Hardware Architecture for High-Performance Regular Expression Matching

Tsern-Huei Lee, *Senior Member, IEEE*

Abstract—This paper presents a bitmap-based hardware architecture for the Glushkov nondeterministic finite automaton (G-NFA), which recognizes a given regular expression. We show that the inductions of the functions needed to construct the G-NFA can be generalized to include other special symbols commonly used in extended regular expressions such as the POSIX 1003.2 format. Our proposed implementation can detect the ending positions of all substrings of an input string T, which start at arbitrary positions of T and belong to the language defined by the given regular expression. To achieve high performance, the implementation is generalized to the NFA, which processes K symbols in each operation cycle. We provide an efficient solution for the boundary condition when the length of the input string is not an integral multiple of K. Compared with previous designs, our proposed architecture is more flexible and programmable because the pattern matching engine uses memory rather than logic.

Index Terms—Hardware acceleration, nondeterministic finite automaton, regular expression.

1 Introduction

Deep packet inspection is an important component in network security appliances such as content firewall, intrusion detection, and antivirus systems. The function of deep packet inspection is to search for predefined patterns in packet payloads. Since a pattern may occur at any position of the payload, it is very time consuming especially when patterns are specified with regular expressions. According to some report [3], the pattern matching module can consume up to 70 percent of CPU computation power in an intrusion detection system. As a consequence, pure software-based pattern matching is not suitable for high-speed networks.

There are hardware accelerators for pattern matching, which can achieve multigigabits-per-second throughput performance. However, most of high-performance hardware accelerators handle only plain strings [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. The architectures proposed in [3], [4], [5], [6], [7], [8], and [9] are based on the famous Aho-Corasick (AC) algorithm [2], which has the advantages of matching multiple patterns simultaneously and providing deterministic performance guarantee under all circumstances. These designs use different approaches such as bitmap [3] and bit-split [4] to tackle the problem of potentially huge amount of memory space required by the AC algorithm. The architectures presented in [10], [11], and [12] are based on the highly efficient Shift-OR algorithm [13]. A pattern boundary vector is adopted in [10] and [11] while parallel shift registers are used in [12] so that multiple patterns can be handled simultaneously. There are other interesting architectures. A good summary of various architectures and their performance can be found in [9]. The architecture based on the Shift-OR algorithm will be reviewed in Section 2 because our design bears some resemblance to it.

Since security attack signatures can be better specified with regular expressions, there is increasing demand of high-speed hardware accelerators for regular expression matching. It is well known that a regular expression can be recognized with a nondeterministic finite automaton (NFA), which is equivalent to a deterministic finite automaton (DFA). Therefore, all hardware accelerators were designed either based on NFA or DFA. In [14], it was shown that an NFA can be efficiently realized with programmable logic array. A high-performance space-efficient FPGA-based implementation of NFA was proposed in [15]. In this design, the NFA is directly converted into logic gates and registers. The drawback of such a design is that the circuit has to be resynthesized when the regular expression is changed. A DFA-based implementation was presented in [16]. It achieves significant improvement in performance but may require large memory space. In [17], a Delayed Input DFA ($\mathrm{D}^2\mathrm{FA}$), which uses default transitions, an idea similar to the failure transition of the AC algorithm, was proposed to reduce the number of state transitions and hence the space requirement of a DFA. The pattern matching engine of this scheme uses memory rather than logic. A reduction of state transitions for more than 95 percent was achieved with different sets of regular expressions used in real products. Therefore, the number of expressions that can be supported by a single chip is largely increased. Although the idea works for selected sets of regular expressions, it still has the risk of resulting in a huge number of states.

In this paper, we present a different approach to implement an NFA. The pattern matching engine of our proposed architecture uses memory, which is more desirable than logic circuit because it provides better programmability. Our implementation is for the Glushkov NFA (G-NFA) [19]. We show that the implementation can handle special symbols commonly used in extended regular expressions such as

• *The author is with the Department of Communication Engineering, National Chiao Tung University, Hsinchu 300, Taiwan, R.O.C. E-mail: tlee@banyan.cm.nctu.edu.tw.*

Manuscript received 6 Dec. 2006; revised 13 July 2008; accepted 28 July 2008; published online 6 Aug. 2008.
Recommended for acceptance by M. Gokhale.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0454-1206.
Digital Object Identifier no. 10.1109/TC.2008.145.

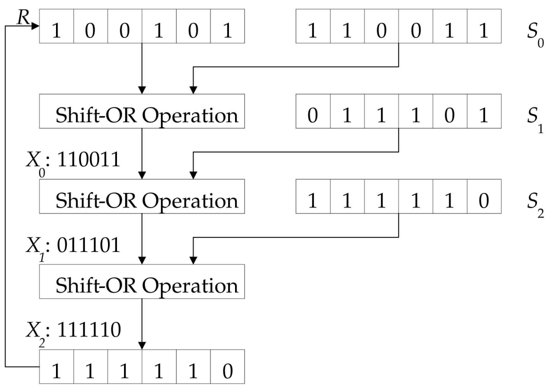


Fig. 1. The Shift-OR architecture for plain string matching.

those conforming to the POSIX 1003.2 format [20]. To achieve high performance, we generalize the implementation so that multiple symbols are processed in an operation cycle.

The rest of this paper is organized as follows: In Section 2, we review the architecture based on the Shift-OR algorithm for plain string matching. Sections 3 and 4 contain, respectively, the construction procedure of the G-NFA and inductions of some functions needed in translating an extended regular expression into a G-NFA. The proposed bitmap-based architecture for the G-NFA is presented in Section 5. Generalization for the K-step NFA, where K symbols are processed in each operation cycle, is provided in Section 6. Two example regular expressions are studied in Section 7. Finally, we draw conclusion in Section 8.

2 The Shift-OR Architecture

In this section, we briefly review the Shift-OR algorithm and the architecture proposed in [10]. We only describe the architecture that matches a single pattern. Multiple patterns can be handled simultaneously by cascading the patterns [10] or using multiple shift registers [12].

Let $P = p_0 \dots p_{N-1}$ be the pattern and $T = \dots t_{i+j} \dots$ be the input string. A state vector $R = R[0]R[1] \dots R[N-1]$ is maintained during the scanning process such that, after t_{i+j} is processed, $R[i] = 0$ if $t_i \dots t_{i+j}$ matches $p_0 \dots p_j$ or 1 otherwise. In addition to the state vector, an array of symbol position vectors is required by the Shift-OR algorithm. Let $S_c = S_c[0]S_c[1] \dots S_c[N-1]$ denote the position vector for symbol c such that $S_c[i] = 0$ if $p_i = c$ or 1 otherwise. Given $R[j]$ after t_{i+j} is processed, we have, after processing t_{i+j+1},

$$R[0] = S_c[0],$$
$$R[j+1] = R[j] \text{ OR } S_c[j+1],$$

where $c = t_{i+j+1}$. A pattern occurrence, which ends at position k of input string T, is found if $R[N-1] = 0$ after t_k is processed.

Fig. 1 shows the single-pattern version of the Shift-OR-based architecture proposed in [10]. In this figure, we assume that three symbols are processed in one operation cycle. The operation is given as follows: First, the current state vector is shifted to the right by 1 bit and bitwise ORed with S_0 to generate intermediate state vector X_0. Second, X_0 is shifted to the right by 1 bit and ORed with S_1 to obtain intermediate state vector X_1. Finally, X_1 is shifted to the right by 1 bit and ORed with S_2 to obtain intermediate state vector X_2, which represents the new state vector for the next cycle computation and is stored back to R. The contents of intermediate state vectors can be expressed as the following equations:

$$X_k[0] = Sk[0] \text{ for all } k,$$
$$X_0[i] = R[i-1] \text{ OR } S_0[i] \text{ for } i > 0,$$
$$X_k[i] = S_k[i] \text{ OR } X_{k-1}[i-1] \text{ for } k > 0, i > 0.$$

It is clear that multiple matches are possible if more than one symbol is processed in an operation cycle. To detect all the matches, the rightmost bit of each intermediate state vector is checked. A pattern occurrence, which ends at the $(k+1)\text{th}$ symbol processed in the current operation cycle, is found if $X_k[N-1] = 0$.

The number of distinct symbols appeared in the pattern could be much smaller than the size of the alphabet. In this case, a symbol encoder can be used to reduce the number of symbol position vectors stored [12]. If no multiport memory is used, then K symbol encoders are needed if K symbols are to be encoded simultaneously.

3 Construction of Glushkov-NFA

Let Σ denote the alphabet and consider a regular expression RE that consists of N symbols in Σ. Let $L(RE)$ represent the language defined by RE. To construct the G-NFA that recognizes all strings belonging to $L(RE)$ (for brevity, we say the NFA recognizes RE), the positions of the symbols in RE are marked, counting only symbols, i.e., excluding special symbols such as (,), $\bullet$ (concatenation), | (or), and $*$ (Kleene star). Denote the marked expression by $\widehat{RE}$ and let $L(\widehat{RE})$ represent its language. As an example, if $RE = (\text{AB}|\text{CA})(\text{ADB}|\text{CEF})^*$, then we have $\widehat{RE} = (\text{A}_1\text{B}_2|\text{C}_3\text{A}_4)(\text{A}_5\text{D}_6\text{B}_7|\text{C}_8\text{E}_9\text{F}_{10})^*$ and $L(\widehat{RE}) = \{\text{A}_1\text{B}_2, \text{C}_3\text{A}_4, \text{A}_1\text{B}_2\text{A}_5\text{D}_6\text{B}_7, \text{A}_1\text{B}_2\text{C}_8\text{E}_9\text{F}_{10}, \text{C}_3\text{A}_4\text{A}_5\text{D}_6\text{B}_7, \text{C}_3\text{A}_4\text{C}_8\text{E}_9\text{F}_{10}, \dots\}$. Let $\text{Pos}(\widehat{RE})$ be the set of positions in $\widehat{RE}$ and $\hat{\Sigma}$ the marked symbol alphabet. Since RE consists of N symbols, we have $\text{Pos}(\widehat{RE}) = \{1, 2, \dots, N\}$. The G-NFA is first built for the marked expression $\widehat{RE}$ and then for RE by erasing the position indices of all the symbols.

To construct a G-NFA that recognizes $\widehat{RE}$, we build $N+1$ states labeled from 0 to N, where state 0 denotes the initial state. We need to know which positions can be entered from state i when a new symbol σ is processed. To answer this question, the following definitions are necessary. In these definitions, σ_k represents the indexed symbol of $\widehat{RE}$ at position k and $\hat{\Sigma}^*$ denotes the set of all strings of symbols in $\hat{\Sigma}$.

Definition 1. *$First(\widehat{RE}) = \{x \in \text{Pos}(\widehat{RE}), \exists\, u \in \hat{\Sigma}^*, \sigma_x u \in L(\widehat{RE})\}$. In our example, $\widehat{RE} = (\text{A}_1\text{B}_2|\text{C}_3\text{A}_4)(\text{A}_5\text{D}_6\text{B}_7|\text{C}_8\text{E}_9\text{F}_{10})^*$, and thus, we have $First(\widehat{RE}) = \{1, 3\}$.*

Definition 2. *$Last(\widehat{RE}) = \{x \in \text{Pos}(\widehat{RE}), \exists\, u \in \hat{\Sigma}^*, u\,\sigma_x \in L(\widehat{RE})\}$. For convenience, state x is called a final state if $x \in Last(\widehat{RE})$. In our example, we have $Last(\widehat{RE}) = \{2, 4, 7, 10\}$.*

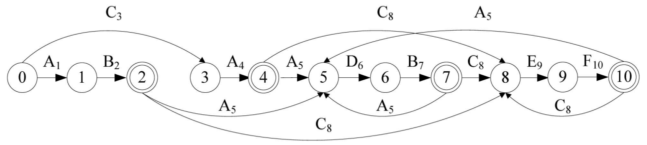


Fig. 2. The G-NFA for $\widehat{RE} = (\mathrm{A_1B_2|C_3A_4})(\mathrm{A_5D_6B_7|C_8E_9F_{10}})^*$.

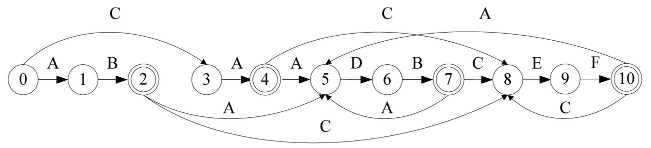


Fig. 3. The G-NFA for $RE = (\mathrm{AB|CA})(\mathrm{ADB|CEF})^*$.

Definition 3. *$Follow(\widehat{RE}, x) = \{y \in \mathrm{Pos}(\widehat{RE}), \exists\ u, v \in \hat{\Sigma}^*, u\sigma_x\sigma_y v \in L(\widehat{RE})\}$. In our example, we have $Follow(\widehat{RE}, 2) = Follow(\widehat{RE}, 7) = \{5, 8\}$.*

The G-NFA for $\widehat{RE}$, denoted by $M_{\widehat{RE}}$, is given by $M_{\widehat{RE}} = (S, \hat{\Sigma}, I, F, \hat{\delta})$, where $S = \{0, 1, \ldots, N\}$ is the set of states, $\hat{\Sigma}$ represents the marked symbol alphabet, $I = \{0\}$ denotes the set of initial state, $F = Last(\widehat{RE})$ is the set of final states, and $\hat{\delta}$ is the state transition function defined by $\forall\ x \in S, \forall\ y \in Follow(\widehat{RE}, x), \hat{\delta}\,(x, \sigma_y) = \{y\}$. One can easily construct $M_{\widehat{RE}}$ as long as $First(\widehat{RE})$, $Last(\widehat{RE})$, and $Follow(\widehat{RE}, x)$ are known. The G-NFA for $\widehat{RE} = (\mathrm{A_1B_2|C_3A_4})(\mathrm{A_5D_6B_7|C_8E_9F_{10}})^*$ is shown in Fig. 2, where every final state is represented by double circle.

As mentioned before, the G-NFA of the original unmarked regular expression, denoted by $M_{RE} = (S, \Sigma, I, F, \delta)$, can be obtained by erasing the position indices in the marked automaton. The major differences between M_{RE} and $M_{\widehat{RE}}$ are 1) Σ is for unmarked symbols and 2) the state transition function δ is defined by $y \in \delta(x, \sigma)$ if $y \in \hat{\delta}(x, \sigma_y)$ and $\sigma_y = \sigma$ if the index y is removed. Fig. 3 illustrates the G-NFA of our example regular expression $RE = (\mathrm{AB|CA})(\mathrm{ADB|CEF})^*$. Note that, in Fig. 3, there is an edge from state x to state y that is labeled with σ if $y \in \delta(x, \sigma)$.

Before leaving this section, we state some well-known properties of the G-NFA.

Property 1. *The G-NFA is ε-free, i.e., there is no ε-transitions, where ε represents the empty string.*

Property 2. *For any state y, if $\hat{\delta}(x, \alpha) = \{y\}$, then it is true that $\alpha = \sigma_y$.*

4 Inductions of $First(\overline{RE})$, $Last(\overline{RE})$, and $Follow(\overline{RE}, x)$

As mentioned previously, the G-NFA for regular expression RE can be constructed if $First(RE)$, $Last(RE)$, and $Follow(RE, x)$ are known. In this section, we present the inductions of $First(\overline{RE})$, $Last(\overline{RE})$, and $Follow(\overline{RE}, x)$, where $\overline{RE} = RE_1|RE_2$, $RE_1 \bullet RE_2$, or RE^*. These results can be found in [21].

Consider the inductions of $First(\overline{RE})$ and $Last(\overline{RE})$. We have:

$\overline{RE} = RE_1|RE_2$: $First(\overline{RE}) = First(RE_1) \cup First(RE_2)$;
$Last(\overline{RE}) = Last(RE_1) \cup Last(RE_2)$.
$\overline{RE} = RE_1 \bullet RE_2$: $First(\overline{RE}) = First(RE_1) \cup First(RE_2)$ if $\varepsilon \in L(RE_1)$ or $First(RE_1)$ otherwise;
$Last(\overline{RE}) = Last(RE_1) \cup Last(RE_2)$ if $\varepsilon \in L(RE_2)$ or $Last(RE_2)$ otherwise.
$\overline{RE} = RE^*$: $First(\overline{RE}) = First(RE)$; $Last(\overline{RE}) = Last(RE)$.

Induction of $Follow(\overline{RE}, x)$ is given as follows:

$\overline{RE} = RE_1|RE_2$: $Follow(\overline{RE}, x) = Follow(RE_1, x)$ if $x \in \mathrm{Pos}(RE_1)$ or $Follow(RE_2, x)$ if $x \in \mathrm{Pos}(RE_2)$.
$\overline{RE} = RE_1 \bullet RE_2$: $Follow(\overline{RE}, x) = Follow(RE_1, x)$ if $x \in \mathrm{Pos}(RE_1) - Last(RE_1)$ or $Follow(RE_1, x) \cup First(RE_2)$ if $x \in Last(RE_1)$ or $Follow(RE_2, x)$ if $x \in \mathrm{Pos}(RE_2)$.
$\overline{RE} = RE^*$: $Follow(\overline{RE}, x) = Follow(RE, x)$ if $x \in \mathrm{Pos}(RE)$-$Last(RE)$ or $Follow(RE, x) \cup First(RE)$ if $x \in Last(RE)$.

The following inductions answer whether or not $\varepsilon \in L(\overline{RE})$.

$\overline{RE} = RE_1|RE_2$: $\varepsilon \in L(\overline{RE})$ if $\varepsilon \in L(RE_1)$ or $\varepsilon \in L(RE_2)$.
$\overline{RE} = RE_1 \bullet RE_2$: $\varepsilon \in L(\overline{RE})$ if $\varepsilon \in L(RE_1)$ and $\varepsilon \in L(RE_2)$.
$\overline{RE} = RE^*$: $\varepsilon \in L(\overline{RE})$.

Note that the above inductions can be generalized to include special symbols ? (zero or one repetition), + (one or more repetitions), and $\{min, max\}$ (minimum of *min*, maximum of *max* repetitions), which are commonly used in various extended regular expressions such as the POSIX 1003.2 format. The inductions for $\overline{RE} = RE^?$ and $\overline{RE} = RE^+$ are given below.

$\overline{RE} = RE^?$: $First(\overline{RE}) = First(RE)$; $Last(\overline{RE}) = Last(RE)$; $Follow(\overline{RE}, x) = Follow(RE, x)$; $\varepsilon \in L(\overline{RE})$.
$\overline{RE} = RE^+$: $First(\overline{RE}) = First(RE)$; $Last(\overline{RE}) = Last(RE)$; $Follow(\overline{RE}, x) = Follow(RE, x)$ if $x \in \mathrm{Pos}(RE)$-$Last(RE)$ or $Follow(RE, x) \cup First(RE)$ if $x \in Last(RE)$; $\varepsilon \in L(\overline{RE})$ if $\varepsilon \in L(RE)$.

The induction for $\overline{RE} = RE^?$ is obviously true because $RE^?$ is equivalent to $\varepsilon|RE$. The correctness of the induction for $\overline{RE} = RE^+$ can be seen as follows: Since $First(\overline{RE}) = First(RE)$; $Last(\overline{RE}) = Last(RE)$; $Follow(\overline{RE}, x) =$ $Follow(RE, x)$ if $x \in \mathrm{Pos}(RE) - Last(RE)$ or $Follow(RE, x) \cup First(RE)$ if $x \in Last(RE)$, a string T is accepted if and only if (iff) it can be written as $T_1 \bullet T_2 \bullet \cdots \bullet T_k$, where $T_i \in L(RE)$ for all i, $1 \le i \le k$, which is exactly the condition for $T \in L(RE^+)$. It is clear that $RE^+ = RE \bullet RE^*$, and therefore, one can use the inductions for $\overline{RE} = RE_1 \bullet RE_2$ and $\overline{RE} = RE^*$. However, compared with the above induction, it requires double the number of states.

Consider the induction of the bound special symbol $\{min, max\}$. Let $\overline{RE} = RE\{min, max\}$ and assume that the length of RE is equal to N. Instead of creating N states, we need to generate $maxN$ states, which are numbered from 1 to $maxN$. Partition these $maxN$ states into max equal-sized groups so that the ith group contains states $(i-1)N+1$, $(i-1)N+2, \ldots$, and iN, $1 \leq i \leq max$. The inductions for $\overline{RE} = RE\{min, max\}$ are given below.

$\overline{RE} = RE\{min, max\}$: $First(\overline{RE}) = \{(i-1)N+x, x \in First(RE), 1 \leq i \leq max\}$ if $\varepsilon \in L(RE)$ or $First(RE)$ otherwise; $Last(\overline{RE}) = \{(i-1)N+x, x \in Last(RE), min \leq i \leq max\}$; for $(i-1)N+1 \leq x \leq iN$, $1 \leq i \leq \max-1$, $Follow(\overline{RE}, x) = \{(i-1)N+y, y \in Follow(RE, x-(i-1)N)\} \cup \{iN+y, y \in First(RE)\}$ if $x-(i-1)N \in Last(RE)$ or $\{(i-1)N+y, y \in Follow(RE, x-(i-1)N)\}$ otherwise; for $(max-1)N+1 \leq x \leq maxN$, $Follow(\overline{RE}, x) = \{(max-1)N+y, y \in Follow(RE, x-(max-1)N)\}$; $\varepsilon \in L(\overline{RE})$ if $\varepsilon \in L(RE)$.

Roughly speaking, the above construction creates max copies of an NFA, which recognizes RE. For convenience, we call state x a candidate last state of the jth copy iff $(j-1)N+1 \leq x \leq jN$ and $x-(j-1)N$ is in $Last(RE)$ of the first copy, which recognizes RE. Similarly, state x is called a candidate first state of the jth copy iff $(j-1)N+1 \leq x \leq jN$ and $x-(j-1)N$ is in $First(RE)$ of the first copy. The above construction assigns $y \in Follow(\overline{RE}, x)$, where y is any candidate first state of the jth copy and x is any candidate last state of the $(j-1)$th copy. Furthermore, state x is a final state iff state x is a candidate last state of the jth copy, where j satisfies $min \leq j \leq max$. The correctness of the above inductions can be argued as follows: Assume that $\varepsilon \in L(RE)$ and consequently $\varepsilon \in L(\overline{RE})$. A string T is accepted iff it can be written as $T_1 \bullet T_2 \bullet \cdots \bullet T_k$ for some k, $min \leq k \leq max$, such that $T_i \in L(RE)$ for all i, $1 \leq i \leq k$, and $t_{j-1} = \varepsilon$ if $t_j = \varepsilon$, which is exactly the condition for $T \in L(\overline{RE})$. Therefore, the inductions are correct for the case $\varepsilon \in L(RE)$. The arguments of correctness for the case $\varepsilon \notin L(RE)$ is similar and thus is omitted.

It is easy to see that $\{min\}$ (repetitions of exactly min times) is a special case of $\{min, max\}$ with $max = min$. In case $max = \infty$, i.e., the bound special symbol is just $\{min, \}$, we need to only create min copies of the NFA, which recognizes RE with a total of $minN$ states. The inductions for this case are shown below.

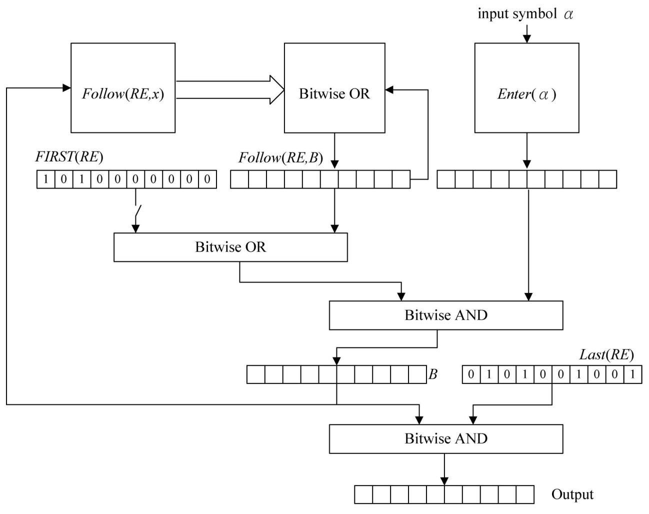
$\overline{RE} = RE\{min, \}$: $First(\overline{RE}) = \{(i-1)N+x, x \in First(RE), 1 \leq i \leq min\}$ if $\varepsilon \in L(RE)$ or $First(RE)$ otherwise; $Last(\overline{RE}) = \{(min-1)N+x, x \in Last(RE)\}$; for $(i-1)N+1 \leq x \leq iN$, $1 \leq i \leq min-1$, $Follow(\overline{RE}, x) = \{(i-1)N+y, y \in Follow(RE, x-(i-1)N)\} \cup \{iN+y, y \in First(RE)\}$ if $x-(i-1)N \in Last(RE)$ or $\{(i-1)N+y, y \in Follow(RE, x-(i-1)N)\}$ otherwise; for $(min-1)N+1 \leq x \leq iN$, $Follow(\overline{RE}, x) = \{(min-1)N+y, y \in Follow(RE, x-(i-1)N)\} \cup \{(min-1)N+y, y \in First(RE)\}$ if $x \in Last(\overline{RE})$ or $\{(i-1)N+y, y \in Follow(RE, x-(i-1)N)\}$ otherwise; $\varepsilon \in L(\overline{RE})$ if $\varepsilon \in L(RE)$.

Note that in the last copy, state $y \in Follow(\overline{RE}, x)$ if state x is a final state and state y is a candidate first state. According to the induction, a string T is accepted iff it can be written as $T_1 \bullet T_2 \bullet \cdots \bullet T_k$ for some $k \geq min$, such that $T_i \in L(RE)$ for all i, $1 \leq i \leq k$, which is exactly the condition for $T \in L(\overline{RE})$. Therefore, the above inductions are correct.

5 A Bitmap-Based Architecture

For convenience, we define $Follow(RE, 0) = First(RE)$. Let $Enter(\alpha) = \bigcup_{x \in S} \delta(x, \alpha)$. According to Property 2, it holds that $\delta(x, \alpha) = Follow(RE, x) \cap Enter(\alpha)$. Let B denote the set of active states after the last symbol of input string T is processed. The string T is accepted, i.e., $T \in L(RE)$, iff $B \cap Last(RE) \neq \emptyset$. As a consequence, one can implement M_{RE} with bitmaps and simple logical operations. Fig. 4 illustrates the architecture of the bitmap-based implementation of our running example regular expression.

The symbol $\sim$, which appears in the $Enter(\alpha)$ table, means any symbol other than A, B, C, D, E, and F. In fact, one can define an equivalence relation so that two symbols α and β are in the same equivalence class iff $Enter(\alpha) = Enter(\beta)$. In our example, there are seven equivalence classes denoted by A, B, C, D, E, F, and $\sim$. The initial content of B is set to zero. The switch connected to $First(RE)$ is closed when the first symbol is processed and then remains open. To find the active states after an input symbol α is processed, the content of B is used to fetch the bitmaps of the $Follow(RE, x)$ table. The bitmap representing $Follow(RE, x)$ is fetched iff the xth bit of B is a 1. The fetched bitmaps are bitwise ORed together and the result is bitwise ANDed with $Enter(\alpha)$ to obtain the updated content of B. Let $Follow(RE, X) = \bigcup_{x \in X} Follow(RE, x)$ for all $X \subset S$. As a result, the updated set of active states after input symbol α is processed is $Follow(RE, B) \cap Enter(\alpha)$. In Fig. 4, the content of $Follow(RE, B)$ is reset to zero before an input symbol is processed. Note that the $Follow(RE, x)$ table may have to be accessed up to N times if all bits of B are 1's. It is possible to reduce this number by precomputation. For example, if the length of RE is equal to 32, then the number of memory accesses and bitwise OR operations could be as many as 32. To reduce this number, one can partition the states into groups and precompute unions of $Follow(RE, x)$ for all possible combinations. Assume that the states are partitioned into four groups so that group i $(1 \leq i \leq 4)$ contains states $8(i-1)+1, 8(i-2)+2, \ldots$, and $8i$. As a consequence, there are 256 combinations within each group. We can store $Follow(RE, X)$ for all possible values of X. As an example, consider the first group. If $X = 19 = (1\,1\,0\,0\,1\,0\,0\,0)$ (states 1, 2, and 5 are active), then we have $Follow(RE, X) = Follow(RE, 1) \cup Follow(RE, 2) \cup Follow(RE, 5)$. By doing so, the number of memory accesses and bitwise OR operations is reduced to four. The trade-off is an increase of memory requirement by 32 times. To further improve system performance, the four groups can be stored in separate memories and fetched simultaneously. After B is updated, a bitwise AND operation is performed for B and $Last(RE)$. The operation repeats until all the symbols of input string T are processed. To decide whether $T \in L(RE)$ or not, we examine the Output register after the last symbol of input string T is processed. The input string T



(a)

State x	$Follow(RE,x)$
1	0 1 0 0 0 0 0 0 0 0
2	0 0 0 0 1 0 0 1 0 0
3	0 0 0 1 0 0 0 0 0 0
4	0 0 0 0 1 0 0 1 0 0
5	0 0 0 0 0 1 0 0 0 0
6	0 0 0 0 0 0 1 0 0 0
7	0 0 0 0 1 0 0 1 0 0
8	0 0 0 0 0 0 0 0 1 0
9	0 0 0 0 0 0 0 0 0 1
10	0 0 0 0 1 0 0 0 0 1

(b)

Symbol α	$Enter(\alpha)$
A	1 0 0 1 1 0 0 0 0 0
B	0 1 0 0 0 0 1 0 0 0
C	0 0 1 0 0 0 0 1 0 0
D	0 0 0 0 0 1 0 0 0 0
E	0 0 0 0 0 0 0 0 1 0
F	0 0 0 0 0 0 0 0 0 1
~	0 0 0 0 0 0 0 0 0 0

(c)

Fig. 4. (a) The bitmap-based architecture for $RE = (\text{AB}|\text{CA})(\text{ADB}|\text{CEF})^*$. (b) The $Follow(RE, x)$ table. (c) The $Enter(\alpha)$ table.

is accepted iff the final content of Output register is not zero. Note that since the Output register is updated after each input symbol is processed, the proposed architecture can actually detect the ending positions of all substrings of T, which start from the first symbol of T and belong to $L(RE)$.

It is not hard to see that the above implementation for M_{RE} does not consider $\varepsilon \in L(RE)$. The architecture can be easily modified by adding another bit (for state 0) to each bitmap and performing the bitwise AND operation before updating the content of B to include the possibility of $\varepsilon \in L(RE)$. The initial content of B is set to 1 for the bit representing state 0 and 0 elsewhere. Also, the implementation only detects all substrings, which start from the first symbol of input string T and belong to $L(RE)$. Let $\bar{M}_{RE} = (S', \Sigma, I', F', \delta')$ be the NFA, which can detect all substrings of T that start from any symbol of T and belong to $L(RE)$. Clearly, one can obtain $\bar{M}_{RE}$ from M_{RE} by letting the initial state to be always active. Consequently, we have $S' = S$, $I' = I$, $F' = F$, and for every $\alpha \in \Sigma$, $\delta'(x, \alpha) = \delta(x, \alpha)$ for all $x \in S$, $x \neq 0$ and $\delta'(0, \alpha) = \delta(0, \alpha) \cup \{0\}$. To realize $\bar{M}_{RE}$, the switch connected to $First(RE)$ is always closed.

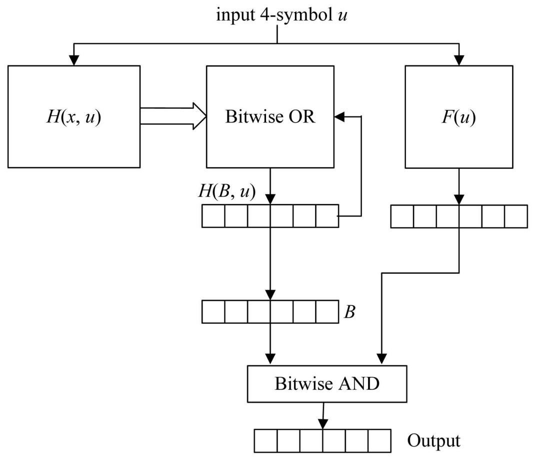
6 A High-Performance Bitmap-Based Architecture

In this section, we generalize the architecture so that $K(\geq 2)$ symbols are processed in each operation cycle. For an integer $d \geq 2$, let $M^d_{RE} = (S_d, \Sigma^d, I_d, S_d \times \Sigma^d, \delta_d)$ denote the d-step NFA, which processes d symbols per operation cycle and accepts all substrings of input string T, which start from the first symbol of T and belong to $L(RE)$. Here, $\Sigma^d = \{u \in \Sigma^*, |u| = d\}$, where $|u|$ represents the length of u. For convenience, we call u a d-symbol if $u \in \Sigma^d$. The state transition function δ_d is defined by $\delta_d(x, ua) = \delta(\delta_{d-1}(x, u), a)$, for all $x \in S$, $ua \in \Sigma^d$, and $a \in \Sigma$, where δ is generalized to become $\delta(X, a) = \bigcup_{x \in X} \delta(x, a)$ for all $X \subset S$. For two states x and y in M_{RE}, we say state y can be accessed by state x in d steps if $y \in \delta_d(x, u)$ for some d-symbol u. The set S_d is a subset of S such that $0 \in S_d$ and $x \in S_d$ if $x \in S$ and can be accessed by state 0 in qd steps for some integer q. The set I_d is the same as I. Different from M_{RE}, the current state is not sufficient for M^d_{RE} to decide whether or not a substring of T, which starts from the first symbol of T, belongs to $L(RE)$. Instead, we need to know the current state and the input d-symbol. Hence, the set of final states F in M_{RE} is replaced by $S_d \times \Sigma^d$ in M^d_{RE}. For $x \in S_d$ and a d-symbol $u = u_1 u_2 \ldots u_d$, the pair $(x, u) \in S_d \times \Sigma^d$ iff $\delta_i(x, u_1 \ldots u_i) \cap Last(RE) \neq \emptyset$ for some i, $1 \leq i \leq d$. Note that it is possible to find multiple matches with current state x and input d-symbol u. From x and u, we are able to determine the number of matches and their ending positions as long as M_{RE} is available. One can easily prove that M^d_{RE} is able to find all substrings of T, which start from the first symbol of T and belong to $L(RE)$. For the purpose of processing K symbols per operation cycle, we need M^K_{RE}.

The NFA $\bar{M}^K_{RE}$ has to be modified if the goal is to find all substrings of input string T, which start from any symbol of T and belong to $L(RE)$. Let $\bar{M}^K_{RE} = (S'_K, \Sigma^K, I'_K, S'_K \times \Sigma^K, \delta'_K)$ denote such an NFA. For $K \geq 2$, it is not true to obtain $\bar{M}^K_{RE}$ from M^K_{RE} by letting the initial state to be always active because, by doing so, one can only detect all substrings of T, which start at the $(qK + 1)$th symbol of T and belong to $L(RE)$. To have a correct $\bar{M}^K_{RE}$, we need to assign $S'_K = S$; $I'_K = I_K$; $\delta'_K(x, u_1 u_2 \ldots u_K) = \delta_K(x, u_1 u_2 \ldots u_K)$ for all $x \in S$, $x \neq 0$, $\delta'_K(0, u_1 u_2 \ldots u_K) = \bigcup_{i=1}^{K} \delta_i(0, u_{K-i+1} \ldots u_K) \cup \{0\}$ for all $u_1 u_2 \ldots u_K \in \Sigma^K$; and $(x, u_1 u_2 \ldots u_K) \in S'_K \times \Sigma^K$ if $\delta_i(x, u_1 u_2 \ldots u_i) \cap Last(RE) \neq \emptyset$ for some i, $1 \leq i \leq K$. The correctness of $\bar{M}^K_{RE}$ can be argued as follows: Let $T = t_1 t_2 \ldots t_L$ and $T_1 = t_e \ldots t_f$, where $e = q_1 K + r_1$, $f = q_2 K + r_2$, $0 \leq r_1$, $r_2 \leq K - 1$, is a substring of T, which belongs to $L(RE)$. Assume that L is an integral multiple of K. We will handle the case when L is not an integral multiple of K later. Let us consider the case that $r_1, r_2 > 0$. (The other cases can be argued similarly.) Since $T_1 \in L(RE)$, M_{RE} accepts T_1, and therefore, there is a sequence of state transitions, which ends at a final state when the symbols of T_1 are completely processed. Let x be the state in the sequence of transitions after the $(K - r_1 + 1)$th symbol of T_1 is processed. Also, let y be the state in the sequence of transitions after the $((q_2 - q_1 - 1)K + K - r_1 + 1)$th symbol of T_1 is processed. Consider the NFA $\bar{M}^K_{RE}$. With our assignment of $\delta'_K(0, u_1 u_2 \ldots u_K)$, state x is active after the $(q_1 + 1)$th K-symbol is processed. Moreover, state y is active after the subsequent $(q_2 - q_1 - 1)$ K-symbols are processed. Finally, when the $(q_2 + 1)$th K-symbol is processed, a match is found and the substring T_1 can be detected if M_{RE} is available.

For $\bar{M}^K_{RE}$, define $Enter(u)$ as the set of states in S'_K, which can be entered after processing input K-symbol u and $Follow(RE, x)$ as the set of states in S'_K such that $y \in Follow(RE, x)$ if there exists a K-symbol v such that $y \in \delta'_K(x, v)$. Since state 0 is always active, we assign $x \in Enter(u)$ for any K-symbol u if $x \in \delta'_K(0, u)$ The equality $\delta'_K(x, u) = Follow(RE, x) \cap Enter(u)$ for any pair of state x and input K-symbol u is in general not true for $\bar{M}^K_{RE}$ when $K > 1$. As an example, for the running example regular expression with $K = 4$, we have $Follow(RE, 0) = \{0, 1, 2, 3, 4, 5, 6, 8\}$, $Enter(EFAD) = \{0, 6\}$, and $Follow(RE, 0) \cap Enter(EFAD) = \{0, 6\}$, which is different from $\delta'_4(0, EFAD) = \{0\}$. As a consequence, the architecture shown in Fig. 4 is not applicable and we need a bitmap table $H(x, u) = \delta'_4(x, u)$ for all pairs of state x and 4-symbol u.

The bitmap-based implementation of $\bar{M}^4_{RE}$ for $RE = (\mathrm{AB}|\mathrm{CA})(\mathrm{ADB}|\mathrm{CEF})^*$ is shown in Fig. 5a. In addition to the $H(x, u)$ table, we need another bitmap table for $F(u)$ whose xth bit is a 1 iff $(x, u) \in S'_4 \times \Sigma^4$. For convenience, the bitmaps are replaced with a set of integers in Fig. 5b. To save space, $H(x, u)$ is not presented. Since the total number of possible K-symbols could be huge, it is important to define equivalence classes for them. For our purpose, two K-symbols u and v are in the same equivalence class iff $H(x, u) = H(x, v)$ for all states x and $F(u) = F(v)$. As illustrated in Fig. 5b, there are 58 equivalence classes, which are represented by different integers called equivalence class ID (ECID). These equivalence classes are partitioned into five groups: Group 1 (ECIDs 1-14), Group 2 (ECIDs 15-21), Group 3 (ECIDs 22-27), Group 4 (ECIDs 28-57), and Group 5 (ECID 58). For ease of description, ~ represents any symbol. Moreover, a K-symbol, which contains at least one ~ , is called a generalized K-symbol. The ECID of the equivalence class, which contains the most specific (generalized) K-symbol, is selected if an input K-symbol matches multiple (generalized) K-symbols in different equivalence classes. A (generalized) K-symbol u is said to be more specific than another generalized K-symbol v if v can be obtained from u by changing one or more symbols, which are not ~ into ~. For example, the input 4-symbol ADBA matches the (generalized) 4-symbols in equivalence classes 5, 15, 19, 22, and 58. ECID 5 is selected because ADBA is more specific than ADB~ , A~~~ , ~~~A, and ~~~~ . Symbol u_i of the (generalized) 4-symbol $u_1 u_2 u_3 u_4$ is underlined if $\delta_i(x, u_1 u_2 \ldots u_i) \cap Last(RE) \neq \emptyset$ for some state $x \in S$. (To be precise, we define $\delta(x, \sim) = \{0\}$ for all states $x \in S$.)



(a)

ECID	(generalized) 4-symbols	$F(u)$
1	$\mathrm{A\underline{B}AD}$, $\mathrm{C\underline{A}AD}$	0
2	$\mathrm{A\underline{B}CE}$, $\mathrm{C\underline{A}CE}$	0
3	$\mathrm{\underline{B}AD\underline{B}}$	1, 6
4	$\mathrm{\underline{B}CE\underline{F}}$	1, 6
5	$\mathrm{AD\underline{B}A}$, $\mathrm{CE\underline{F}A}$	2, 4, 7, 10
6	$\mathrm{AD\underline{B}C}$, $\mathrm{CE\underline{F}C}$	2, 4, 7, 10
7	$\mathrm{AAD\underline{B}}$	3
8	$\mathrm{\underline{A}CE\underline{F}}$	3
9	$\mathrm{D\underline{B}AD}$	5
10	$\mathrm{D\underline{B}CE}$	5
11	$\mathrm{E\underline{F}AD}$	8
12	$\mathrm{E\underline{F}CE}$	8
13	$\mathrm{\underline{F}AD\underline{B}}$	9
14	$\mathrm{\underline{F}CE\underline{F}}$	9
15	$\mathrm{AD\underline{B}}$~, $\mathrm{CE\underline{F}}$~	2, 4, 7, 10

Fig. 5a. The bitmap architecture of $\bar{M}^4_{RE}$ for $RE = (\mathrm{AB}|\mathrm{CA})(\mathrm{ADB}|\mathrm{CEF})^*$.

A 4-symbol u is in Group 1 iff it satisfies $\delta_4(x, u) \cap S \neq \emptyset$ for some state $x \in S$. Every generalized 4-symbol in Group 2 contains at least one ~ at the end. A generalized 4-symbol $u = u_1 u_2 \ldots u_i \sim \ldots \sim$, where $u_1 u_2 \ldots u_i \in \Sigma^i$, is in Group 2 iff $\delta_i(x, u_1 \ldots u_i) \cap Last(RE) \neq \emptyset$ for some state $x \in S$. Note that, for a generalized 4-symbol u in Group 2, $H(x, u) = \{0\}$ for all states x. Besides, for u in Group 1 and v in Group 2, we have $F(v) \subset F(u)$ if u is more specific than v. The generalized 4-symbols in Group 3 contain at least one ~ at the beginning and are necessary for the states that can be accessed by state 0 in less than four steps. A generalized 4-symbol $u = \sim \ldots \sim u_i \ldots u_4$ is in Group 3 iff $\delta_{5-i}(0, u_i \ldots u_4) \cap S \neq \emptyset$. For a generalized 4-symbol u in Group 3, $F(u)$ is either $\emptyset$ or $\{0\}$. Moreover, for a (generalized) 4-symbol u in Group 1 or Group 3 and another generalized 4-symbol v in Group 3, we have $H(x, v) \subset H(x, u)$ for every state x and $F(v) \subset F(u)$ if u is more specific than v. The equivalence classes that form Group 4 are obtained by "intersecting" the equivalence classes of Group 2 with those of Group 3. Consider a

16	$\mathrm{A\underline{B}\sim\sim, C\underline{A}\sim\sim}$	0
17	$\mathrm{D\underline{B}\sim\sim}$	5
18	$\mathrm{E\underline{F}\sim\sim}$	8
19	$\mathrm{\underline{A}\sim\sim\sim}$	3
20	$\mathrm{\underline{B}\sim\sim\sim}$	1, 6
21	$\mathrm{\underline{F}\sim\sim\sim}$	9
22	$\mathrm{\sim\sim\sim A}$	$\emptyset$
23	$\mathrm{\sim\sim\sim C}$	$\emptyset$
24	$\mathrm{\sim\sim A\underline{B}}$	0
25	$\mathrm{\sim\sim C\underline{A}}$	0
26	$\mathrm{\sim A\underline{B}A, \sim C\underline{A}A}$	0
27	$\mathrm{\sim A\underline{B}C, \sim C\underline{A}C}$	0
28	$\mathrm{A\underline{B}\sim A, C\underline{A}\sim A}$	0
29	$\mathrm{A\underline{B}\sim C, C\underline{A}\sim C}$	0
30	$\mathrm{A\underline{B}A\underline{B}, C\underline{A}A\underline{B}}$	0
31	$\mathrm{A\underline{B}C\underline{A}, C\underline{A}C\underline{A}}$	0
32	$\mathrm{D\underline{B}\sim A}$	5
33	$\mathrm{D\underline{B}\sim C}$	5
34	$\mathrm{D\underline{B}A\underline{B}}$	0, 5
35	$\mathrm{D\underline{B}C\underline{A}}$	0, 5
36	$\mathrm{E\underline{F}\sim A}$	8
37	$\mathrm{E\underline{F}\sim C}$	8
38	$\mathrm{E\underline{F}A\underline{B}}$	0, 8
39	$\mathrm{E\underline{F}C\underline{A}}$	0, 8
40	$\mathrm{\underline{A}\sim\sim A}$	3
41	$\mathrm{\underline{A}\sim\sim C}$	3
42	$\mathrm{\underline{A}\sim A\underline{B}}$	3
43	$\mathrm{\underline{A}\sim C\underline{A}}$	3
44	$\mathrm{\underline{A}A\underline{B}A, \underline{A}C\underline{A}A}$	0, 3
45	$\mathrm{\underline{A}A\underline{B}C, \underline{A}C\underline{A}C}$	0, 3
46	$\mathrm{\underline{B}\sim\sim A}$	1, 6
47	$\mathrm{\underline{B}\sim\sim C}$	1, 6
48	$\mathrm{\underline{B}\sim A\underline{B}}$	0, 1, 6
49	$\mathrm{\underline{B}\sim C\underline{A}}$	0, 1, 6
50	$\mathrm{\underline{B}A\underline{B}A, \underline{B}C\underline{A}A}$	0, 1, 6
51	$\mathrm{\underline{B}A\underline{B}C, \underline{B}C\underline{A}C}$	0, 1, 6
52	$\mathrm{\underline{F}\sim\sim A}$	9
53	$\mathrm{\underline{F}\sim\sim C}$	9
54	$\mathrm{\underline{F}\sim A\underline{B}}$	0, 9
55	$\mathrm{\underline{F}\sim C\underline{A}}$	0, 9
56	$\mathrm{\underline{F}A\underline{B}A, \underline{F}C\underline{A}A}$	0, 9
57	$\mathrm{\underline{F}A\underline{B}C, \underline{F}C\underline{A}C}$	0, 9
58	$\sim\sim\sim\sim$	$\emptyset$

(b)

Fig. 5b. The $F(u)$ table.

generalized 4-symbol $u = u_1 \dots u_i \sim \dots \sim$ in Group 2 and another generalized 4-symbol $v = \sim \dots \sim v_j \dots v_4$ in Group 3. A generalized 4-symbol $w = u_1 \dots u_i \sim \dots \sim v_j \dots v_4$ is created in Group 4 if $j - i > 1$. If $j - i = 1$, then the 4-symbol $w = u_1 \dots u_i\, v_j \dots v_4$ is created in Group 4 if it does not appear in Group 1. It is worth to be pointed out that $H(x, w) = H(x, v)$ and $F(w) = F(u) \cup F(v)$ if w (in Group 4) is created by intersecting u (in Group 2) and v (in Group 3). For example, DBAB (ECID 34) is derived from $\mathrm{DB}{\sim}{\sim}$ (ECID 17) and $\sim\sim\mathrm{AB}$ (ECID 24) and thus $H(x, \mathrm{DBAB}) = H(x, \sim\sim\mathrm{AB})$ and $F(\mathrm{DBAB}) = F(\mathrm{DB}{\sim}{\sim}) \cup F(\sim\sim\mathrm{AB})$. Group 5 only contains one generalized 4-symbol, i.e., $\sim\sim\sim\sim$, and represents the complement of the other groups.

The operation of the NFA engine shown in Fig. 5a is given as follows: The bitwise AND operation for B and $F(u)$ is performed before updating the content of B. Matches are found if the outcome is not zero. The initial content of B is set to 1 for the bit representing state 0 and 0 elsewhere. To update the set of active states after input K-symbol u is processed, the content of B and the ECID of u are used to fetch the bitmaps of $H(x, u)$ and obtain $H(B, u) = \bigcup_{x \in B} H(x, u)$ with the bitwise OR operation. The result of $H(B, u)$ is then saved as new content of B. Note that the content of $H(B, u)$ is reset to zero before processing an input K-symbol. The operation repeats until all the K-symbols of the input string T are processed. Clearly, the length of input string T may not be an integral multiple of K. This case will be handled later.

The hierarchical architecture proposed in [5], which is shown in Fig. 6, can be used to find the ECID of an input K-symbol. In this figure, for the input 4-symbol $u = u_1 u_2 u_3 u_4$, table $\mathrm{C}(1, i)$, $i = 1, 2, 3$, and 4, is used to find the ECID of u_i. Table $\mathrm{C}(2, i)$, $i = 1, 2$, is used to find the ECID of $u_{2i-1} u_{2i}$ using the ECIDs of u_{2i-1} and u_{2i} as inputs. Finally, table C(4,1) is used to find the ECID of u using the ECIDs of $u_1 u_2$ and $u_3 u_4$ as inputs. Fig. 6 also shows the pipelined architecture of the overall NFA, which includes the component of finding the ECID for the input 4-symbol and the NFA engine. With the pipelined design, the throughput performance can be improved by four times.

As mentioned previously, the length of input string T may not be an integral multiple of K. Let the length of T be $qK + r$, $0 \le r \le K - 1$. There is no problem if $r = 0$. Assume that $r > 0$ and let $u = u_1 \dots u_r$ be the last r symbols of T. A simple solution is to pad $(K - r)$ symbols at the end of u. For example, one can pad $(K - r)$ z's to make $u\, z^{K-r}$ a K-symbol, where z^d means symbol z repeats d times. Since the active states after $u\, z^{K-r}$ is processed is irrelevant, all we need to modify is the set $S'_K \times \Sigma^K$. The pair $(x, u\, z^{K-r})$ is added to $S'_K \times \Sigma^K$ if $\delta_i\, (x, u_1 \dots u_i) \cap Last(RE) \neq \emptyset$ for some i, $1 \le i \le r$. It is possible to create false positives if $(x, u\, z^{K-r})$ was in $S'_K \times \Sigma^K$ and $\delta_i\, (x, u\, z^j) \cap Last(RE) \neq \emptyset$ for some j, $1 \le i \le K - r$. However, the false positives can be eliminated because we know the value of r.

Let us now compare the complexity of the Shift-OR architecture for plain string matching with that of our proposed architecture for regular expression matching. The comparison is for a single pattern of length N. We will emphasize on comparison of memory space requirements. Consider the architectures that process one symbol in each operation cycle. For the Shift-OR architecture, it requires one N-bit register for the state vector R, $|\Sigma| \times N$ ($|\Sigma|$ denotes the size of Σ) bits memory space to store symbol position vectors, and N OR gates. For our proposed architecture, it requires five N-bit registers for $First(RE)$, $Last(RE)$, B, $Follow(RE, B)$, and Output, N^2 bits memory

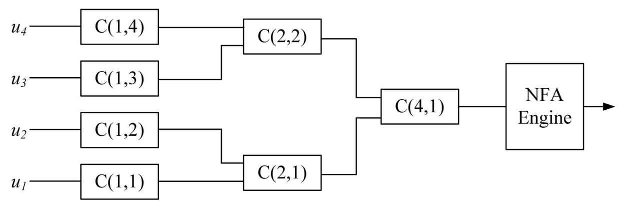


Fig. 6. The pipelined architecture for $\bar{M}^4_{RE}$.

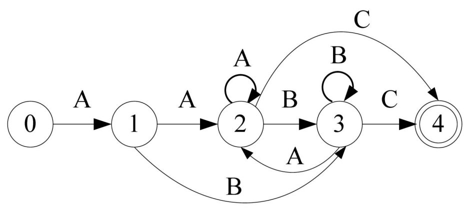
space for the $Follow(RE, x)$ table, $|\Sigma| \times N$ bits for the $Enter(\alpha)$ table, $2N$ OR gates, and $2N$ AND gates. Register B plays the role of state vector and has to be saved to memory if data arrives in small segments such as packets. If symbol encoder is adopted, then the memory space required by the symbol position vectors of the Shift-OR architecture and the $Enter(\alpha)$ table of our proposed architecture reduces to $|\Gamma| \times N$ bits, where Γ is the set of equivalence classes whose size is equal to the number of distinct symbols that appear in the pattern plus 1. The trade-off is more logic for the symbol encoder, which can be implemented with table lookup. If every symbol appears in any position of the pattern with equal probability, then the expected size of Γ is given by $|\Sigma|\{1 - [(|\Sigma| - 1)/|\Sigma|]^N\} + 1$.

Assume that $K(\geq 2)$ symbols are processed in each operation cycle. Assume further that no multiport memory is used. The memory space required by the Shift-OR architecture becomes $|\Sigma| \times N \times K$ bits if symbol encoders are not adopted or $|\Gamma| \times N \times K$ bits otherwise. As for our proposed architecture, the hierarchical symbol encoders are needed. Otherwise, the number of K-symbols would become prohibitively large when K is large. The $F(u)$ table requires $|\Gamma| \times (N + 1)$ bits and the $H(x, u)$ table needs $|\Gamma|\ x\ (N + 1)^2$ bits. Obviously, the space requirement highly depends on $|\Gamma|$. How to derive the expected value of $|\Gamma|$ is an interesting but challenging work. We provide an upper bound here. Let l_i, $1 \leq i \leq K - 1$, denote the number of i-symbols $u \in \Sigma^i$ such that $\delta_i(0, u) \cap S \neq \emptyset$ and $L_i = \sum_{j=1}^{i} l_j$. Let n_i, $1 \leq i \leq K - 1$, represent the number of i-symbols u such that $\delta_i\ (x, u) \cap Last(RE) \neq \emptyset$ for some state $x \in S$. Finally, let g be the number of K-symbols u such that $\delta_K(x, u) \cap S \neq \emptyset$ for some $x \in S$. The number of equivalence classes is upper bounded by $g + L_{K-1} + \sum_{i=1}^{K-1} n_i + \sum_{i=1}^{K-1} n_i L_{K-i} + 1$. For our running example, we have $g = 18$, $l_1 = 2$, $l_2 = 2$, $l_3 = 4$, $n_1 = 3$, $n_2 = 4$, and $n_3 = 2$. As a result, the number of equivalence classes is upper bounded by $18{+}8{+}9{+}3{\times}8{+}4{\times}4{+}2{\times}2{+}1{=}80$, which does not differ a lot from the actual number 58.

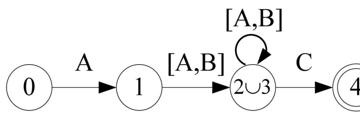
It is clear that the space complexity of our proposed architecture is higher than that of the Shift-OR architecture. This is the price paid for matching regular expressions rather than plain strings.

7 Some Example Regular Expressions

In this section, we study two extended regular expressions selected from Snort [1]. It is possible to reduce the number of states in a G-NFA if we allow an edge to be labeled with multiple symbols. Two states m and n can be merged into one, called state $m \cup n$, if



(a)



(b)

Fig. 7. (a) The G-NFA and (b) the reduced G-NFA of $\mathrm{A(A|B)^+C}$.

1. both are final states or both are nonfinal states,
2. $m \in Follow(RE, x)$ implies $n \in Follow(RE, x)$,
3. $x \in Follow(RE, m)$ implies $x \in Follow(RE, n)$, and
4. $n \in Follow(RE, m)$ implies $m \in Follow(RE, n)$.

After states m and n are merged, state $m \cup n$ satisfies the following conditions:

1. state $m \cup n$ is a final state if both states m and n are final states,
2. $m \cup n \in \delta(x, \alpha)$ if $m \in \delta\ (x, \alpha)$ or $n \in \delta(x, \alpha)$,
3. $x \in \delta\ (m \cup n, \alpha)$ if $x \in \delta(m, \alpha)$ and $x \in \delta\ (n, \alpha)$, and
4. $m \cup n \in \delta(m \cup n, \alpha)$ if $m \in \delta(m, \alpha)$ or $n \in \delta(n, \alpha)$.

Clearly, the process of merging states can be performed iteratively. For convenience, the resulting NFA when no more merging is possible is called the reduced G-NFA. As an example, Figs. 7a and 7b illustrate, respectively, the G-NFA and the reduced G-NFA of the regular expression $\mathrm{A(A|B)^+C}$. In Fig. 7b, [A, B] represents $\mathrm{A|B}$.

Example 1. Consider the extended regular expression $\mathrm{{}^\wedge rcpt\backslash\, s^{+}to\backslash\, x3a\backslash\, s^{*}[|, \backslash x3b]/\mathbf{mi}}$. The symbol $^\wedge$ means match the beginning of the line. The symbol $\backslash\mathrm{s}$ denotes white space. Symbols $\backslash\mathrm{x3a}$ and $\backslash\mathrm{x3b}$ represent 3a and 3b in hexadecimal, which are ":" and ";", respectively. The options $\mathbf{m}$ and $\mathbf{i}$ indicate match on all line breaks and case insensitive, respectively. Let us consider the case of $K = 1$. There are 11 states in the reduced G-NFA and the set of equivalence classes for input symbols are $\{[\mathrm{r, R}], [\mathrm{c, C}], [\mathrm{p, P}], [\mathrm{t, T}], [\mathrm{o, O}], \backslash\mathrm{s}, \backslash\mathrm{x3a}, [|, \backslash\mathrm{x3b}], \sim\}$. To implement option $\mathbf{m}$, we reset the G-NFA whenever a newline symbol is encountered. We implemented the architecture presented in Section 5 with Xilinx ML 310 platform. Note that there is at most one active state at any moment, and therefore, the bitwise OR logic can be removed. Also, there is only one final state, which means that the $Last$ bitmap is not needed. Hardware resources used in the implementation are two slices, three slice flip flops, four (input) LUTs, and one BRAM. The NFA constructed with the approach proposed in [15] uses 20 slices, four slice flip flops, and 36 LUTs. For $K = 4$, we have $\delta(x, u) = Follow(RE, x) \cap Enter(u)$ for any state x and 4-symbol u. Therefore, the architecture presented in Section 5 is applicable. For this example, there are 22 equivalence classes for all the 4-symbols. We used 14 slices, 16 slice flip flops, 25 LUTs, and four BRAMs in the implementation.

Example 2. Consider the extended regular expression $\hat{}\mathrm{PRIVMSG}\backslash\ s^{+}[\hat{}\backslash\ s]^{+}\backslash\ s^{+}\backslash\ \mathrm{x3a}\backslash\ s^{*}\backslash\mathrm{x01SENDLINK}\backslash\ \mathrm{x7c}[\hat{}\backslash\mathrm{x7c}]\{69\}/$**smi**. The symbol $[\hat{}\backslash s]$ represents any symbol, which is not white space. The bound special symbol {69} can be handled with the inductions described in Section 4. The total number of states in the reduced G-NFA is 92. The option s means that the dot metacharacter includes newline. However, it is redundant for this example because the dot metacharacter does not appear in the regular expression. Again, there is at most one active state at any moment and exactly one final state. For $K = 1$, there are 17 equivalence classes for input symbols. To reduce memory requirement, bitmap B is replaced with a register to store the current active state. Besides, states are classified into two groups. State x is in Group 1 if there exists only one equivalence class, denoted by $\mathrm{EC}(x)$, such that $\delta(x, u) \neq \emptyset$ only if u is in $\mathrm{EC}(x)$. State x is in Group 2 if it is not in Group 1. For this example, Group 1 contains 87 states. For convenience, we renumber the states so that state $x \in \mathrm{Group\ 1}$ iff $x \leq 86$. For state $x \in \mathrm{Group\ 1}$, we use 2 bytes to store $\mathrm{EC}(x)$ and $\delta(x, u)$ for any u in $\mathrm{EC}(x)$. For state $y \in \mathrm{Group\ 2}$, we store an array of 17 elements where the $i\mathrm{th}$ entry is $\delta(x, u)$ for any u in ECID i. With such modifications, the total memory requirement for the NFA engine is about 2 Kbits. The ECID decoder requires 2 Kbits of memory (we use 1 byte for ECID). The implementation uses 43 slices, 45 slice flip flops, 63 LUTs, and two BRAMs. The NFA constructed with the approach proposed in [15] uses 128 slices, 32 slice flip flops, and 227 LUTs. For $K = 4$, the equality $\delta(x, u) = Follow(RE, x) \cap Enter(u)$ does not always hold, and thus, we need to use the architecture presented in Section 6. Implementation for $K = 4$ is similar to that for $K = 1$. For this example, Group 1 contains 84 states. The memory requirement for the NFA engine is about 4.5 Kbits and that for the ECID decoder is about 7.5 Kbits. In our implementation, we use 45 slices, 47 slice flip flops, 75 LUTs, and five BRAMs.

The above two examples are studied only for proof of concept. The clock rates for our proposed architectures are slightly larger than that for the logic-based design proposed in [15]. We achieved more than 4 Gbps throughput for both examples with $K = 4$. With some manipulations, it is possible to reduce the required hardware resources. For example, both the $Follow(RE, x)$ and the $Enter(\alpha)$ tables can be compressed. As another example, one can implement the bound special symbol {69} shown in Example 2 with a counter. By doing so, the number of states is reduced to 24. Compared with logic-based designs, our proposed architectures require additional memory but less logic circuit. One major advantage of our proposed architectures is that they can process data that arrives in small segments, such as packets while logic-based designs cannot.

8 Conclusion

We have presented in this paper a bitmap-based hardware architecture for G-NFA. The architecture is generalized to an NFA that processes multiple symbols per operation cycle to improve system performance. Our proposed bitmap-based architecture is suitable for regular expressions of small and moderate lengths.

We prototyped the proposed NFA engine with Xilink ML310 platform and achieve more than 4 Gbps throughput for $K = 4$. In our experiment, we only selected a few regular expressions from Snort rules to prove our design concept. A hardware-accelerated intrusion detection system based on Snort is currently under development. In the system, we will implement a four-step NFA in hardware to achieve high system throughput and another one-step NFA in software for match verifications. After deeply examining the rules, we can hopefully develop efficient implementation techniques specifically for Snort and generate some guidelines for writing rules to facilitate efficient hardware acceleration.

Acknowledgments

This work was supported by the National Science Council (NSC) under Contract NSC96-2221-E-009-018-MY2. The author would like to thank the anonymous referees for their helpful suggestions.

References

[1] SNORT, http://www.snort.org, 2008.

[2] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliography Search," *Comm. ACM,* vol. 18, no. 6, pp. 333-340, 1975.

[3] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," *Proc. IEEE INFOCOM '04,* pp. 333-340, 2004.

[4] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *Proc. Int'l Soc. Computers and Their Applications (ISCA),* 2005.

[5] Y. Sugawara, M. Inaba, and K. Hiraki, "Over 10 Gbps String Matching Mechanism for Multi-Stream Packet Scanning Systems," *Proc. 14th Int'l Conf. Field Programmable Logic and Applications (FPL),* 2004.

[6] T.H. Lee and J.C. Liang, "A High-Performance Memory-Efficient Pattern Matching Algorithm and Its Implementation," *Proc. IEEE Technical Conf. (TENCON),* 2006.

[7] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," *Proc. 12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* 2004.

[8] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Content Filtering," *Proc. ACM/IEEE Symp. Architecture for Networking and Comm. Systems (ANCS),* 2005.

[9] S. Yusuf and W. Luk, "Bitwise Optimized CAM for Network Intrusion Detection Systems," *Proc. 15th Int'l Conf. Field Programmable Logic and Applications (FPL),* 2005.

[10] S. Kim, "Pattern Matching Acceleration for Network Intrusion Detection Systems," *Proc. Fifth Int'l Workshop Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS),* 2005.

[11] D. Kim, S. Kim, L. Choi, and H. Kim, "A High-Throughput System Architecture for Deep Packet Filtering in Network Intrusion Prevention," *Proc. 19th Int'l Conf. Architecture of Computing Systems (ARCS),* 2006.

[12] H.C. Roan, W.J. Hwang, and C.T. Lo, "Shift-Or Circuit for Efficient Network Intrusion Detection Pattern Matching," *Proc. 16th Int'l Conf. Field Programmable Logic and Applications (FPL),* 2006.

[13] R.A. Baeza-Yates and G.H. Gonnet, "A New Approach to Text Searching," *Proc. ACM 12th Int'l Conf. Research and Development in Information Retrieval (SIGIR),* 1989.
[14] R.W. Floyd and J.D. Ullman, "The Compilation of Regular Expression into Integrated Circuits," *J. ACM,* vol. 29, no. 3, pp. 603-622, July 1982.
[15] R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching Using FPGAs," *Proc. Ninth IEEE Symp. Field-Programmable Custom Computing Machines (FCCM),* 2001.
[16] C.R. Clark and D.E. Schimmel, "Efficient Reconfigurable Logic Circuit for Matching Complex Network Intrusion Detection Patterns," *Proc. 13th Int'l Conf. Field Programmable Logic and Applications (FPL),* 2003.
[17] J. Moscola et al., "Implementation of a Content-Scanning Module for an Internet Firewall," *Proc. IEEE Workshop FPGAs for Custom Computing Machines,* Apr. 2003.
[18] S. Kumar et al., "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," *Proc. ACM SIGCOMM,* 2006.
[19] V.M. Glushkov, "The Abstract Theory of Automata," *Russian Math. Surveys,* vol. 16, pp. 1-53, 1961.
[20] *POSIX 1003.2 Regular Expressions,* ISO/IEC 9945, 2003.
[21] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

Tsern-Huei Lee received the BS degree in electrical engineering from the National Taiwan University, Taipei, in 1981, the MS degree in electrical engineering from the University of California, Santa Barbara, in 1984, and the PhD degree in electrical engineering from the University of Southern California, Los Angeles, in 1987. Since 1987, he has been a member of the faculty of the National Chiao Tung University, Hsinchu, Taiwan, where he is a professor in the Department of Communication Engineering and a member of the Center for Telecommunications Research. He serves as a consultant of various research institutes and local companies. His current research interests are in network security, broadband switching systems, network traffic management, and wireless communications. He received an Outstanding Paper Award from the Institute of Chinese Engineers in 1991. He is a senior member of the IEEE.