank Input Buffer employs a polling arbiter to process the data requests issued by each array. Otherwise, the arbiter is disabled, and the data from the Bank Input Buffer is broadcast to all arrays.

Each bank includes a 64-entry ping-pong Bank Output Buffer, and each array contains a 2-entry FIFO as an Array Output Buffer. Bank Output Buffer collects data from arrays through a bus because the match rate is typically lower than 10%, and thus the bandwidth is low. When the Bank Output Buffer is full, an interruption is sent to the CPU, prompting it to retrieve reports and clear all entries.

4 Compilation and Mapping

We have implemented a regex-to-hardware compiler that allows for high-level hardware programming. We compile each regex into one of the three modes available in RAP (defined in §3): NBVA, LNFA, and NFA. The choice is based on the characteristics of the regex in order to optimize the space and energy costs. Fig. 9 presents the decision process to choose the RAP mode for each regex.

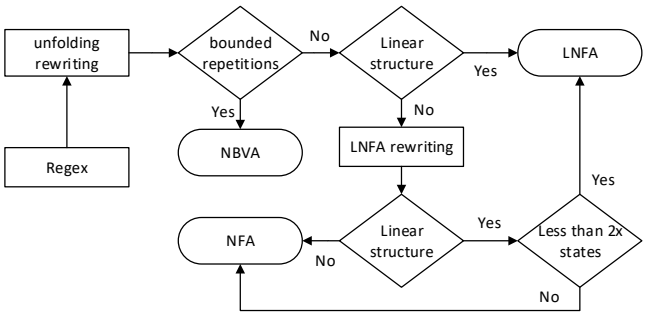


Figure 9: Decision graph for the compilation of RAP between the NBVA, LNFA and NFA modes.

Compilation Procedure. Compiling a regex into a RAP mode involves three steps: (1) regex rewriting to execute it in the chosen mode, (2) mapping of the regex to hardware (STEs and bit vectors in tiles), and (3) the regex-to-automaton construction. In this section, we present the compilation of NBVA and LNFA, and omit the NFA procedure that corresponds to the classical Glushkov construction.

4.1 Compilation for NBVA

The NBVA compiler considers several key parameters: the unfolding threshold, which controls the bounded repetitions to be unfolded into NFA states, and the depth and width of the CAM hardware.

Unfolding rewriting. The first step that we operate is the unfolding rewriting. It consists of unfolding a bounded repetition whenever its upper bound is below the unfolding threshold.

Example 4.1. Consider that the unfolding threshold is 4, the regex `ab(cd){2}e{1,3}f{2,}g{5}` is rewritten into `abcdcdee?e?fff*g{5}` after unfolding the first 4 bounded repetitions. Only $g\{5\}$ is not unfolded because the bound is greater than the threshold.

Bounded repetition rewriting. After the unfolding, we rewrite bounded repetitions to map with the BV actions supported by RAP. As the hardware does not support natively arbitrary read between m and n, we rewrite regexes of the form `r{m,n}` into `r{m}r{0,n-m}` where `r{m}` is simulated with $r(m)$ and `r{0,n-m}` with *rAll*.

Example 4.2. Consider the regex `ab{10,48}cd{34}ef{128}` and a depth $d = 16$. First, the compiler replaces the bounded repetition `b{10,48}` with `b{10}b{0,38}` to support the bounded repetition using *rAll*. Then, it rewrites `d{34}` into `d{32}dd` to use $r(32)$ for a width of 2. We do not rewrite `f{128}` as it can already be supported by BV action $r(128)$ for a width of 8.

Splitting. Once all BV transitions can be simulated with the hardware BV actions, we need to map the regex states into hardware STEs and tiles. The RAP compiler splits regexes that cannot be mapped to a single tile into several tiles. There are two main constraints on NBVA tiles: (1) at most 128 CAM columns, and (2) no r or *rAll* actions in the same tile.

Example 4.3. Consider the regex $r =$ `a{1024}bc{0,16}` and depth $d = 4$. For `a{1024}`, we would use one CAM column for the character class a and a width of $w = 1024/4 = 256$, which accounts

for 256 columns for the bit vector. In total, we need 258 (one extra column for *set1* action) columns, which exceeds the limitation of 128 CAM columns per tile (as well as width ≤ 128). We designed an algorithm that splits regexes based on a set of constraints. For the bounded repetition `a{1024}`, we proceed with a dichotomic search to find the splitting point k such that cols(`a{k}`) ≤ 128. We find that for $k = 504$, we need only $1 + 1 + (504/4) = 128$ CAM columns for $a\{504\}$. Therefore, we split `a{1024}` into three tiles: `a{504}`, `a{504}` and `a{16}`. Because the last tile contains only 3 CAM columns, we could add `bc{0,16}` in the same tile. However, the RAP design disallows for *r* and *rAll* actions in the same tile. Therefore, we put `bc{0,16}` in a fourth tile.

4.2 Compilation for LNFA

We use the Shift-And algorithm to execute regexes in the LNFA mode of RAP. Shift-And can be viewed as the simulation of a homogeneous automaton, where the states can be placed in order $q_0, q_1, \ldots, q_{n-1}$ on a line and each transition is from a state q_i to a neighboring state q_{i+1}. In addition to computing masks to execute the Shift-And algorithm, our compiler performs rewriting to execute more regexes with LNFA instead of NFA, inspired by [23]. As shown in Fig. 9, a regex will be compiled to LNFA if the rewriting does not increase the number of states by more than 2×, considering the smaller area of the LNFA mode over the NFA mode.

Example 4.4. Consider the regex $r =$ `a(b{1,2}|c)e`. As it is, r cannot be compiled into LNFA mode because its equivalent automaton has a transition between states *a* and *e* that are at positions 0 and 4 respectively (i.e., not a line). Our compiler unfolds the bounded repetition and distributes the union over concatenation to rewrite r into `a(b|bb|c)e` and finally `abe|abbe|ace` where each member of the union can be separately executed in LNFA mode. This transformation comes at the cost of more STEs on the hardware.

4.3 Hardware Mapping

Through the mapping process, we determine the mode of each RAP array and the associated regexes. Then, we configure RAP arrays accordingly during deployment ahead of runtime. For NFA and NBVA models, the mapper uses a greedy algorithm to map regexes to RAP arrays while allocating resources for CCs and BVs. It groups regexes together and ensures that each group can be mapped into a single RAP array without hardware violation.

For LNFAs, we perform binning before mapping. We first sort LNFAs based on their size. Next, we group the regexes into the bin with the largest number of regexes along this sorting sequence. When the size of the regex exceeds the upper limits supported by the bin, we reduce the number of LNFA in the bin by half. This process is repeated until each bin contains only one LNFA. Finally, we treat each bin as one regex and apply the greedy mapping algorithm. Overall, we can achieve an average utilization rate higher than 90% across all benchmarks and RAP modes.

5 Experimental Evaluation

This section presents the evaluation setups and performance analysis of RAP. We first conduct a design space exploration to determine the depth of BVs in NBVA mode and the upper bound for the number of LNFAs within a bin for each benchmark. We then compare the performance between NFA and NBVA modes for NBVA-based workloads in Table 2, as well as the performance between NFA and LNFA modes for LNFA-based workloads in Table 3. Finally, we evaluate the performance of RAP against SotA ASIC [18, 41, 52], GPU [23], CPU [51], and FPGA [49] solutions.

5.1 Datasets

We evaluate the performance of RAP over 7 applications, which contain over 20,000 regexes collected from real applications. These benchmarks are: (1) the **Snort** [33, 40] and (2) **Suricata** benchmarks [42] that contain patterns for network traffic, (3) the **Prosite** benchmark that contains protein motifs from the PROSITE database [34, 39], (4) the **Yara** [45] and (5) **ClamAV** [9] benchmarks with patterns for virus detection, (6) the **SpamAssassin** benchmark [12] that includes patterns for detecting spam email, and (7) the **RegexLib** dataset [31] which is a collection of regexes for validating user input. The complete dataset is publicly available here. Compared to the popular benchmarks ANMLZoo [46] and AutomataZoo[47], our set of benchmarks contains a more up-to-date list of regexes. Additionally, regexes with bounded repetitions are unfolded in those benchmarks, making them unsuitable for evaluating the performance of the NBVA module. Our set of benchmarks is a superset of the benchmarks considered in BVAP [52]. For a fair comparison with hAP [50], a SotA FPGA design, we use the same ANMLZoo [46] dataset to evaluate the performance of RAP in order to directly compare with the results reported in [50].

5.2 Experimental Setup

Methodology. The regexes of the dataset are first compiled with the custom compiler in Rust. Next, we combine the regexes into groups by the mapper implemented in Python, whose regexes are mapped to an array. Finally, we use a custom cycle-accurate simulator designed for RAP simulation in Python, which can also simulate existing in-memory automata accelerators like BVAP [52], CAMA [18], and CA[41]. The simulator uses the actual dataflow to emulate the cycle-accurate hardware behavior. Meanwhile, we performed consistency checks on the datasets to verify the functionality of RAP under all modes and the correctness of the hardware simulator by comparing matching results of the simulator against a production software matcher called Hyperscan [51]. For the CPU and GPU experiments, we use a desktop machine running on Ubuntu 22.04 and equipped with an Intel Core i9-12900K CPU, an NVIDIA GeForce RTX 4060 Ti, and 32 GB of memory. For the energy measurement on the GPU, we adopt an approach derived from [21]. It consists of measuring the average power consumption with the NVIDIA Management Library (NVML) [28] at the constant rate of 50Hz, which is the sampling frequency of the power hardware counter. In this setting, one thread runs in the background and reads the counter every 20ms. The average power is computed from those measured points. For measuring the average power on the CPU, we use the official Intel tool called Intel SoC Watch [19] to measure the average power of the CPU socket. For the throughput measurements, we exclude the IO time, i.e. the copy times between the CPU and the GPU, and between the CPU and memory.

We focus on two system metrics: energy efficiency and compute density. Energy efficiency is the processing throughput divided by

Table 1: Circuit models in 28nm.

Type	Size	Energy (pJ)	Delay (ps)	Area (μm^2)	Leakage (μA)
8T SRAM	128×128	1-14	298	5655	57
	256×256	2-55	410	18153	228
8T CAM	32×128	4	325	2626	14
Local Controller	N/A	2	90	2900	18
Global Controller	N/A	2	400	1400	9
Global wire	1 mm	0.07	66	50	N/A

the total power, while compute density is the throughput divided by the hardware area, measuring the hardware's area efficiency.

Circuit-Level Modeling. Table 1 lists circuit models used in our evaluations, including access energy, delay, and area. Values of SRAM and CAM arrays are derived from SPICE simulations on custom-designed circuits in TSMC 28nm CMOS. The local and global controllers are designed in Verilog and then synthesized using Synopsys DC. The global wire estimation is based on the data provided in CA [41]. For a fair comparison, all other automata processor architectures reported in this paper adopt 128×128 FCB as local switches and are simulated with the same circuit model and simulator. We also use the same greedy algorithm for mapping. The RAP tile shares a similar area as a CAMA tile, so RAP 's global wire delay is estimated to be 26.1ps, which is reported by CAMA [18]. The largest delay of the RAP pipeline stage is 436.1 ps, which sets its clock frequency to 2.08 GHz. Note that all chosen clock frequencies include a 10% safety margin.

5.3 Design Space Exploration

The depth of BVs in the NBVA mode and the maximum number of regexes inside the bins (bin size) are two user-controlled RAP parameters that optimize area, energy, and throughput performance for different workloads. A larger depth of BVs offers a higher compression rate, reducing area and energy, but suffers from throughput punishment during the bit-vector processing phase. A larger bin size concentrates initial states in fewer tiles and saves energy for LNFA processing, but potentially increases the area due to redundancy caused by the mapping procedure. This experiment includes all regexes compiled to NBVA and LNFA modes in all seven benchmarks. No regex has been compiled to NBVA in Prosite.

Because the depth of the BVs affects area, energy, and throughput, we seek proper tradeoffs, as shown in Fig. 10(a). We choose the depth that improves energy and area while offering acceptable throughput. We find that datasets containing large bit vectors within a large portion of NBVAs favor deeper BV depths, as larger bit vectors benefit from the higher compression rates and reduce the area of BVs. For example, regex `AppPath=[C-Z]:\\\\[^\\]{1,64}\.exe` in Yara benefits from a large depth. The size of its bit vector is 64, thus benefiting from a high compression rate. It also has a complex prefix that leads to a low activation rate of the bit vector. In contrast, datasets with smaller bit vectors prefer lower depths due to limited area improvement and noticeable throughput penalty. For example, regex `Jeste.{1,8}firm.{1,8}` in SpamAssassin requires a small depth because of the limited size of bit vectors.

Fig. 10(b) depicts the area and energy consumption of LNFA mode. A larger bin size reduces energy consumption when the redundancy area is negligible. So, we choose the bin size with the highest energy efficiency without a significant area increment. The results indicate that a dataset with small LNFAs benefits from a larger bin size, as it groups small LNFAs into a single bin, concentrating their initial states within one tile rather than dispersing them across the array to reduce state matching energy. Conversely, a dataset with few small LNFAs favors a smaller bin size, as insufficient small LNFAs can't fill a large bin, which prevents wasted area. Fig. 10 presents the parameters we chose for the best performance. Hence, the largest bin size we need to support is 32.

5.4 Performance Analysis of RAP

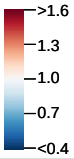
We assess the performance improvements of NBVA and LNFA modes by matching 100,000 input characters against all regexes compiled to NBVAs and LNFAs by our compiler. We unfold all regexes to basic NFAs to obtain NFA mode results.

Table 2 compares the total energy, area, and throughput between the NBVA and NFA modes of RAP, as well as SotA ASIC designs [18, 41, 52], for regexes that can be compiled to NBVA within the benchmarks. Compared to the NFA mode of RAP, NBVA mode consistently offers lower energy consumption because compressing multiple standard STEs to one BV-STE reduces state-matching and state-transition energy. It also provides smaller footprints, benefiting from area saving in CCs storage and increased compression rate for large bit vectors. NFA mode's energy and area surpass NBVA mode by 3.7× and 4.0× on average. Compared to BVAP, a SotA dedicated to NBVA execution, RAP's NBVA mode consumes merely 20% more energy, while keeping a smaller area averaged across benchmarks. This is because dynamic allocation eliminates the area wasted by unused bit vectors, and RAP can achieve a higher compression rate than BVAP in some cases. NBVA mode of RAP consistently achieves lower energy and area than CAMA and CA because we use BVs to compress the bounded repetitions, except for RegexLib. In RegexLib, the ratio and size of BVs are both low, making the benefits of NBVA mode insufficient to compensate for the overhead introduced by the local controller.

In processing LNFAs, RAP reduces the energy consumption by 79% on average compared to NFA mode, as shown in Table 3. The reason is that most columns of CAM are power-gated by active vector in LNFA mode, but all columns need to work under NFA mode. The binning procedure maps initial states to fewer tiles, so more tiles are power-gated when inactive, further enhancing the energy efficiency. LNFA utilizes both CAM and local switches for storage of CCs, which decreases the area by 2× in theory. However, additional states introduced by LNFA conversion and redundancy caused by binning reduce the area gain. Overall, the area achieved in LNFA mode is 33% lower than in NFA mode. For the Yara dataset, the area of LNFA mode is 13% larger than that of NFA mode, but it is still acceptable due to its energy improvements. LNFA and NFA modes of RAP achieve the same throughput as they both process one input character per cycle and run at the same clock frequency. Compared to CAMA, BVAP, and CA, RAP's LNFA mode achieves lower energy and area because state transition is performed by the active vector, saving the energy of the routing switches.

The overall energy and area requirements of RAP are evaluated with all regexes in all benchmarks. Fig. 11 shows the breakdown of

(a)	RegexLib				SpamAssassin				Snort				Suricata				Yara				ClamAV			
Depth	4	8	16	32	4	8	16	32	4	8	16	32	4	8	16	32	4	8	16	32	4	8	16	32
Energy	1.0	1.0	1.0	1.0	1.0	1.2	1.2	1.2	1.0	0.9	1.0	0.9	1.0	0.9	1.0	0.9	1.0	0.7	0.7	0.5	1.0	0.9	0.9	0.9
Area	1.0	1.0	1.0	1.0	1.0	1.1	1.2	1.2	1.0	0.9	0.9	0.8	1.0	0.9	0.9	0.8	1.0	0.6	0.6	0.5	1.0	0.9	0.9	0.8
Latency	1.0	1.7	1.7	3.3	1.0	1.3	2.0	3.3	1.0	1.1	1.1	1.5	1.0	1.1	1.4	2.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0



(b)	RegexLib					Prosite					SpamAssassin					Snort					Suricata					Yara					ClamAV				
Bin size	1	8	16	32	64	1	8	16	32	64	1	8	16	32	64	1	8	16	32	64	1	8	16	32	64	1	8	16	32	64	1	8	16	32	64
Energy	1.0	0.8	0.7	0.6	0.5	1.0	1.0	0.8	0.6	0.7	1.0	0.8	0.6	0.4	0.3	1.0	0.7	0.6	0.5	0.4	1.0	0.7	0.6	0.5	0.4	1.0	0.7	0.5	0.4	0.3	1.0	0.3	0.3	0.3	0.6
Area	1.0	1.1	1.1	1.1	1.1	1.0	1.0	1.0	1.0	1.1	1.0	1.0	1.0	1.0	1.1	1.0	1.0	1.1	1.1	1.2	1.0	1.0	1.0	1.1	1.3	1.0	1.5	1.6	1.8	2.0	1.0	1.1	1.4	1.5	1.8
Latency	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Figure 10: Energy, area, and throughput of design space exploration of (a) NBVA (normalized to depth=4) and (b) LNFA (normalized to bin size=1). The chosen parameters are in red text.

Table 2: Comparison of NBVA mode of RAP (baseline), NFA mode of RAP, CAMA [18], BVAP [52], and CA [41].

Dataset	Energy (μJ)					Area (mm^2)					Throughput (Gch/s)				
	NBVA	NFA	CAMA	BVAP	CA	NBVA	NFA	CAMA	BVAP	CA	NBVA	NFA	CAMA	BVAP	CA
RegexLib	71	71	54	40	40	1.36	1.37	1.15	1.23	1.87	1.70	2.08	2.14	1.96	1.82
SpamAssassin	43	72	53	36	45	0.86	1.69	1.42	0.92	2.33	1.91	2.08	2.14	1.72	1.82
Snort	188	937	692	132	607	3.67	19.39	16.34	5.51	25.39	1.69	2.08	2.14	1.83	1.82
Suricata	191	847	629	136	540	3.80	17.08	14.39	4.83	22.71	1.87	2.08	2.14	1.74	1.82
Yara	62	328	235	67	224	1.36	7.69	6.51	2.26	9.16	2.07	2.08	2.14	1.85	1.82
ClamAV	1632	8294	5846	1887	5725	35	198	167	57	257	1.00	2.08	2.14	1.26	1.82
Average (normalized to NBVA)	1.0×	3.7×	2.7×	0.8×	2.5×	1.0×	4.0×	3.4×	1.4×	5.2×	1.0×	1.3×	1.3×	1.0×	1.1×

Table 3: Comparison of LNFA mode of RAP (baseline), NFA mode of RAP, CAMA [18], BVAP [52], and CA [41].

Dataset	Energy (μJ)					Area (mm^2)					Throughput (Gch/s)				
	LNFA	NFA	CAMA	BVAP	CA	LNFA	NFA	CAMA	BVAP	CA	LNFA	NFA	CAMA	BVAP	CA
RegexLib	45	121	92	92	66	1.59	2.29	1.92	3.12	3.18	2.08	2.08	2.14	2.00	1.82
Prosite	26	72	54	54	47	0.87	1.41	1.18	1.93	1.99	2.08	2.08	2.14	2.00	1.82
SpamAssassin	135	576	436	437	309	7.05	10.71	9.00	14.70	15.16	2.08	2.08	2.14	2.00	1.82
Snort	48	214	163	163	123	2.47	4.01	3.38	5.41	5.42	2.08	2.08	2.14	2.00	1.82
Suricata	44	196	149	150	113	2.25	3.69	3.10	4.97	4.96	2.08	2.08	2.14	2.00	1.82
Yara	8	30	23	23	16	0.63	0.55	0.46	0.75	0.78	2.08	2.08	2.14	2.00	1.82
ClamAV	5	53	40	40	30	0.67	1.01	0.85	1.38	1.43	2.08	2.08	2.14	2.00	1.82
Average (normalized to LNFA)	1.0×	4.7×	3.6×	3.6×	2.7×	1.0×	1.5×	1.2×	2.0×	2.0×	1.0×	1.0×	1.0×	1.0×	0.9×

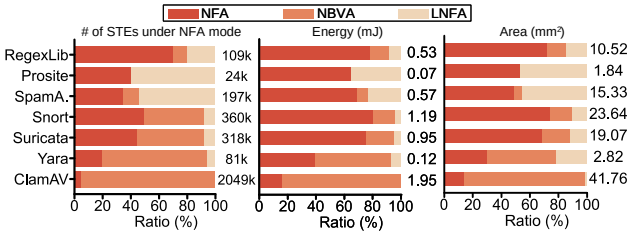


Figure 11: The proportion of the number of STEs, energy, and area in NFA, NBVA, and LNFA modes. The absolute total values of each measurement are annotated in the figures.

the number of STEs, energy, and area into three types of automata models. It can be observed that the proportion of energy and area consumed by NFAs is higher than the percentage of STEs in NFA modes, indicating the effectiveness of NBVA and LNFA modes.

5.5 Comparison with State-of-the-arts

We evaluated our proposed architecture using all benchmarks collected from real-world applications with different workloads. We compile all regexes using the optimal automata model and parameters derived from design space exploration. To integrate the energy and area results of different modes within the RAP architecture, we allocate additional resources to the RAP arrays in the NBVA mode to increase throughput. If the throughput of a RAP array in the NBVA mode is lower than 2 Gch/s, we assign another RAP array to work on the same regexes to share the workload and increase the throughput. This method introduces an area overhead of less than 3%. Overall, we use the throughput of the NBVA mode after allocating additional resources as the system throughput.

Comparison against ASICs. Fig. 12 summarizes the area, throughput, energy efficiency, compute density, and power of RAP, BVAP [52], CAMA [18], and CA [41]. RAP benefits significantly from NBVAs and LNFAs while incurring little overhead in NFAs. As a result, RAP enhances overall performance by combining the results of all three modes, effectively handling datasets that are a mixture of NFA, NBVA, and LNFA. Compared to BVAP, RAP improves the compute density by 1.6×. With datasets dominated by NFA and LNFA workloads, such as RegexLib and Prosite, BVAP incurs redundant area overhead due to underutilized bit vectors. For ClamAV and Yara, RAP decreases the area by 40%, because the depth of BVs used in RAP is larger than that used in BVAP, leading to a lower area of BVs. Although RAP takes 20% higher energy for NBVAs, the energy improvements in LNFA mode compensate for this. Overall, RAP achieves a comparable energy efficiency to BVAP.

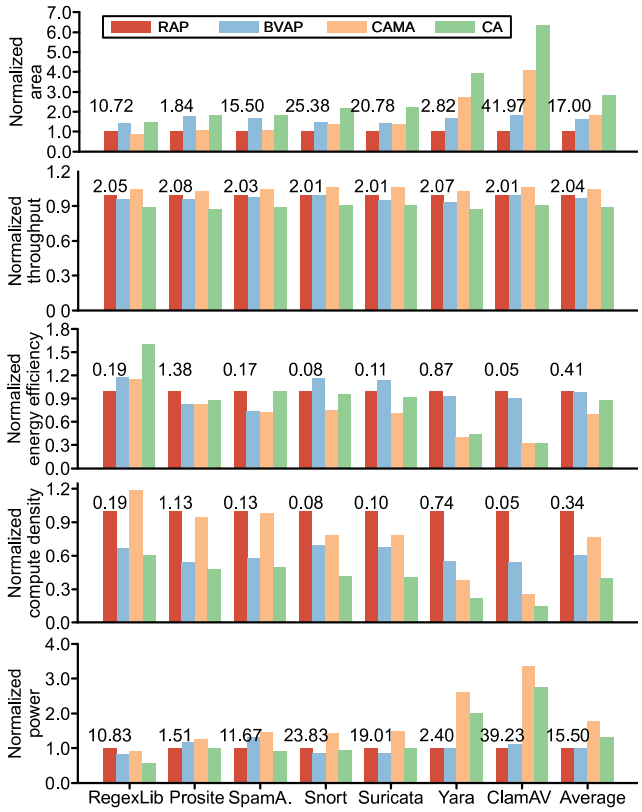


Figure 12: Performance comparison among RAP, BVAP, CAMA, and CA. All values are normalized to that of RAP, which are annotated in the figure.

Averaging across all benchmarks, RAP achieves notable improvement in energy efficiency, compute density, and power over CAMA by 1.5×, 1.3×, and 1.8×, and CA by 1.2×, 2.5×, and 1.3×. Prosite and SpamAssassin are LNFA dominant workloads, allowing RAP to achieve 28% higher energy efficiency while maintaining similar compute density compared to CAMA. On Snort and Suricata datasets, RAP 's energy efficiency surpasses CAMA by 28%, and their compute density exceeds CAMA by 22%, and CA by 59%. These datasets have a similar workload distribution between NFA and NBVA, so the overhead from NFA is minimal, but RAP greatly benefits from NBVA. Yara and ClamAV workloads are dominated by NBVA, where RAP significantly reduces energy and area by efficiently compressing bit vectors in NBVAs. Therefore, RAP enhances energy efficiency and compute density by 2.5× and 2.7× compared to CAMA and by 2.3× and 4.5× compared to CA. Taking into account similar throughput, RAP also reduces power by 61% and 50% compared to CAMA and CA. Because NFA mode in RAP incurs area and energy overhead due to the local controller, which results in a 20% performance degradation in the RegexLib dataset, as it predominantly uses NFA.

Comparison against CPU and GPU. Meanwhile, various hardware platforms are evaluated to perform general-purpose regex matching. Fig. 13 compares the power and throughput of RAP, GPU (HybridSA [23]), and CPU (Hyperscan [51]). Hyperscan [51] is a state-of-the-art multi-pattern engine for modern CPUs. Its matching algorithm uses SIMD instructions and a variant of Shift-And to accelerate matching. HybridSA [23] is a hybrid CPU-GPU engine that executes most regexes on the GPU with variants of Shift-And while the rest are executed on the CPU. We experiment using the GPU mode of HybridSA. The GPU engine consumes 16× power compared to RAP, while RAP achieves 9.8× higher throughput on average. RAP also uses 1.1% of power compared to CPU but achieves 60× higher throughput. Overall, RAP achieves >100× and >1000× greater energy efficiency compared to GPU and CPU solutions.

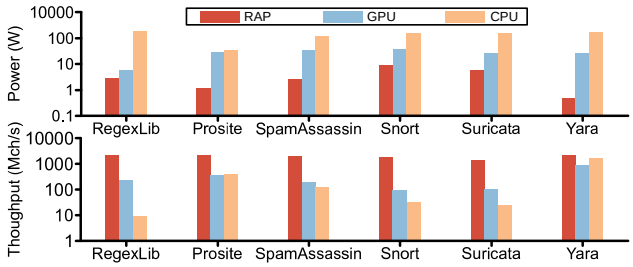


Figure 13: Comparison of RAP, GPU [23], and CPU [51].

Table 4: Comparison between RAP and FPGA [49].

Dataset (ANMLzoo)	RAP		hAP [49]	
	Power (W)	Throughput (Gch/s)	Power (W)	Throughput (Gch/s)
Brill	5.17	2.08	1.56	0.18
ClamAV	3.22	2.07	1.42	0.18
Dotstar	2.48	2.08	1.47	0.18
PowerEN	3.28	2.08	1.52	0.18
Snort	7.79	2.08	1.41	0.15

Comparison against FPGA. We also compare our design to hAP [49] on the ANMLzoo benchmark [46] as shown in Table 4. Among all datasets, only ClamAV includes regexes with large bounded repetitions. RAP achieves a throughput that is 11.5×-13.8× higher than hAP, but the power only increases by 1.7×-5.5×. RAP shows a higher energy efficiency compared to FPGA designs.

6 Conclusion

We present RAP, the first reconfigurable automata processor designed for efficient regular pattern matching across diverse workloads. It supports the NBVA model by unified storage for both character classes and bit vectors, accompanied by an encoding scheme for BV actions. The RAP hardware incorporates a specialized routing path to efficiently realize the linear transfer function of LNFA, further optimized by a binning algorithm. Through cross-stack co-design, RAP achieves higher efficiency and higher compute density across real-world benchmarks, over state-of-the-art regex matching solutions on various hardware platforms.

Acknowledgments

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

References

[1] Junaid Arshad, Muhammad Ajmal Azad, Muhammad Mahmoud Abdeltaif, and Khaled Salah. 2020. An intrusion detection framework for energy constrained IoT devices. *Mechanical Systems and Signal Processing* 136 (2020), 106436. https://doi.org/10.1016/j.ymssp.2019.106436

[2] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. 2016. Scalable Algorithms for NFA Multi-striding and NFA-based Deep Packet Inspection on GPUs. *IEEE/ACM Transactions on Networking* 24, 3 (2016), 1704–1717. https://doi.org/10.1109/TNET.2015.2429918

[3] Ricardo Baeza-Yates and Gaston H. Gonnet. 1992. A New Approach to Text Searching. *Commun. ACM* 35, 10 (1992), 74–82. https://doi.org/10.1145/135239.135243

[4] 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

[5] Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. 2019. Automata Processing in Reconfigurable Architectures: In-the-Cloud Deployment, Cross-Platform Evaluation, and Fast Symbol-Only Reconfiguration. *ACM Trans. Reconfigurable Technol. Syst.* 12, 2 (2019). https://doi.org/10.1145/3314576

[6] 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. https://doi.org/10.1145/1880153.1880157

[7] Milan Ceška, Vojtech Havlena, Lukáš Holík, Jan Korenek, Ondrej Lengál, Denis Matoušek, Jirí Matoušek, Jakub Semric, and Tomáš Vojnar. 2019. Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (New York, New York). IEEE, 109–117. https://doi.org/10.1109/FCCM.2019.00025

[8] Jian Chen, Xiaoyu Zhang, Tao Wang, Ying Zhang, Tao Chen, Jiajun Chen, Mingxu Xie, and Qiang Liu. 2022. 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* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 1029–1041. https://doi.org/10.1145/3470496.3533043

[9] ClamAV. 2023. ClamAV - Open Source Antivirus Engine. Available at https://www.clamav.net/. [Online; Accessed 6 Nov, 2024].

[10] 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

[11] Tim Dunn, Harisankar Sadasivan, Jack Wadden, Kush Goliya, Kuan-Yu Chen, David Blaauw, Reetuparna Das, and Satish Narayanasamy. 2021. Squigglefilter: An accelerator for portable virus detection. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 535–549.

[12] Apache Software Foundation. 2022. Apache SpamAssassin. Available at https://spamassassin.apache.org/. [Online; Accessed 6 Nov, 2024].

[13] Esteban Garzón, Roman Golman, Zuher Jahshan, Robert Hanhan, Natan Vinshtok-Melnik, Marco Lanuzza, Adam Teman, and Leonid Yavits. 2022. Hamming Distance Tolerant Content-Addressable Memory (HD-CAM) for DNA Classification. *IEEE Access* 10 (2022), 28080–28093. https://doi.org/10.1109/ACCESS.2022.3158305

[14] Esteban Garzón, Eyal Rechef, Roman Golman, Odem Harel, Yuval Harary, Paz Snapir, Marco Lanuzza, Adam Teman, and Leonid Yavits. 2025. A 128-kbit Approximate Search-Capable Content-Addressable Memory (CAM) With Tunable Hamming Distance. *IEEE Journal of Solid-State Circuits* (2025), 1–11. https://doi.org/10.1109/JSSC.2025.3529715

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

[16] Robert Hanhan, Esteban Garzón, Zuher Jahshan, Adam Teman, Marco Lanuzza, and Leonid Yavits. 2022. EDAM: edit distance tolerant approximate matching content addressable memory. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 495–507. https://doi.org/10.1145/3470496.3527424

[17] Qinwen Hu, Muhammad Rizwan Asghar, and Nevil Brownlee. 2017. Evaluating network intrusion detection systems for high-speed networks. In *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*. 1–6. https://doi.org/10.1109/ATNAC.2017.8215374

[18] Yi Huang, Zhiyu Chen, Dai Li, and Kaiyuan Yang. 2022. CAMA: Energy and Memory Efficient Automata Processing in Content-Addressable Memories. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, New York, NY, USA, 25–37. https://doi.org/10.1109/HPCA53966.2022.00011

[19] Intel. 2023. Intel SoC. https://www.intel.com/content/www/us/en/docs/socwatch/get-started-guide/2023-1/overview.html Accessed: Nov. 18, 2024.

[20] 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 2022)*. ACM, New York, NY, USA, 733–748. https://doi.org/10.1145/3519939.3523456

[21] Jens Lang and Gudula Rünger. 2013. High-resolution power profiling of GPU functions using low-resolution measurement. In *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings 19*. Springer, 801–812.

[22] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching Using Bit Vector Automata. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 92 (2023), 30 pages. https://doi.org/10.1145/3586044

[23] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2024. HybridSA: GPU Acceleration of Multi-pattern Regex Matching using Bit Parallelism. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 331 (Oct. 2024). https://doi.org/10.1145/3689771

[24] Dai Li and Kaiyuan Yang. 2020. A Dual-Port 8-T CAM-Based Network Intrusion Detection Engine for IoT. *IEEE Solid-State Circuits Letters* 3 (2020), 358–361. https://doi.org/10.1109/LSSC.2020.3022006

[25] 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

[26] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous Automata Processing on GPUs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 1, Article 27 (2023), 27 pages. https://doi.org/10.1145/3579453

[27] Kai Ni, Xunzhao Yin, Ann Franchesca Laguna, Siddharth Joshi, Stefan Dünkel, Martin Trentzsch, Johannes Müller, Sven Beyer, Michael Niemier, Xiaobo Sharon Hu, and Suman Datta. 2019. Ferroelectric ternary content-addressable memory for one-shot learning. *Nat Electron* 2 (2019). https://doi.org/10.1038/s41928-019-0321-3

[28] NVIDIA. 2024. NVIDIA Management Library. https://docs.nvidia.com/deploy/nvml-api/index.html Accessed: Nov. 18, 2024.

[29] posix 2023. PCRE Syntax. Available at https://www.pcre.org/original/doc/html/pcrepattern.html. [Online; Accessed 18 July, 2023].

[30] 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

[31] RegexLib. 2023. Regular Expression Library. Available at https://regexlib.com/. [Online; Accessed 6 Nov, 2024].

[32] M. Sadegh Riazi, Mohammad Samragh, and Farinaz Koushanfar. 2017. CAMsure: Secure Content-Addressable Memory for Approximate Search. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 136 (Sept. 2017), 20 pages. https://doi.org/10.1145/3126547

[33] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Seattle, Washington) *(LISA '99)*. USENIX Association, USA, 229–238. https://www.usenix.org/legacy/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf

[34] 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

[35] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. FlexAmata: A Universal and Efficient Adaption of Applications to Spatial Automata Processing Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland). Association for Computing Machinery, 219–234. https://doi.org/10.1145/3373376.3378459

[36] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin 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)*. 86–98. https://doi.org/10.1109/HPCA47549.2020.00017

[37] 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

[38] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. 2017. Frequent subtree mining on the automata processor: challenges and opportunities. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3079079.3079084

[39] Christian J. A. Sigrist, Lorenzo Cerutti, Edouard de Castro, Petra S. Langendijk-Genevaux, Virginie Bulliard, Amos Bairoch, and Nicolas Hulo. 2009. PROSITE, A Protein Domain Database for Functional Characterization and Annotation. *Nucleic Acids Research* 38, suppl_1 (2009), D161–D166. https://doi.org/10.1093/nar/gkp885

[40] Snort. 2023. Snort - Network Intrusion Detection & Prevention System. Available at https://www.snort.org/. [Online; Accessed 6 Nov, 2024].

[41] 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

[42] Suricata. 2023. Suricata - Open Source Intrusion Detection and Prevention Engine. Available at https://suricata.io/. [Online; Accessed 6 Nov, 2024].

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

[44] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC '11)*. IEEE Computer Society, USA, 216–225. https://doi.org/10.1109/IISWC.2011.6114181

[45] VirusTotal. 2023. YARA: The pattern matching swiss knife for malware researchers. Available at https://virustotal.github.io/yara/. [Online; Accessed 6 Nov, 2024].

[46] 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

[47] 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

[48] Lei Wang, Shuhui Chen, Yong Tang, and Jinshu Su. 2011. Gregex: GPU Based High Speed Regular Expression Matching Engine. In *Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS '11)*. IEEE Computer Society, USA, 366–370. https://doi.org/10.1109/IMIS.2011.107

[49] Xuan Wang, Lei Gong, Jing Cao, Wenqi Lou, Weiya Wang, Chao Wang, and Xuehai Zhou. 2023. hAP: A Spatial-von Neumann Heterogeneous Automata Processor with Optimized Resource and IO Overhead on FPGA. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA). Association for Computing Machinery, 185–196. https://doi.org/10.1145/3543622.3573190

[50] Xuan Wang, Lei Gong, Jing Cao, Wenqi Lou, Weiya Wang, Chao Wang, and Xuehai Zhou. 2023. hAP: A Spatial-von Neumann Heterogeneous Automata Processor with Optimized Resource and IO Overhead on FPGA. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 185–196. https://doi.org/10.1145/3543622.3573190

[51] 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, Boston, MA, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[52] 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 *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, New York, NY, USA, 151–166. https://doi.org/10.1145/3620665.3640412

[53] 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

[54] 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

[55] Xiaodong Yu and Michela Becchi. 2013. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '13)*. ACM, New York, NY, USA, Article 18, 10 pages. https://doi.org/10.1145/2482767.2482791

[56] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1083–1100. https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[57] 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* (New Orleans, Louisiana, USA) *(PPoPP '12)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/2145816.2145833

A Artifact Appendix

A.1 Abstract

The artifact contains the source code used to simulate and evaluate the RAP designs proposed in this paper and the state-of-the-art automata processors compared with in Section 5. We have provided a cycle-level simulator for RAP. The artifact provides information on how to reproduce key results of the paper, namely the data presented in Section 5, Fig. 10 (RAP design space exploration); Section 5, Table 2 and Table 3 (performance comparison of the NBVA mode and LNFA mode of RAP); and Section 5, Fig. 12 (performance comparison between RAP and SotA automata processors).

A.2 Artifact check-list (meta-information)

- **Program:** RAP simulator
- **Data set:** Prosite, RegexLib, SpamAssassin, Snort, Suricata, Yara, ClamAV
- **Hardware:** Intel(R) Xeon(R) Gold 6136 CPU @ 3.00GHz
- **Metrics:** Area, Energy, Throughput, Compute density, Energy efficiency, Power
- **Output:** Metrics results in CSV and JSON format, and visualization figures in PDF format
- **How much disk space is required (approximately)?:** 6GB
- **How much time is needed to prepare workflow (approximately)?:** < 1 hour
- **How much time is needed to complete experiments (approximately)?:** 72 hours (with 40 cores) on all datasets. 10 hours (with 5 cores) for the SpamAssassin dataset
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License 2.0
- **Archived (provide DOI)?:** 10.5281/zenodo.15080391

A.3 Description

A.3.1 How to access. We have published the artifact on Zenodo. The Zenodo DOI URL is: https://doi.org/10.5281/zenodo.15080391

A.3.2 Hardware dependencies. Any computing cluster should be sufficient to run the relevant experiments. In our evaluations, we used 40 cores (in total) to run the experiments within a reasonable amount of time. We recommend using 4 GB of memory per core.

A.3.3 Software dependencies.

- A recent Linux or Windows distribution
- Python 3.10 or newer
- Conda
- Additional Python packages listed in the `RAP.yaml` file

A.3.4 Datasets. All the pre-compiled datasets are available in the repository (https://doi.org/10.5281/zenodo.15080391). Upon downloading and unzipping the repository, the datasets are located under the `./mnrl/` folder.

A.4 Installation

A minimal installation can be achieved using the following commands:

```
# create a new conda environment
conda env create -f RAP.yaml
conda activate RAP
```

Next, download the artifact at https://doi.org/10.5281/zenodo.15080391 and extract it with the following command:

```
# unzip the artifact and go to the root folder
unzip RAP_simulator.zip
cd RAP_simulator
```

A.5 Experiment workflow

To run the experiments, choose a subset of the dataset and a task to perform (Design Space Exploration, NFA, LNFA, or NBVA). Use the following syntax to run experiments (here, LNFA for the Yara and Prosite datasets):

```
python main_gap.py --data "Yara Prosite"
--task LNFA
```

The following syntax can be used to run an experiment over all the datasets:

```
python main_gap.py --data "All" --task <experiment>
```

You can choose the number of processors used by changing the `num_worker` variable in the `./utils/cfg.py` config file. An explanation of how to use the script is discussed in the next section.

A.6 Evaluation and expected results

We explain how to reproduce the key results of our paper.

A.6.1 Design Space Exploration. To reproduce the Design Space Exploration (DSE) results of Fig. 10 (NBVA and LNFA modes), run the following command:

```
python main_gap.py --data "All" --task DSE
```

The output JSON files are stored in the `./result/nbva_sweep_depth/` (resp. `./result/final_gap_sccs_fcb_map/`) folder for the NBVA (resp. LNFA) mode. Each output file is prefixed by the dataset name. Output tables are also generated and should match the results of Fig. 10.

A.6.2 RAP NBVA against RAP NFA and ASICs. To reproduce the comparison between RAP in NBVA (baseline) and NFA modes, and the state-of-the-art ASIC designs in Table 2, run the following command:

```
python main_gap.py --data "All" --task NBVA
```

The output CSV file is `table_2.csv` and is stored in the working directory (`RAP_simulator`). We expect the energy, area, and throughput to match the result of Table 2.

A.6.3 RAP LNFA against RAP NFA and ASICs. To reproduce the comparison between RAP in LNFA (baseline) and NFA modes, and state-of-the-art ASIC designs of Table 3, run the following command:

```
python main_gap.py --data "All" --task LNFA
```

The output CSV file is `table_3.csv` and is stored in the working directory (`RAP_simulator`). We expect the energy, area, and throughput to match the result of Table 3.

A.6.4 RAP vs ASICs. To reproduce the comparison between RAP and state-of-the-art ASIC designs of Fig. 12, which shows the overall performance comparison between RAP and state-of-the-art ASIC designs, run the following commands:

```
python main_gap.py --data "All" --task NFA
python main_gap.py --data "All" --task ASIC
```

You need to run the commands in A.6.2 and A.6.3 before running A.6.4. The output JSON files are stored in the `./result/final_gap/` folder. Each output file is prefixed by the dataset name. Output figures `fig12_<metrics>.pdf` are also generated and should match the results of Fig. 12.

A.7 Experiment customization

The experiments can be extended with different input strings by replacing the input files in the `./input/` folder.