Reduced Power Consumption via Fewer Memory Accesses for Deep Packet Inspection[*]

HANSOO KIM[1,2], YOUNGLOK KIM[2] AND JU WOOK JANG[2,+]
[1]*Digital Technology and Biometry Division*
National Forensic Service
Yangcheongu, Seoul 158-707, Korea
[2]*Electronic Engineering Department*
Sogang University
Mapogu, Seoul 121-742, Korea

We propose a new mapping scheme for AC-DFA tries to be used in FPGA implementation of deep packet inspection (DPI). Our scheme greatly reduces number of memory accesses which are responsible for most of the power consumption in DPI. We vary strides in the construction of AC-DFA tries in such a way that the number of memory accesses is minimized without increasing the memory space. Compared with the state-of-the-art DPI architecture [3], our scheme shows 34% reduction in power consumption and 14% reduction in memory space.

Keywords: intrusion detection, pattern matching, variable stride, deep packet inspection, Snort

1. INTRODUCTION

Deep packet inspection (DPI, Snort [2] for example) has become one of the most reliable and trustworthy systems to eliminate network threats such as viruses, malicious packets and DDoS attempts. The functions of DPI systems rely on multi-pattern string matching, which scans the input stream to find all occurrences of a predefined set of string-based patterns [3].

Along with the considerations of speed and accuracy, the memory space and power consumption required for the functioning of the DPI system is of significant concern. The majority of the memory space and power consumption used for DPI is for multi-pattern string matching [4, 5], hence a large amount of research on reducing the memory space and power consumption is focused on string matching.

Caching with partitioning the patterns [17] and two-stage decomposing [18] reduce the memory space, but they are less effective. A scalable architecture with pipelining [3] remarkably reduces the memory space, but it results in large hardware usage with correspondingly larger power consumption. Some hashing algorithm designs [4, 8, 14] have increased speed and added efficiency to string searching, but problems remain regarding the implementation cost and additional power consumption.

Many multi-pattern string matching solutions adopt the well-known Aho-Corasick (AC) algorithm, where the system is modeled as a deterministic finite automaton (DFA) [6]. Motivated by the these observations, we propose a new mapping scheme for pipeline

Received November 19, 2012; revised February 17, 2013; accepted March 4, 2013.
Communicated by Cho-Li Wang.
[*] This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (NRF-2013R1A1A2011856).
[+] Corresponding author: jjang@sogang.ac.kr.

implementation which traverses an AC-DFA trie with varying strides depending on the degrees of the nodes in such a way to minimize memory accesses. It is known that memory accesses are responsible for most of the power consumption [5] in DPI systems. Our main contribution can be summarized as follows. First we introduce formulas to calculate the number of memory accesses as we change strides in traversing regular tries. Second we use this formula to develop a heuristic algorithm to minimize the number of accesses for all tries including irregular ones. To further reduce the number of memory accesses, we also employ a binary search scheme to access the memory where the patterns are located. As a result, our new mapping technique reduces power consumption and memory space by 34% and 14% respectively when compared against the state-of-the-art implementation [3].

The rest of this paper is organized as follows. Section 2 reviews multi-pattern string matching and relevant recent studies, and Section 3 shows the proposed mapping scheme which reduces power consumption. Section 4 shows the experimental results and Section 5 concludes this paper.

2. RELATED WORKS

2.1 Compression of an AC-DFA Trie with the Stride *s*

AC-DFA converts a pattern set which contains *n* characters into a deterministic finite automaton with *O*(*n*) states. Once the DFA which can be stored as a state transition table is built, it reads the input stream one character per clock cycle. Each input character is processed only once and results in exactly one state transition [3].

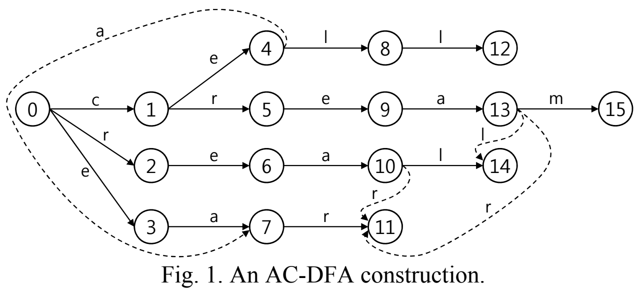


Fig. 1. An AC-DFA construction.

Fig. 1 illustrates a construction of AC-DFA for four patterns of “ell”, “cream”, “real” and “ear” AC-DFA starts with constructing a trie (AC-trie) where the root is the default non-matching state. Each pattern to be matched adds a state to the trie, one state per character, starting at the root and going to the end of the pattern [3]. This is called a *goto* transition [6]. For example, the pattern “cell” adds states 1, 4, 8, and 12 as shown in Fig. 1. In case there is a mismatch in the goto transition, an additional transition is added which is called a *failure* transition [7]. All states except states 4, 10 and 13 use the root as default failure transition state. Some patterns share strings with other patterns. In this case, failure transition can be made to other non-root state. This is illustrated in the following example. Patterns “real” and “ear” share a string “ea”. At state 10 in Fig. 1, if input character “r” is encountered transition can be made to state 11 instead of the root state. The goto transitions are called the *forward* transitions, and the failure transitions are called the *cross* transitions. The forward transitions are denoted as solid lines with arrows while dotted lines with arrows denote the cross transitions (The cross transitions to the root state are not shown).

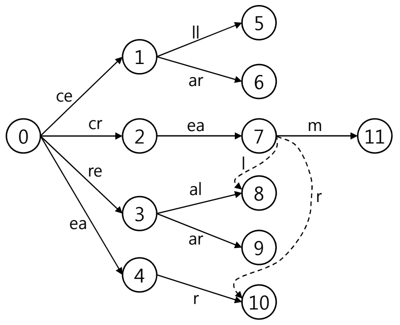


Fig. 2. The compressed version of the AC-DFA construction in Fig. 1.

In addition to this, the compressed AC-DFA trie [10] has one transition on multiple input characters, by combining *k* consecutive states of the original Aho-Corasick DFA trie. The original AC-DFA trie can be compressed by dividing each pattern to be matched into a certain number of characters, which is called a *stride*. As an example, Fig. 2 shows the compressed AC-DFA trie with a stride of 2, for the original AC-DFA trie shown in Fig. 1. Two input characters are fetched at a time and compared against the patterns on edges from the current node. If there is any match for these two input characters, transition is made accordingly and the next two input characters are fetched. If there is no match for these two input characters on any edges from the current node, then next two input characters should be fetched with one-character offset to ensure that no patterns are missed. Assuming that the input stream is “crear”, the first two input characters, “cr”, are read and compared against “ce”, “cr”, “re”, and “ea” in this order. Transition is made to node 2. And then next two input characters “ea” are fetched and compared against “ea”. Since it is a match, transition is made to node 7. Finally “r” is fetched and compared against “m” and “r”. Since this is a match for cross transition, node 10 is reached. If input stream is “dreal”, the first two input characters, “dr”, are fetched and compared against “ce”, “cr”, “re”, and “ea” in this order. No match. Two input characters with one-character offset to the previous input characters (*i.e.* “re”) are fetched compared against “ce”, “cr”, “re”, and “ea” in this order. Note that no patterns will be missed if we use this one-character offset on no match for multiple input characters.

2.2 Removing Cross Transitions with Pipelining

Fig. 3 shows a mapping of the AC-DFA trie in Fig. 1 onto a pipeline with each level to one stage. Using the aspect that the cross transitions are added into the AC-DFA trie for a failed match in order to reuse the history information without restarting from the root, Pao *et al.* [8] and Jiang *et al.* [3] deliver the input characters to all the pipeline stages in parallel, including the new input characters to the root with a one character offset at each clock cycle so as to remove the cross transitions to the stages that are on the pipelines. With this approach, all the cross transitions can be removed.

Suppose that the input stream is "crear" and at clock 1, the first input character "c" is fetched to the root node and comparison is made against "c", "r", and "e" in order. Match. Node 1 in stage 0 is reached and second input character, "r", is fetched at clock 2. At the same time, the character "r" is also fed to the root node in stage 0 to process an input stream staring with "r". In this way we process virtually multiple input streams, "crear", "rear", "ear", "ar", and "r" simultaneously. The cross transition from 13 to 11 can be removed since node 11 is eventually reached while processing "ear". Thus, the memory space for storing these cross transitions is also removed.

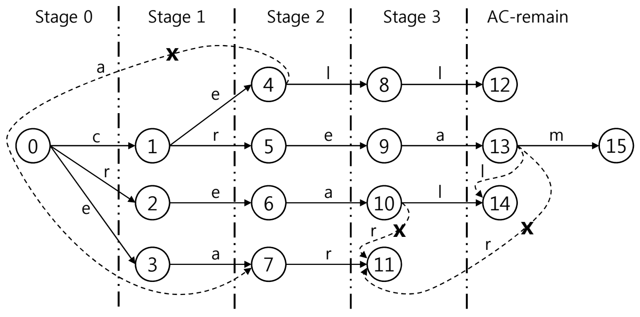


Fig. 3. The pipelined AC-DFA.

2.3 Memory Accesses Responsible for Most of Power Consumption

For clarity, we define the following terms.

- *s*: the stride of an AC-DFA trie
- *d*: the degree of a node
- *C*: the alphabet size
- *H*: the height of an AC-DFA trie
- *Msp*(*d*, *C*, *H*): the memory space to store the patterns as a function of *d*, *C* and *H* of the AC-DFA trie
- *Mna*(*d*, *C*, *H*): the number of memory accesses to fetch the patterns as a function of *d*, *C* and *H* of the AC-DFA trie

The *stride* (*s*) of a node is defined to be the number of input characters to be matched at a time in traversing an AC-DFA trie (see Section 2.1 for illustration). Compression of the AC-DFA trie in Fig. 1 by increasing the stride to 2 is shown in Fig. 2. The *depth* of a node *M* in a trie is the length of the path from the root of the trie to *M*. The *height* (*H*) of a trie is the depth of the deepest node in the trie. All nodes of depth p are at *level p* in the trie. The root is the only node at level 0, and its depth is 0. The *degree* (*d*) of a node is defined as the number of outgoing edges [11]. The *alphabet size* (*C*) is the number of different patterns (characters) that can be mapped onto an edge in an AC-DFA trie (in bytes). For ASCII character set [1], the alphabet size would be 256 (one for each character in the alphabet) [4].

It is known that memory accesses are responsible for most of the power consumption in DPI systems, as much as 85% [5]. This implies that reducing the number of memory accesses will greatly reduce the overall power consumption. Removing cross transitions via pipelining [3, 8] eliminates memory space and memory accesses needed for cross transitions. To further reduce power consumption, we propose to change stride (degree of compression) in such a way to minimize number of memory accesses without increase in memory space. We show effectiveness of our scheme in mathematical analysis with regular AC-DFA tries. We introduce formulas to calculate the number of memory accesses as we change strides in traversing the tries. Second we use this formula to develop a heuristic algorithm to minimize the number of accesses for all tries including irregular ones. To further reduce the number of memory accesses, we also employ a binary search scheme to access the memory where the patterns are located. As a result, our new mapping technique reduces power consumption and memory space by 34% and 14% respectively when compared against the state-of-the-art implementation [3].

3. PROPOSED ARCHITECTURE

The memory space and the number of memory accesses of an AC-DFA trie may depend on d, C and H. It is not easy to derive formulas relating these parameters to the memory space and the number of memory accesses for all possible AC-DFA tries. Therefore, we first consider regular AC-DFA tries in which we assume that the degrees of all the nodes are the same except the leaf nodes in the AC-DFA trie. We derive formulas for these regular tries, and the formulas can be used to identify the impact of parameters on the memory space and the number of memory accesses. Then we use this result to develop a mapping scheme to adjust strides in such a way to minimize the number of memory accesses for all AC-DFA tries including irregular ones.

Fig. 4 shows the example of the AC-DFA trie in which the degrees of all the nodes except the leaf nodes are the same, with $d = 2$. The parameters of the AC-DFA trie in Fig. 4 are as follows.

- the stride $s = 1$
- the degree of the nodes $d = 2$
- the alphabet size $C = 256$
- the height $H = 4$

3.1 Calculating the Memory Space Requirement for Regular Tries

We can obtain the memory space required to store all the patterns, by counting the number of all the characters in the edges of the AC-DFA trie. In Fig. 4, the number of characters in all the edges is 30 (characters).

The number of edges in level i for regular tries is calculated by d^i. With the alphabet size C and the height H, the memory space can be written as a function of d, C and H.

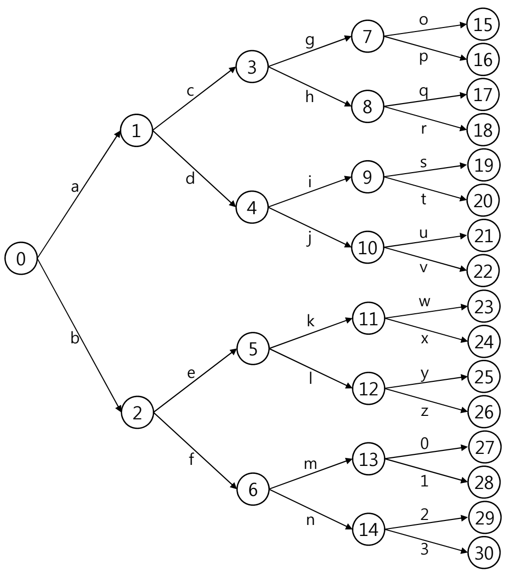


Fig. 4. A regular AC-DFA trie with $d = 2$.

From this, we can derive Eq. (1) in bytes.

$$Msp(d, C, H) = \left\lceil \frac{1}{8}\log_2 C \right\rceil \times \sum_{i=1}^{H} d^i = \left\lceil \frac{1}{8}\log_2 C \right\rceil \times \frac{d(d^H - 1)}{d - 1}(bytes) \tag{1}$$

In Fig. 4, we have $d = 2$ and $H = 4$. Each edge has one alphabetic character which is represented by an 8 bit ASCII code [1], which implies $C = 256$. With this, we can obtain the memory space for Fig. 4, as shown in Eq. (2).

$$Msp(2, 256, 4) = \left\lceil \frac{1}{8}\log_2 256 \right\rceil \times \sum_{i=1}^{4} 2^i = 2 + 4 + 8 + 16 = 30(bytes) \tag{2}$$

Meanwhile, Fig. 5 shows the example of the compressed AC-DFA trie from the original AC-DFA trie in Fig. 4. The following values are obtained for Fig. 5.

- the stride $s = 2$
- the degree of the nodes $d = 4$
- the alphabet size $C = 256^2$
  (two ASCII characters in one edge)
- the height $H = 2$

Using Eq. (1), we can get the memory space in Fig. 5 as shown in Eq. (3).

$$Msp(4, 256^2, 2) = \left\lceil \frac{1}{8}\log_2 256^2 \right\rceil \times \sum_{i=1}^{2} 4^i = 2 \times (4 + 16) = 40(bytes) \tag{3}$$

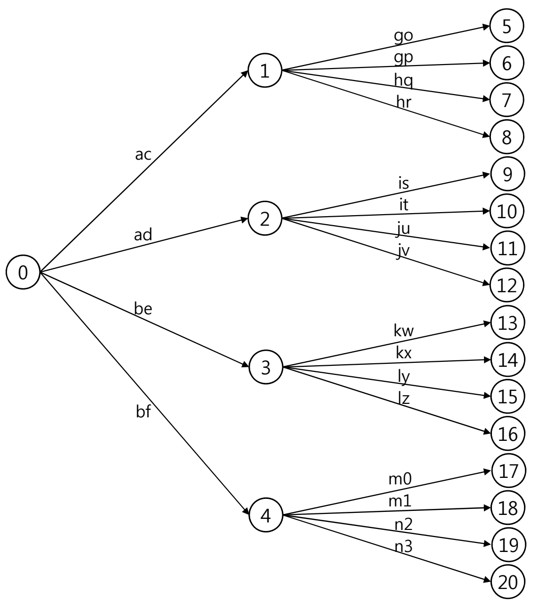


Fig. 5. A compressed AC-DFA trie from Fig. 4.

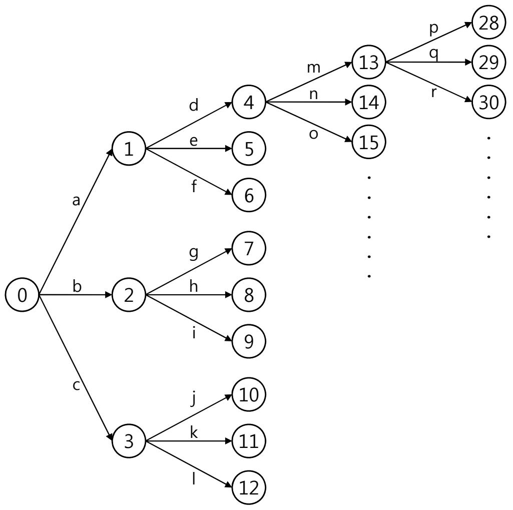


Fig. 6. A regular AC-DFA trie with $d = 3$.

Fig. 6 shows another example of regular AC-DFA tries, where the degree of the nodes $d = 3$. In Fig. 6, $d = 3$, $H = 4$ and $C = 256$. We can apply these parameters to Eq. (1) and obtain the memory space of Fig. 6, as shown in Eq. (4).

$$Msp(3,256,4) = \left\lceil \frac{1}{8}\log_2 256 \right\rceil \times \sum_{i=1}^{4} 3^i = 3 + 9 + 27 + 81 = 120(bytes) \tag{4}$$

3.2 Calculating the Average Number of Memory Accesses for Regular Tries

We can obtain the number of memory accesses to fetch the required patterns, by summing the probability of accessing the edge of the AC-DFA trie to fetch the patterns assigned to that edge. Considering Fig. 4, we can obtain the number of memory accesses as follows.

At level 0 in Fig. 4, the probability of accessing the first edge with 'a' is 1, for all the input characters should be compared to the first character on the first edge. The probability of accessing the second edge with 'b' is the probability of mismatch at the first edge with 'a', so it is $(C - 1)/C = 255/256$. Assuming all characters are equally probable, we estimate the average number of memory accesses at level 0 as 1 + 255/256 = 1.996.

At level 1, the probability of accessing the first edge with 'c' equals the probability of matching 'a' at level 0, so it is $1/C = 1/256$. The probability of accessing the second edge 'd' equals the probability of matching 'a' at level 0 times the probability of mismatch at the first edge 'c', so it is $1/C * (C - 1)/C = 1/256 * 255/256$. Likewise, the probability of accessing the third edge 'e' is 1/256, and the probability of accessing the fourth edge 'f' is 1/256 * 255/256. The number of memory accesses at level 1 is estimated as 2/256 * (1 + 255/256) = 2/256 * 1.996.

At level 2, the probability of accessing the first edge 'g' equals the probability of matching 'a' at level 0 times the probability of matching 'c' at level 1, so it is $1/C * 1/C = 1/256^2$. The probability of accessing the second edge 'h' equals the probability of matching 'a' at level 0 times the probability of matching 'c' at level 1 times the probability of mismatching at the first edge 'g', so it is $1/256^2 * 255/256$. By using similar calculation for the rest of edges, we can obtain that the number of memory accesses at level 2 is $4/256^2 * (1 + 255/256) = 4/256^2 * 1.996$.

At level 3, the probability of accessing the first edge 'o' equals the probability of matching 'a' at level 0 times the probability of matching 'c' at level 1 times the probability of matching 'g' at level 2, which is $1/256^3$. The probability of accessing the second edge 'p' equals the probability of matching 'a' at level 0 times the probability of matching 'c' at level 1 times the probability of matching 'h' at level 2 times the probability of mismatching at the first edge 'g', so it is $1/256^3 * 255/256$. Likewise, we can obtain that the number of memory accesses at level 3 is $8/256^3 * (1 + 255/256) = 8/256^3 * 1.996$. Finally we can obtain the total number of memory accesses as $1.996 + 2/256 * 1.996 + 4/256^2 * 1.996 + 8/256^3 * 1.996$, which is approximately 2.021.

Generalizing this with d, C and H for regular tries, the number of memory accesses can be represented with a function of d, C and H as shown in Eq. (5).

$$Mna(d, C, H) = \left(\frac{d^0}{C^0} + \frac{d^1}{C^1} + \frac{d^2}{C^2} + \ldots + \frac{d^{H-1}}{C^{H-1}}\right) \times \left(\frac{C}{C} + \frac{C-1}{C} + \frac{C-2}{C} + \ldots + \frac{C-(d-1)}{C}\right) = \sum_{i=1}^{H} \frac{d^{i-1}}{C^{i-1}} \times \sum_{j=0}^{d-1} \frac{C-j}{C} \tag{5}$$

Assuming $C = 256$ and $d << C$, we can approximate Eq. (5) as follows.

$$\sum_{j=0}^{d-1} \frac{C-j}{C} = 1 + 0.996 + 0.992 + \ldots \cong d \text{ (for small } d) \tag{6}$$

$$\sum_{i=1}^{H}\frac{d^{i-1}}{C^{i-1}}=\frac{d^0}{C^0}+\frac{d^1}{C^1}+\frac{d^2}{C^2}+\frac{d^3}{C^3}+\ldots=\frac{1-\left(\frac{d}{C}\right)^H}{1-\frac{d}{C}}\cong\frac{1}{1-\frac{d}{C}}\quad\text{(for large }H\text{)}\tag{7}$$

With Eqs. (6) and (7), we can obtain Eq. (8).

$$Mna(d,C,H)=\sum_{i=1}^{H}\frac{d^{i-1}}{C^{i-1}}\times\sum_{j=0}^{d-1}\frac{C-j}{C}\cong\frac{1}{1-\frac{d}{C}}\cdot d=\frac{C\cdot d}{C-d}\tag{8}$$

By using Eq. (8), the number of memory accesses for Fig. 4 can be obtained as in Eq. (9).

$$Mna(2,256,4)\cong\frac{256\cdot 2}{256-2}\cong 2.016\tag{9}$$

Note that this is close to the value obtained in the above (2.021).

Also, we can obtain the number of memory accesses of the compressed AC-DFA trie shown in Fig. 5. At level 0, the probability of accessing the first edge 'ac' is 1 and the second edge 'ad' is the probability of mismatching at the first edge 'ac', $(C-1)/C = (256^2-1)/256^2$. The probability of accessing the third edge 'be' is the probability of mismatching at the first and the second edges, $(C-2)/C = (256^2-2)/256^2$. The probability of accessing the fourth edge 'bf' is the probability of mismatching at the first, the second and the third edges, $(C-3)/C = (256^2-3)/256^2$. Continuing in this manner, we obtain the total number of memory accesses as 3.998.

Using Eq. (8), the number of memory accesses in Fig. 5 can be obtained as in Eq. (10) with similar value.

$$Mna(4,256^2,2)\cong\frac{256^2\cdot 4}{256^2-4}\cong 4.000\tag{10}$$

Considering the examples in Figs. 4 and 5, we can state that d, C and H of the compressed AC-DFA trie with the stride s are equal to d_1^s, C_1^s and H_1/s respectively, where d_1, C_1 and H_1 are the parameters of the original AC-DFA trie.

In addition, the number of memory accesses in Fig. 6 can be obtained by summing the expected number of accesses to all edges, which is approximately 2.9883 + 0.0350 + 0.0004 + 0.0001 = 3.024. The number of memory accesses in Fig. 6 using Eq. (8) is shown in Eq. (11).

$$Mna(3,256,4)\cong\frac{256\cdot 3}{256-3}\cong 3.036\tag{11}$$

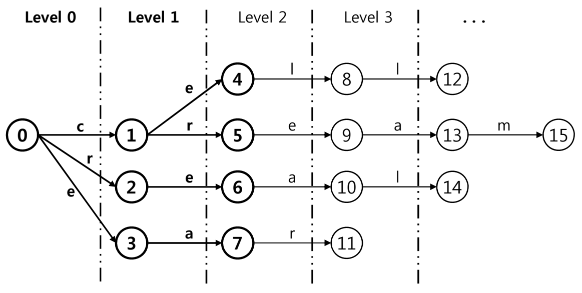
We see that the approximation (3.036) using Eq. (8) is very close to actual value (3.024).

3.3 Determining the Strides of Irregular Tries

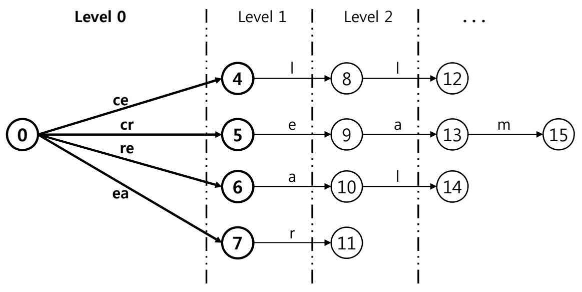
As seen so far, the memory space and the number of memory accesses depend on *d*, *C* and *H* of the AC-DFA trie. In contrast to the fixed stride architecture [3], we change the stride to minimize the number of memory accesses.

Since the actual AC-DFA trie may not be regular, we devise the following iterative scheme. We map each level of the AC-DFA trie to a stage of a pipeline, to eliminate the cross transitions as described in Section 2.2. Starting from the root of the original AC-DFA trie, we keep compressing the trie as far as the number of memory accesses is reduced by doing so. For this we do the following at current level starting from the root: (1) Check if incrementing stride reduce the number of memory accesses. If so, increase the stride at the current level by one (*i.e.* the next level will be combined into the current level to be compressed) and go to (1). Otherwise, start a new stride with setting the next level to current level and go to (2). (2) If the current level has no leaf nodes then stop. Otherwise go to (1). The resulting compressed trie may have different strides for different levels.

Step 1: Compare the number of memory accesses for the configuration in Fig. 7 (a) and the configuration in Fig. 7 (b). In Fig. 7 (a), the number of memory accesses of level 0 is 2.988 and the number of memory accesses of level 1 is 0.016, so the number of memory accesses for Fig. 7 (a) is 3.004. The number of memory accesses of level 0 for Fig. 7 (b) is 3.999. We choose Fig. 7 (a) with stride = 1 for the first level since the number of memory accesses is smaller.

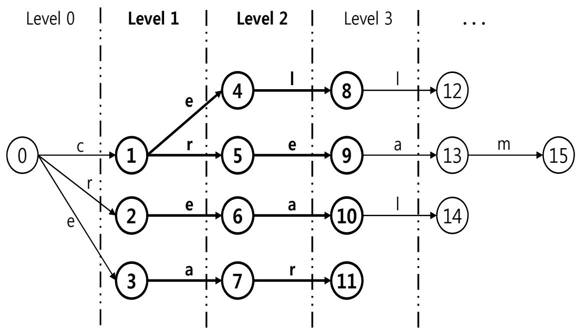


(a) Level 0 (root) and level 1 are not combined (stride = 1 at level 0).

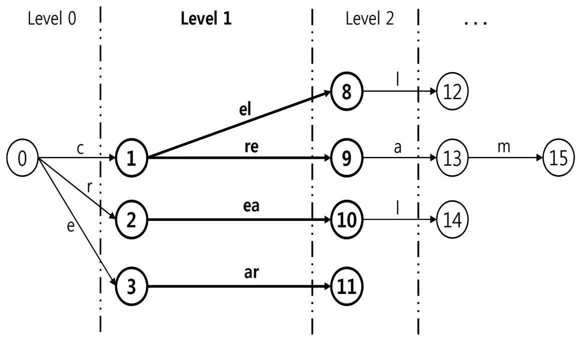


(b) Level 0 (root) and level 1 are combined together (stride = 2 at level 0).
Fig. 7. Checking if combining level 0 and level 1 reduce the number of memory accesses.

Step 2: Compare the number of memory accesses for the configuration in Fig. 8 (a) and the configuration in Fig. 8 (b). In Fig. 8 (a), the number of memory accesses of level 1 is 0.01562 and the number of memory accesses of level 2 is less than 0.0001, so the number of memory accesses for Fig. 8 (a) is 0.01562. The number of memory accesses of level 1 for Fig. 8 (b) is 0.01558. We choose Fig. 8 (b) with stride = 2 for the second level since the number of memory accesses is smaller.



(a) Level 1 and level 2 are not combined (stride = 1 at level 1).



(b) Level 1 and level 2 are combined together (stride = 2 at level 1).

Fig. 8. Checking if combining level 1 and level 2 reduce the number of memory accesses.

Step 3: Continue in this manner until the last level of the original AC-DFA trie.

Table 1 shows the pseudo-code of the proposed algorithm, calculating the number of memory accesses of each condition and determining the strides iteratively. For the calculation, we define a part of the compressed AC-DFA trie and its properties as follows.

- $P(a, b)$: a part of the compressed AC-DFA trie (nodes of one level and their edges, compressed from level a to level b of the original AC-DFA trie)
- H: the height of the original AC-DFA trie
- G: the height of the compressed AC-DFA trie
- $s(j)$: the stride of jth level of the compressed AC-DFA trie
- $Mna[P(a, b)]$: the number of memory accesses of $P(a, b)$

Table 1. Pseudo-code.

```
Input: the original AC-DFA trie
Otput: $s(j)$, $G$

str = 1, $i = 0$, $j = 0$
while ($i \leq H - 1$)
{
    if    $Mna[P(i, i + \text{str})] + Mna[P(i + \text{str}, i + \text{str} + 1)] \leq Mna[P(i, i + \text{str} + 1)]$
    then  $s(j) = \text{str}$, $i$ += str, $j$++, str = 1
    else  str++
}
$G = j$
```

4. EXPERIMENTAL RESULTS

Experiments on the ruleset from Snort [2], the well-known DPI system, are performed. We construct the original AC-DFA trie from the *content* field text data of the ruleset and apply our algorithm. The number of distinct rule sets is 610 and the constructed AC-DFA trie has a maximum of 150 levels (the largest ruleset has 150 characters) with a total of 6435 characters. We only consider the levels of the AC-DFA trie which are the destinations of the cross transitions [3, 8], so the first 48 levels of the constructed AC-DFA trie are reconstructed using the proposed method shown in Section 3.3. All AC-DFA tries are implemented, the memory space and the number of memory accesses are enumerated and the input data is randomly generated with Microsoft Visual C++ 2008 and SystemC 2.2 [12].

Table 2. Performance evaluation on randomly generated data.

Architectures	Stride	Memory Space (bytes)	Number of Memory Accesses	Total Power Consumption (J)
Jiang *et al.* [3]	4	4236	104	827719.7
Jiang *et al.* [3]	8	4760	106	833617.9
Proposed	variable	4089	11	556400.6

Table 3. Performance evaluation on an English text.

Architectures	Stride	Memory Space (bytes)	Number of Memory Accesses	Total Power Consumption (J)
Jiang *et al.* [3]	4	4236	1063	8510154.66
Jiang *et al.* [3]	8	4760	1094	8602322.40
Proposed	variable	4089	94	5652954.72

We compare our architecture against that proposed by Jiang *et al.* [3] which uses fixed strides on the compressed AC-DFA. In our experiment with the DDR3 DRAM [13] in the commonly used DPI system, two input data are used and the power consumption is estimated using the modified PowerSC [12, 15]. 32,768 characters are randomly generated and the English text of 341,362 characters (a text version of English Fairy Tales from [16]) is used as the input data. The experimental results are shown in Table 2 and Table 3. On randomly generated input data, our architecture reduces 14% of the memory space, 89 % in the number of memory accesses and 34% of the power consumption compared to the state-of-the-art architecture [3].

5. CONCLUSION

The proposed architecture reduce the memory power consumption by 34% and the required memory space by 14%, on the Snort pattern set, compared against the state-of-the-art architecture [3]. Our algorithm also shows that the varying strides depending on the degree of the nodes with binary search scheme for patterns is effective in reducing the memory space and the power consumption, for multi-pattern string matching such as Snort, the well-known DPI system. We will extend our algorithm to other multi-pattern string matching applications and power-aware systems in the future.

REFERENCES

1. ISO/IEC 8859-1, "Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1," *International Standards Organization*, April 1998.
2. "Snort: network intrusion detection system," Sourcefire Inc., http://www.snort.org.
3. W. Jiang, Y. E. Yang, and V. K. Prasanna, "Scalable multi-pipeline architecture for high performance multi-pattern string matching," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2010, pp. 1-12.
4. N. Hua, H. Song, and T. V. Lakshman, "Variable-stride multi pattern matching for scalable deep packet inspection," in *Proceedings of the 28th IEEE Conference on Computer Communications*, 2009, pp. 415-423.
5. W. Jiang and V. K. Prasanna, "Reducing dynamic power dissipation in pipelined forwarding engines," in *Proceedings of the 27th IEEE International Conference on Computer Design*, 2009, pp. 144-149.
6. A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, Vol. 18, 1975, pp. 333-340.
7. P.-C. Lin, Y.-D. Lin, T.-H. Lee, and Y.-C. Lai, "Using string matching for deep packet inspection," *Computer*, Vol. 41, 2008, pp. 23-28.
8. D. Pao, W. Lin, and B. Liu, "Pipelined architecture for multistring matching," *Computer Architecture Letters*, Vol. 7, 2008, pp. 33-36.
9. K. S. Kim and S. Sahni, "Efficient construction of pipelined multibit-trie router-tables," *IEEE Transactions on Computers*, Vol. 56, 2007, pp. 32-43.
10. M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network IDS/IPS," in *Proceedings of IEEE International Conference on Network Protocols*, 2006, pp. 187-196.
11. C. A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, 3rd ed., Prentice Hall, Upper Saddle River, NJ, 1997.
12. "Open systemC initiative," *Accellera Systems Initiative*, http://www.accellera.org.

13. "Samsung consumer DRAM," Samsung Electronics Co., Ltd., http://www.samsung.com/global/business/semiconductor/product/consumer-dram/catalogue.
14. Y. Choi, E. Hong, T. Kim, S. Baek, I. Choi, and H. Oh, "A traffic pattern matching hardware for a contents security system," *Magazine of the IEEK*, Vol. 46, 2009, pp. 88-95.
15. F. Klein, G. Araujo, and R. Azevedo, "PowerSC: a SystemC framework for power estimation," *6th North American SystemC User's Group Meeting*, 2007.
16. "English fairy tales," Joseph Jacobs Page, PSU's Electronic Classics Site, http://www.hn.psu.edu/faculty/jmanis/jimspdf.htm.
17. T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *Proceedings of the 27th IEEE Conference on Computer Communications*, 2008, pp. 166-170.
18. H. Chen, D. H. Summerville, and Y. Chen, "Two-stage decomposition of SNORT rules towards efficient hardware implementation," in *Proceedings of the 7th International Workshop on Design of Reliable Communication Networks*, 2009, pp. 166-170.

Hansoo Kim received the B.E. and M.E. degrees in Electronic Engineering from Sogang University, Korea, in 2002 and 2004, respectively. After that, he worked as a Junior Engineer at the Digital Media Research Laboratory in LG Electronics until 2005, and as a Senior Engineer in Nextreaming Corp. until 2006. He is currently a researcher and forensic expert in National Forensic Service, Korea, and also a Ph.D. candidate in Electronic Engineering from Sogang University. His research interest and work experience include application-layer network protocols, home networking, IPv6, multimedia streaming, digital forensics, document analysis, Internet security and cybercrime.

Younglok Kim received B.S. degree in Electronic Engineering from Sogang University, Seoul, Korea in 1991 and the M.S. and Ph.D. degrees in Electrical Engineering from Polytechnic Institute of NYU, Brooklyn, NY in 1993 and 1998 respectively. From 1999 to 2003, he was a Senior System Engineer at InterDigital communication Corp. in Melville, NY working on 3GPP wireless systems. Since 2003, he joined the Department of Electronic Engineering in Sogang University, where he is now a Professor. His research interests include DSP algorithms, signal processing for communication and radar.

Ju Wook Jang received the B.S. degree in Electronic Engineering from Seoul National University, Seoul, Korea, the M.S. degree in Electrical Engineering from the Korea Advanced Institute of Science and Technology (KAIST), and the Ph.D. degree in Electrical Engineering from the University of Southern California (USC), Los Angeles, USA. From 1985 to 1988 and 1993 to 1994, he was with Samsung Electronics, Suwon, Korea, where he was involved in the development of a 1.5-Mb/s video codec and a parallel computer. Since 1995, he has been with Sogang University, Seoul, Korea, where he is currently a Professor. His current research interests include WiMAX protocols, mobile networks and next generation networks. He received LG Yonam overseas research grant in 2001. He has also built systems for videoconferencing, streaming, home networks and ad hoc networks using protocols like RTP, SIP, multicast, and IPv6.