Compact State Machines for High Performance Pattern Matching

Piti Piyachon and Yan Luo
Dept. of Electrical and Computer Engineering, University of Massachusetts Lowell
Lowell, MA, USA
piti_piyachon@student.uml.edu, yan_luo@uml.edu

ABSTRACT

Pattern matching is essential to a wide range of applications such as network intrusion detection, virus scanning, etc. Pattern matching algorithms normally rely on state machines to detect predefined patterns. Recently, parallel pattern matching engines, based on ASICs, FPGAs or network processors, perform matching with multiple state machines. The state migration in the matching procedure incurs intensive memory accesses. Thus, it is critical to minimize the storage of state machines such that they can be fit in on-chip or other fast memory modules to achieve high-speed pattern matching. This paper proposes novel optimization techniques, namely state re-labeling and memory partition, to reduce state machine storage. The paper also presents architectural designs based on the optimization strategy. We evaluate our design using realistic pattern sets, and the results show state machine memory reduction up to 80.1%.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: microprocessor/microcomputer applications; C.4 [**Performance of Systems**]: design studies; C.1.4 [**Parallel Architectures**]: Distributed architectures

General Terms

Algorithms, Design, Performance, Security

Keywords

Pattern Matching, Parallel Processing

1. INTRODUCTION

Pattern matching searches for predefined patterns in data streams. A pattern, interchangeably called keyword, can be either a part of a rule of a network intrusion detection, or a signature string of a virus. The length of a pattern and its location in the data stream usually vary [7, 9]. A pattern set (or database) can contain thousands of patterns and keep expanding for enforcing new security policies or capturing new viruses. For example, the rule set of the well-known intrusion detection system, Snort[9], contains 2733 patterns as of Dec 2005, and new patterns are added constantly. Variable pattern length and location, and increasingly large pattern sets make pattern matching a challenging task.

Pattern matching algorithms, such as Aho-Corasick[1] and D^2FA [6], rely on state machines whose size is proportional to the size of pattern databases. Pattern matching takes in a data stream and follows the state migration until the input stream is exhausted. There exist lots of studies and several products on performing pattern searching with ASICs [3, 10], FPGAs [2, 4]. Recently, network processors (NPs) have also emerged as programmable processing units for pattern matching [5, 8].

State machines are normally stored in linear memory space, Pattern matching procedure incurs intensive memory accesses while traversing state machines. Fast memory access time can benefit the performance of pattern matching significantly. Thus, it is necessary to minimize the storage requirement of state machines such that they can be put into on-chip or other fast memory modules to achieve high-speed pattern matching.

This paper proposes novel schemes to reduce the memory required to store state machines in a linear memory space. One scheme is to *re-label* the states in a state machine such that the matching states are clustered at the beginning of a memory space, giving opportunities for compact storage. The other scheme is to *partition* state machines and use separate memory modules to store keywords and next-state pointers. We then present architectural design using the proposed schemes. We evaluate the performance of the optimization strategies using realistic pattern sets from Snort [9]. The experiment results show significant reduction (up to 80.1%) on the memory consumption of state machines.

The paper is organized as follows. Section 2 introduces the backgound of a parallel pattern matching architecture and Section 3 motivates our research. Section 4 describes the proposed optimization schemes. Section 5 presents the architectural design using the schemes. Section 6 evaluates the performance of our design and compares it with existing work. Finally, Section 7 concludes the paper.

2. BACKGROUND

The storage requirement of state machines significantly affects the performance of pattern matching. Traversing the state machine introduces intensive memory accesses, thus

DAC 2007, June 4–8, 2007, San Diego, California, USA.

it is desirable that state machines are small enough to fit in high-speed memory to reduce the memory access time. There exist research works addressing the state machine storage issue [10, 8]. Particularly, Tan-Sherwood proposed to leverage eight bit-level state machines to match eight individual bits of a byte in parallel, instead of matching byte by byte. Such bit-level state machines significantly reduce the number of next states to two (2^1), whereas a state in the classical state machine has 256 (2^8) possible next states.

Fig. 1 depicts a generic architecture of parallel pattern matching. The pattern matching machine consists of N parallel pattern matching engines. Each engine is instructed to search for a subset of the patterns, thus called Subset Engine (SE). A SE consists of multiple processing elements (PEs) can work in parallel to detect patterns [10, 8]. The results from PEs are AND-ed to generate the final result of a SE. Such a pattern matching machine, along with its subset engines, is technology-independent, i.e., it can be based on different technologies such as ASIC, FPGA or NP. SEs store state machines in either their local memory or a shared global memory module.

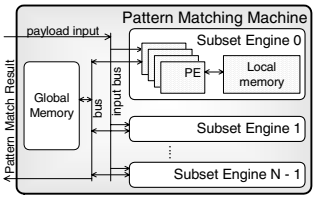

Figure 1: A generic parallel pattern matching architecture.

Both bit-level and byte-level state machines, represented with state diagrams, are implemented in a linear memory space in pattern matching system. Making easier to read, we hide irrelevant edges of state diagram, and draw the diagram as a trie structure (Fig.2). Fig.2 depicts a bit-level state machine for an example subset with 16 patterns, along with its corresponding linear memory layout. The annotations associated with double-circled nodes in Fig. 2(a) indicate detection of some patterns. In the memory layout (Fig. 2(b)), each state, represented in a row, consists of *keywordID* vectors, representing matched patterns, and *next-state* vectors (pointers), representing the directed edges. For example, state 5 in Fig. 2(a) denotes possible matches to patterns p1, p4 and p9, thus in Fig 2(b), the keywordID field at row 5 has bit 1, 4 and 9 set to '1'. Such a bit vector will be AND-ed with vectors identified from other PEs to generate the final result. While this example state machine has 21 states, we found that the largest state machine for Snort rule set contains 402 states, which requires the PEs be able to hold the worst case 512 (2^9) vectors for ASIC based designs. This introduces substantial memory wastage because not all the state machines can fill up the storage for worst-case scenario.

3. MOTIVATION OF THE RESEARCH

We are interested to study the memory usage of the keywordID and next-state matrix and find out how these two fields contribute to the total memory usage of state machines. We again use Snort pattern set and divide it into subsets in our study. The number of the subsets, and subsequently the number of parallel matching engines, are inversely related to the size k (i.e. the number of patterns of a subset). For each subset, a state machine is constructed and stored in the memory space, and a pattern matching engine is assigned to the state machine. Note that ASIC and FPGA based designs have to handle the worst case state machine, i.e., its memory module has to be large enough to accommodate a state machine with the maximal number of states.

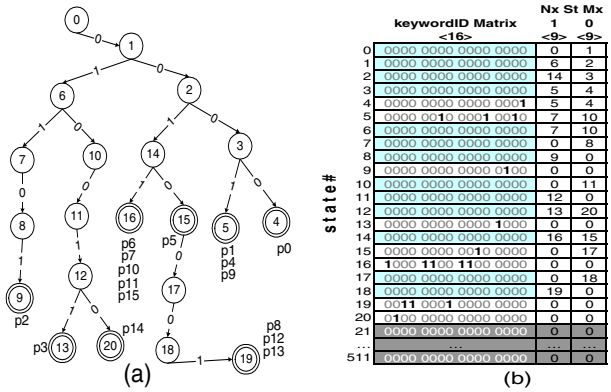

Figure 2: An example of a bit-level state machine and its memory layout.

We plot the memory consumption breakdown in Fig. 3 (a). We study how the total memory usage (y axis) changes when k (x axis) varies from 8 to 256. The upper and lower segments of the bars represent the memory usage of keywordID matrix and next-state matrix, respectively. One interesting observation is when k increases, the next-state matrix memory drops; but the keywordID matrix increases rapidly. The majority of the memory is spent on storing keywordIDs. Thus, it is indispensable to reduce the memory usage of keywordID matrix to minimize the total memory usage.

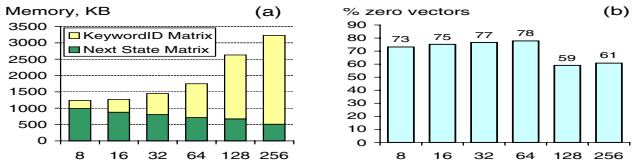

Figure 3: (a) Memory usage breakdown for keywordID and next-state matrix. (b) Percentage of zero vectors in keywordID matrix

4. OPTIMIZATION SCHEMES

In this section, we present an optimization technique to organize state machines memory in a compact fashion. The technique consists of two steps: state re-labeling and memory partition.

4.1 State Re-labeling

We are motivated by the results in Fig. 3 (a) to study compact organization of keywordID vectors. We revisit Fig. 2 and find that the keywordID vectors actually contain a large number of *zero* vectors, indicating no matching at all. These vectors correspond to the intermediate nodes along the trie before reaching any double-circled nodes. In the classical Aho-Corasick (AC) algorithm, all the nodes in the trie are

labeled in the sequence when they are created. As a result, the *zero* vectors are interleaved with the non-zero vectors as shown in Fig. 2(b).

We show the percentage of zero vectors among all the keywordID vectors in Fig. 3(b). These results are obtained with Snort pattern set. As we can see, the percentage of zero vectors is consistently high, 59.1% and more, throughout the range of k under study. This implies that compression or elimination of such zero vectors can significantly reduce the memory usage.

We propose to *re-label* the states after the trie is constructed. We re-label the double-circled nodes (matching nodes, or output states) in the trie shown in Fig. 4 such that none of their identifiers is larger than those of single-circled intermediate nodes. As a result, when storing the state machine in a linear memory space, the keywordID vectors corresponding to the matching nodes are clustered at the beginning of the keywordID matrix, shown in Fig. 4. At the same time, the structure of the trie is maintained to ensure proper state migration.

The state re-labeling is performed after the initial trie structure is created using Aho-Corasick algorithm, or actually any other state machine based algorithms. The following pseudo code describes the proposed re-labeling algorithm. The input of the re-labeling algorithm is a table mapping from patterns to the states, either an intermediate state or a matching state. Such a table is provided by the original AC or other state machine based algorithms. The output of re-labeling is an updated mapping table. It can been seen that complexity of the re-labeling process is $O(n)$, i.e. re-labeling introduces little overhead.

```
k is the amount of patterns per subset;
p(i) is the label of the pattern i;

1. FOR i = 0 to k - 1:
2.     Find the matching state of pattern i.
3.     IF this matching state has already been relabeled
4.         Do nothing.
5.     ELSE
6.         Re-label this state as i.
7. ENDFOR
8. Re-label the rest states (intermediate states.)
9. Connect the directed edges of the relabeled
   automaton by traversing the original trie.
```

4.2 Memory Partition

After re-labeling, we propose to separate the storage of keywordID matrix from the storage of next-state matrix. This is based on the observation that the number of non-zero keywordID vectors is bounded by the number of patterns in the subset. When the AC algorithm constructs a trie, each pattern will be mapped to a double-circled node, and only be mapped to that double-circled node (i.e. the matching node). That is, the following assertion holds.

$$N_{keywordID} \leq N_{rules}$$

Therefore, only a $k \times k$ matrix is needed to store keywordID vectors, where k is the amount of patterns in the subset. This observation is particularly helpful in reducing the local memory needed for parallel subset engines shown in Fig. 1.

On the other hand, non-null next-state vectors can be arbitrarily more than non-zero keywordID vectors since there can be an arbitrary number of intermediate or non-matching nodes in the trie. While these intermediate nodes are necessary to maintain the trie structure, their associated keywordID vectors are all zeros (e.g. state 0 and 1 in Fig. 2). In regular memory organization, the keywordID and next states are tightly coupled. We propose to separate the storage of keywordID matrix from next-state matrix, as shown in Fig. 4. The $k \times k$ bit matrix is used to store

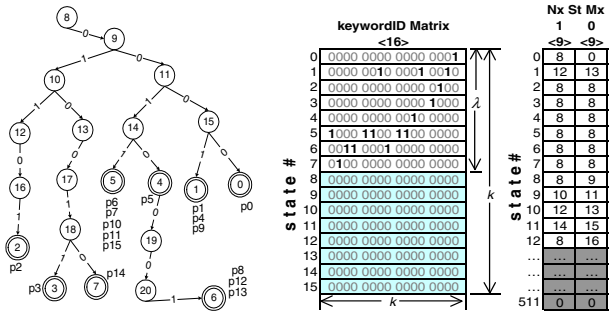

Figure 4: An example bit-level state machine *after* re-labeling, and its separated memory layouts.

keywordID matrix, and the next state matrix resides in an independent memory module. Thus, the zero keywordID vectors of the intermediate states are eliminated. We call this technique "k-Square".

In fact, the $k \times k$ bit matrix can contain zero vectors because some matching nodes can contain more than one pattern. In such a case, we need only λ matching nodes, where $\lambda \leq k$. This gives opportunities to further reduce the memory usage. We can store only λ non-zero keywordID vectors in one shared global memory module. We call this technique "k-Lamda".

5. ARCHITECTURE

We present the architectural design based on the above proposed optimization techniques in this section.

The State Processing Element (SPE) is shown in Fig. 5. A SPE consists of a Next State Decision Unit (NSDU), an Output State Detector (OSD), and memory modules storing next-states matrix and keywordID matrix. The NSDU supplies the current state to next-state matrix to obtain the next state based on the input bit. Meanwhile, the keywordID vector corresponding to the current state will be fetched from the keywordID matrix. The OSD determines outputting the keywordID vector if it is a valid one.

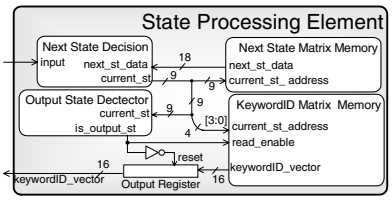

Figure 5: State Processing Element.

Fig. 6 (left) depicts the Next State Decision Unit, which is responsible for reaching the next state based on the current state and input bits. The Current State Register hold the current state. The value is used as an address to access the row in the next-state matrix memory. In the next cycle, the next-state matrix will give the next states (2 next states in this example) as its output, which are fed to the Holding Register. The multiplexer in the figure will determine the next state based on the 1-bit input from the data stream. Re-labeling may change the label of the entry state from its default value '0'. For example, the label of the entry state in Fig. 2 is '0'. After re-labeling, its label is changed to '8' in Fig. 4. Thus, the system needs to load the re-labeled initial state when the matching procedure starts.

Fig. 6 (right) depicts Output State Detector. After re-labeling states, the label of matching states (double circle nodes) are always less than k. For example, k=16, their label values are 0, 1, 2, up to 15. With this property, we can use a comparator to detect whether a current state is an matching state. If so, the Output State Detector will enable the keywordID matrix mem-

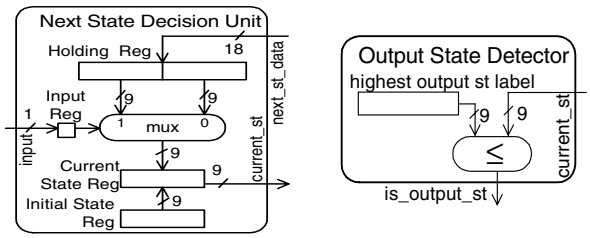

Figure 6: Next State Decision Unit, and Output State Detector.

ory in Fig. 5. If not, the detector will reset the Output Register indicating no match. The benefit is that the detector reduces the amount of accessing to the keywordID matrix memory, leading to lower power consumption and less likelihood of performance bottleneck. As a result, the keywordID matrix is suitable to be put in a shared global memory.

6. PERFORMANCE EVALUATION

6.1 Memory Reduction

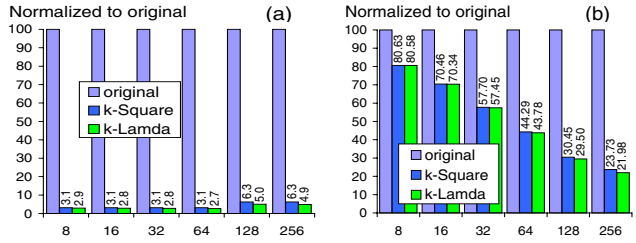

Figure 7: (a) KeywordID memory reduction, and (b)total memory reduction with the proposed optimization techniques.

We first compare the memory utilization of regular state machines with those optimized with state re-labeling and memory partition. We use Snort pattern set to evaluate the memory usage. Fig. 7 (a) shows the normalized memory consumption of keywordID matrix only, where "k-Square" refers to the $k \times k$ matrix and "k-Lambda" refers to the $k \times \lambda$ matrix. It can be seen that up to 96.9% memory is saved by "k-Square" and 97.2% by "k-Lambda". Fig. 7 (b) depicts the normalized total memory usage (keywordID + next-state matrix). The memory reduction is again significant. Up to 20% saving is achieved when k is 8. As k increases, the memory saving increases as the keywordID matrix contributes more in the original state machine storage. When k is 256, the memory reduction reaches over 80% with the "k-Lambda" scheme.

6.2 Comparison with Existing Work

We then compared the memory usage of our optimized storage of state machines with the results from previous research such as Tan-Sherwood's [10] and Brodie-Cytron-Taylor's [3]. Approach in [3] is based on pipelined state machine and just appeared recently. A significant improvement is shown on the memory density ($characters/mm^2$) over Tan's (however, they use newer SRAM technology 65nm instead of 130nm in Tan's.) We compare our compact state machines with Tan's and Brodie's methods and plot the memory requirement results in Fig. 8. We assume using the same SRAM technology and then translate Tan's and Brodie's results into the bytes needed for Snort pattern set. The memory required by Tan's approach is estimated through simulating their architecture. For Brodie's, we estimate from the density (423 $characters/mm^2$) and 6T SRAM (2 x $10^{-6} mm^2/bit$), both of which are from [3]. Hence, Brodie's approach supports 846 x 10^{-6} chars/bit. Snort set (as of April 2004) has 25616 chars, so the memory required in Brodie's is 3696.16KB. The figure shows that our proposed methods outperforms both Tan's and Brodie's. With "k-Square" matrix, the memory consumption is within 46% of [10] and 20.1% of [3], respectively. The "k-Lambda" matrix brings the memory usage to 38.7% of [10] and 16.9% of [3], respectively. The results can be explained as follows. The hardware design in [10] requires all the "tiles" used to store keywordID vectors having the same amount of rows as next-state matrix. In fact, however, a large percentage of the tiles are wasted for zero vectors. On the contrary, our approach uses the keywordID matrix memory as it is needed, i.e. no memory is wasted.

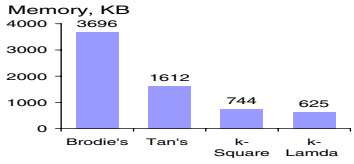

Figure 8: Memory requirement comparison with Tan's and Brodie's approaches.

7. CONCLUSION

This paper proposes novel schemes to reduce the memory required to store state machines in a linear memory space for pattern matching. We propose to *re-label* the states in a state machine such that the matching states are clustered at the beginning of a memory space, giving opportunities for compact storage. We then partition state machines and use separate memory modules to store keywordID and next-state matrixes. We present the architectural design using the proposed schemes. We evaluate the performance of the optimization strategies using realistic pattern sets from Snort [9]. The experiment results show significant reduction (up to 80.1%) on the memory consumption of state machines. This research can benefit the design of high-speed pattern matching machines or pattern matching co-processors for a wide range of applications such as network intrusion detection and virus scanning.

8. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] S. Artan and J. Chao. Multi-packet Signature Detection using Prefix Bloom Filters. In *IEEE Global Telecommunications Conference*, Nov 2005.

[3] B. C. Brodie, R. K. Cytron, and D.E. Taylor. A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching. In *ISCA*, Boston, MA, June 2006.

[4] C. R. Clark and D. E. Schimmel. Modeling the Data-Dependent Performance of Pattern-Matching Architectures. In *International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb 2006.

[5] Intel Corp. Intel IXP2400 Network Processor Product Brief, 2005.

[6] S. Kumar, S. Dharmapurikar, P. Crowley, J. Turner, and F. Yu. Algorithms to Accelerate Multiple Regular Expression Matching for Deep Packet Inspection. In *SIGCOMM 2006*, Pisa, Italy, 2006.

[7] Linux Layer 7 Packet Classifier. http://sourceforge.net/projects/l7-filter/.

[8] P. Piyachon and Y. Luo. Efficient memory utilization on network processors for deep packet inspection. In *Proceedings of ACM Symposium on Architectures for Networking and Communications Systems*, Dec 2006.

[9] Snort. www.snort.org, 2003.

[10] L. Tan and T. Sherwood. Architectures for Bit-Split String Scanning in Intrusion Detection. *IEEE Micro*, Jan-Feb 06.