Towards Efficient Security Policy Lookup on Many-Core Network Processing Platforms

WANG Xiang[1, 3], QI Yaxuan[2], WANG Kai[3], XUE Yibo[3, 4], LI Jun[3, 4]

[1]Department of Automation, Tsinghua University, Beijing 100084, China

[2]Yunshan Networks Inc., Beijing 100083, China

[3]Research Institute of Information Technology, Tsinghua University, Beijing 100084, China

[4]Tsinghua National Lab for Information Science and Technology, Beijing 100084, China

Abstract: Modern network security devices employ packet classification and pattern matching algorithms to inspect packets. Due to the complexity and heterogeneity of different search data structures, it is difficult for existing algorithms to leverage modern hardware platforms to achieve high performance. This paper presents a Structural Compression (SC) method that optimizes the data structures of both algorithms. It reviews both algorithms under the model of search space decomposition, and homogenizes their search data structures. This approach not only guarantees deterministic lookup speed but also optimizes the data structure for efficient implementation on many-core platforms. The performance evaluation reveals that the homogeneous data structure achieves 10Gbps line-rate 64byte packet classification throughput and multi-Gbps deep inspection speed.

Keywords: packet classification; pattern matching; algorithms; data structures

I. INTRODUCTION

Modern network security appliances require the inspection of both packet header and packet payload. Accordingly, packet classification and pattern matching algorithms play a key role in network security processing. Due to the flexibility and high performance, most network security systems are developed on many-core network processing platforms. However, the existing packet inspection algorithms are difficult to be implemented effectively on these platforms for two reasons.

On one hand, due to the employment of intricate heuristics, the data structures of existing algorithms are often complicated and heterogeneous. These data structures may have different sub-structures of various memory sizes, which often hamper the optimization of memory allocation and access. Besides, complicated data structure may hurt the instruction locality in software implementation. For example, the well-known decision tree based packet classification algorithm HiCuts [1] employs the heuristics of greedy search at each internal node to select the locally optimized number of cuttings, which makes different tree nodes have different number of child node pointers. Therefore, direct compression of the HiCuts data structure is difficult and the traversal of HiCuts nodes might require unstable I/O bandwidth.

This paper presents a Structural Compression (SC) method that optimizes the data structures of both algorithms. It reviews both algorithms under the model of search space decomposition, and homogenizes their search data structures.

On the other hand, the worst-case look-up time is not guaranteed because most algorithms trade search time for memory space. The decision tree based packet classification algorithms, such as HiCuts and HyperCuts [2], do not have explicit worst-case tree depth because they perform variable stride cutting at internal nodes and linear search at leaf nodes. The DFA based pattern matching algorithms, such as D^2FA [3] and its improved variation [4], are difficult to guarantee the number of memory access per input character because the number of inter-state and intra-state transitions are not strictly bounded. These limitations often cause unstable performance among different policy sets.

This paper optimizes the classic packet inspection algorithms for many-core network processing platforms. An effective Structural Compression (SC) method is proposed for both packet classification and pattern matching algorithms. The major contributions include:

1) Algorithm analysis: Based on the study of decision tree based packet classification algorithms, the paper investigates the relationship between the implementation of data structures and the model of search space decomposition. Two types of information are abstracted, i.e., *addressing information* and *partition information*. The addressing information defines the affiliation among different nodes, and the partition information defines the traversal paths for tree search. Most existing algorithms do not explicitly distinguish them in their implementations, and there might be spatial redundancy within the tree data structures.

2) Structural compression: After the separation of addressing and partition information, the paper introduces the *structural compression* approach that effectively reduces the spatial redundancy. In packet classification, the approach (SC-Tree) restricts the employment of heuristics in building search data structures and enforces fixed-stride cutting strategy. Subject to a two-stage compression of tree nodes, the spatial redundancy is reduced at each internal node and also is globally eliminated using a shared storage. Moreover, this paper extends the approach to DFA based pattern matching algorithms by adding relay states and a locality-aware encoding scheme. The structural compression approach for DFA (SC-FA) can guarantee the memory access times when looking up different policy sets, and uses homogeneous data structures for efficient memory operations.

3) Hardware evaluation: Both SC-Tree and SC-FA are optimized and implemented on a 64-core Tilera TILEPro64 [5] platform for hardware evaluation. Each tree node is compressed into one 64bit word and is aligned in consecutive memory. The searching procedure runs in parallel with linear speedup along with the increment of the processing core number. The results of performance evaluation demonstrate that the SC-Tree reaches 10Gbps line-rate 64byte packet classification speed, and the SC-FA achieves multi-Gbps pattern matching speed.

The rest of this paper is organized as follows. Section 2 analyzes the structural redundancy of the exiting algorithms. Section 3 and Section 4 present the structural compression algorithm for both packet classification and pattern matching algorithms. Section 5 describes the evaluation of the structural compression approach on many-core network processing platforms. Section 6 draws the conclusion.

II. Data Structure Redundancy

2.1 Two types of information

Figure 1 shows a typical HiCuts decision tree. Node-0 is the tree root, and its related 2-dimensional overall search space is equally cut into 4 unit-spaces on ***Y*** dimension. After aggregation, the 1st and the 2nd unit-space are mapped to the 1st and the 2nd subspace, respectively; the 3rd and the 4th unit-spaces are aggregated into the 3rd subspace. Similar decomposition is performed on node-1 ~ node-3. Node-4 ~ node-9 are leaf nodes as

the recursion is terminated. The *tree outline*, i.e., tree nodes and their traversal paths, and the *tree texture*, i.e., space decomposition pattern residing in internal nodes, consist in the decision tree. According to this example, there are two key points in a typical tree data structure:

Addressing: The addressing information is used for identifying nodes during decision tree building and traversal, which is determined by the tree outline. Multiple schemes can be employed to address tree nodes. For example, the general scheme is using node *pointer* or node global *index*. To reduce the memory requirement, the parent node can use the *base plus offset* scheme, which is based on its first child node address and the number of its child nodes.

Partition: The partition information includes cut dimension, cut stride (the number of cuttings) and aggregated subspace mapping. It texturizes the tree with possible search paths. The search procedure uses this information and packet header field value to select the next traversal node. Most algorithms decide the cut dimension sequence using different evaluation functions. Besides, it usually employs variable-stride cutting strategy during decision tree building, and generates different amount of unit-spaces and subspaces.

2.2 Structural redundancy

From the experimental results on real-life policies, it could be observed that the existing packet classification trees have a lot of structural redundancy, i.e., a large number of internal nodes have similarity in two aspects: 1) *the number of subspaces*; 2) *the mapping between unit-spaces and subspaces*. The first similarity is due to the fact that the fan-out of most internal nodes is small, i.e., most internal nodes have only a small number of child nodes. The second similarity is caused by the common practice of subnet address allocation and application port specification, i.e., most rules have the same source IP addresses, and popular network services, e.g., web / mail / DNS, bind well-known ports. In addition, standard application ports are limited to a small value range, and many rules have the same destination port. In Figure 1, the internal nodes have the following two similarities. First, all of node-1, node-2 and node-3 have 2 subspaces. Second, node-1 and node-2 have the same space mapping, i.e., the 1st, 3rd and 4th unit-spaces are aggregated to the 1st subspace, and the 2nd unit-space is mapped to the 2nd subspace. Although these similarities can be found in most internal nodes, they have not been exploited by the existing algorithms. *Most implementation adopts the array of pointer to store both addressing and partition information. Different nodes are addressed with different pointers, which hamper the elimination of the partition similarity.*

Based on the above analysis, it is required to separate the addressing and partition information. After decoupling, the partition information can be independently expressed. And it is free to employ different compression techniques to extract the most significant information. Overall, it is the decoupling that

Table I *A 2-field policy with 4 rules*

Rule	Priority	X-field	Y-field	Action
R1	1	[0100,0111]	[0000,0011]	act1
R2	2	[0000,1111]	[0101,0101]	act1
R3	3	[1000,1111]	[1000,1111]	act1
R4	4	[0000,1111]	[0000,1111]	act2

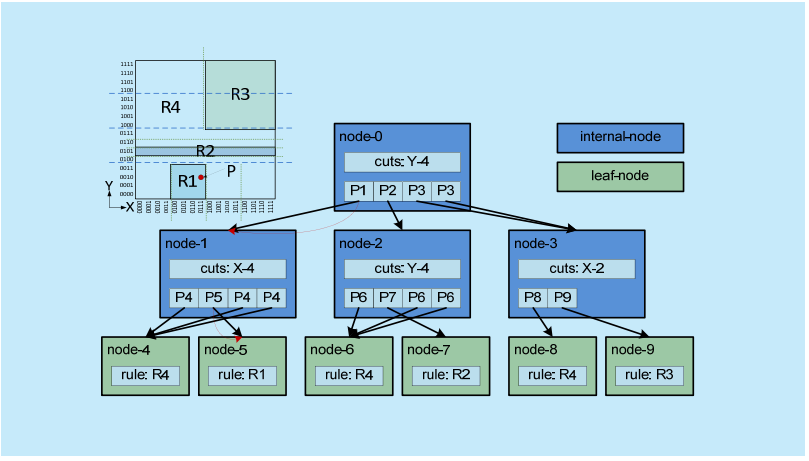

Fig.1 *Geometric view and HiCuts tree for policy in Table 1*

opens the way to structurally compress the spatial redundancy in data structures.

III. Structural Compression for Packet Classification

3.1 Decision tree based packet classification

Many existing packet classification algorithms are based on decision tree data structure. These algorithms partition the overall search space hierarchically into subspaces until each of subspaces contains only a few rules. At each internal tree node, the current search space is partitioned into a certain number of subspaces via equal-sized cutting(s) on selected packet header field(s). The number of cuttings and the cutting field(s) may be determined by policy-related heuristics. The space decomposition terminates at each leaf node.

As an example, Table 1 shows a 2-field policy with 4 rules. Each rule has range specifications on both ***X*** and ***Y*** fields. The priority of rules is in decent order. ***R***4, the default rule, overlaps ***R***1, ***R***2 and ***R***3. The left part of Figure 1 is the geometrical view these rules. All rules are in the entire search space ***S*0**={x∈[0000,1111], y∈[0000,1111] }. The objective of classifying a packet ***P***={x=0111,y=0010} is to find the rectangle with the highest priority that contains ***P***. The right part of Figure 1 shows the HiCuts tree as an example classifier. At the root node, the ***S0*** is partitioned into 4 unit-spaces using equal-sized cuttings on the ***Y***-field:

***U*1**={x∈[0000,1111],y∈[0000,0011]}
***U*2**={x∈[0000,1111],y∈[0100,0111]}
***U*3**={x∈[0000,1111],y∈[1000,1011] }
***U*4**={x∈[0000,1111],y∈[1100,1111]}

According to HiCuts algorithm, ***U*3** and ***U*4** are aggregated into a single subspace, because they contain the same sub-rule set of rules {***R***3, ***R***4}. Thus, at the tree root, the ***S0*** is partitioned into 3 subspaces:

S1={x∈[0000,1111],y∈[0000,0011]}
S2={x∈[0000,1111],y∈[0100,0111]}
S3={x∈[0000,1111],y∈[1000,1111]}

After space partition, three child nodes are created. A pointer array is allocated and stored in the root node to map ***U1*** ~ ***U4*** to ***S1*** ~ ***S3***. By performing similar space partition at node-1, node-2 and node-3, the ***S0*** is further partitioned into six subspaces:

S4={x∈[0000,0011]∪[1000,1111], y∈[0000,0011]}
S5={x∈[0100,0111],y∈[0000,0011]}
S6={x∈[0000,1111], y∈[0100,0100]∪[0110,0111]}
S7={x∈[0000,1111],y∈[0101,0101]}
S8={x∈[0000,0111],y∈[1000,1111]}
S9={x∈[1000,1111],y∈[1000,1111]}

As each subspace of ***S***4 ~ ***S***9 is *fully covered* by a specific set of rules, the rule with the highest priority in the set is the final classification result. Therefore, node-4 ~ node-9 are leaf-nodes and no space partition is needed.

Compared to HiCuts, HyperCuts applies equal-sized cuttings on multiple fields simultaneously to reduce the average depth of decision tree. EffiCuts [6] divides the original classifier into a set of sub-classifiers to reduce the memory usage. Although they use different strategies to achieve the better tradeoff between time and space, the basic structure of packet classification trees are similar.

3.2 SC-Tree for packet classification

Three steps are taken to obtain the SC-Tree. The first step is to separate the partition information from the addressing information at each internal tree node. Then the per-node partition information is compressed using a two-stage space aggregation technique. In the third step, only unique per-node partition information is extracted and then shared among all internal nodes. After these three steps, all pointers are eliminated, and a compact SC-Tree data structure is obtained.

STEP*-1**: *Separation of two types of information*. All nodes are restricted to fixed memory size, and all child nodes of an internal node are stored in consecutive memory. Assume each node has the size ***N_SIZE

and the number of cuttings ***N_CUTS***. Using children base plus offset addressing scheme, the i^{th} pointer ***Pi*** in the pointer array can be rewritten as follows:

$$Pi=P0+offset[i]*N_{SIZE}, 0 \le i < N_{CUTS}$$

where ***P0*** is the address of the first child node, and ***offset***[***N_CUTS***] stores the offset of all child nodes. As shown in the top of Figure 2, a pointer array with 8 pointers can be separated into two parts. One is the base address and the number of child nodes. The other is the offset array. The former represents the addressing information, and we can count on this information alone to access all child nodes. The latter is the partition information, as it only uses local offsets for space mapping. And it does not contain any addressing information. Since the length of pointer array and the maximum value in offset array are equivalent, the former can be eliminated to optimize the memory usage.

As the partition information has been abstracted and is independent from the addressing information, it is possible to remove the structural redundancy of data structures. Unfortunately, the fan-out of a tree is small, and most offset arrays are very sparse when the number of cuttings is large. Direct redundancy removal based on these offset arrays is not efficient. As a consequence, in the second step, the local redundancy in each offset array will be first removed.

STEP-2: *Compression of partition information*. At this step, the offset arrays are compressed using the bitmap technique [7]. Given an *offset array* $OA[k]$, $0 \le k < K$, a K-bit bitmap and an M-element offset list $OL[m]$, $0 \le m < M \le K$ can be generated by taking following steps:

- Clear the first bit of the bitmap, let $m = 0$, and set $OL[m] = OA[0]$.
- For each $1 \le k < K$, if $OA[k]$!= $OA[k+1]$, set the $(k+1)$th bit of the bitmap, let m += 1, and set $OL[m] = OA[k]$; else clear the $(k+1)$th bit of the bitmap.

The bottom of Figure 2 shows an example of bitmap compression, and it can be seen that an 8-element offset array is compressed into a 4-element offset list using an 8-bit bitmap.

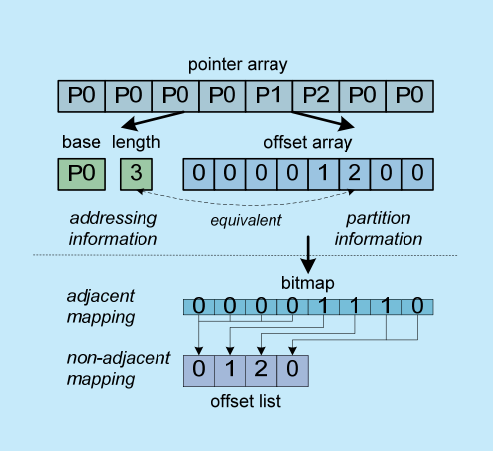

Fig. 2 *The separation and compression of two types of information*

STEP-3: *Elimination of partition redundancy*. After the first two steps, there is an important observation in packet classification trees: *The number of unique bitmaps and the number of unique offset lists are both small compared to the one of tree nodes*. In other words, many nodes have the same bitmaps or offset lists. According to the experimental results on real-life policies, the numbers of unique bitmaps and offset lists are both at least 1~2 *orders* less than the one of tree nodes. As an example, by replacing all pointer arrays in Figure 1 with bitmaps and offset lists and using fixed-stride 4-cuttings, the decision tree shown in Figure 3 could be obtained. There are only 2 unique bitmaps (0110 and 0010) and 3 unique offset lists (012, 010 and 01) among all 4 internal nodes. As the offset list 01 in node-3 is a prefix of the offset list 012 in node-0, the latter can be also used as the offset list of node-2. Thus, the number of unique offset lists is reduced to 2.

Both unique bitmaps and offset lists are extracted from all internal nodes and shared among them. After that, each internal node only needs to store two indices (*bmpID* and *offID*) to index the shared bitmap table and offset list table. Besides, the base pointer in each node can also be replaced with the first child node *global ID*, e.g., ***N1***, to save memory usage. Figure 4 shows the final SC-

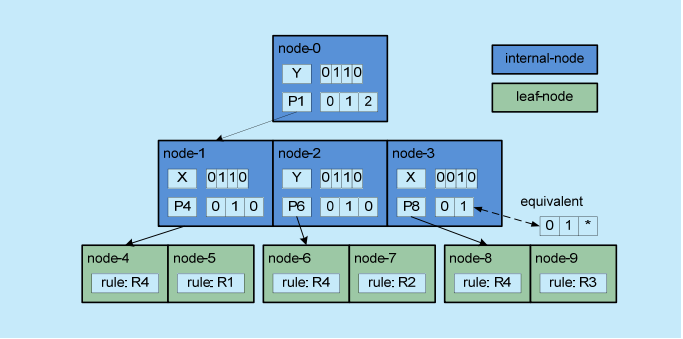

Fig. 3 *Packet classification tree with bitmaps and offset lists*

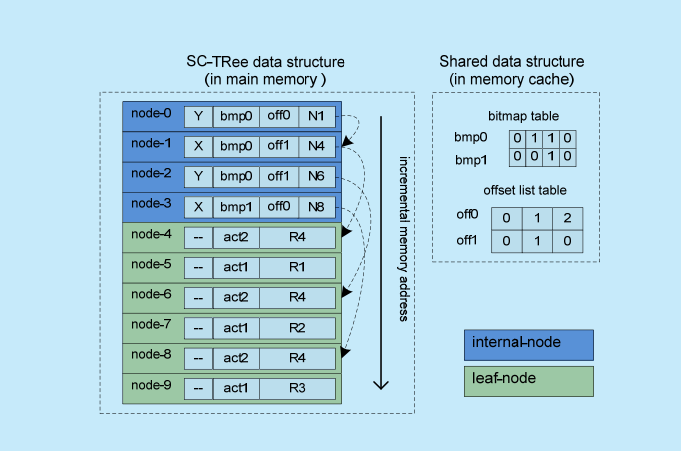

Fig. 4 *SC-Tree for policy in Table 1*

Tree data structure.

To search the SC-Tree data structure, we start from the root node and then traverse down the tree through a series of internal nodes until reach a leaf node. For example, to classify packet ***P***={x=0111,y=0010} by searching the SC-Tree shown in Figure 4, it takes the following steps: At the root node, as we use fixed-stride 4-cuttings and the first two bits of value on cutting dimension ***Y*** is 00, the packet falls into the 1st unit-space. The procedure counts the number of 1s of the first 1 bit of ***bmp*0**, and gets the result 0. It reads the next node whose ID is ***N*1** + ***off*0[0]**, i.e., node-1. Similarly, at node-1, as the first two bits of value on cutting dimension ***X*** is 01, the packet falls in the 2nd unit-space. The procedure counts the number of 1s of the first 2 bits of ***bmp*0**, and gets the result 1. It reads the next node whose ID is ***N4*** + ***off1***[1], i.e., node-5. As node-5 is a leaf node, the procedure can find the best matched rule ***R*1** with action ***act*1** associated.

IV. Structural Compression for Pattern Matching

4.1 DFA based pattern matching

Different from packet classification, which is range matching on a limited number of packet header fields, pattern matching conducts byte-level matching on an unlimited number of packet payload. The number of inspected bytes is the overall size of a flow. Many existing pattern matching algorithms are based on DFA. A DFA consists of a finite set of input symbols, denoted as Σ, a finite set of states, and a transition function δ. In network applications, $|\Sigma| = 256$, which is the size of extended ASCII symbols. Among these states, there is a single initial state and a set of accepting states. The transition function δ takes a state and an input symbol as the input and returns the next state as the output.

The left part of Figure 5 shows an example DFA for matching 2 regular express1ions ***RE1*** /.*be+/ and ***RE2*** /.*dad/ over the symbol set Σ = {a, b, c, d, e, f, g, h}. For any input symbol c∈Σ and state ***Si***, the next state ***Sj*** can be found by following the transition labeled with the symbol ***c***. All transitions to ***S0*** are not shown for clarity. To lookup the DFA, we can use the DFA transition table shown in the right part of Figure 5. ***Pi*** is the memory address of ***Si***. Given an input character stream "*babedad*" and the initial state ***S0***, it will take the following steps to lookup the transition table:

$$S0 \xrightarrow{b} S1 \xrightarrow{a} S0 \xrightarrow{b} S1 \xrightarrow{e} S3(RE1) \xrightarrow{d} S2 \xrightarrow{a} S4 \xrightarrow{d} S5(RE2)$$

4.2 Structural redundancy in DFA

According to the existing studies, the DFA transition table has a lot of redundancies [3] [4]. Based on the observation of real-life DFA transition tables, the redundancy can be categorized in two classes [8]:

Intra-state redundancy: The number of

unique transitions in a certain state is usually small, because frequently used characters in pattern matching rules are a small subset of the ASCII symbol set. Therefore, most states have only a few unique transitions at these symbols.

Inter-state redundancy: A set of states are usually of similar transitions, because many state transitions point to several common failure states [9]. Besides, some unambiguous states may be replicated when automata is combined [10].

Figure 5 shows that at least 5 out of 8 transitions are identical in each state, which point to ***P0***. In the group of states {***S0***, ***S2***, ***S5***}, 7 out of 8 transitions in each state are identical to one another. Such intra-state and inter-state redundancies are more apparent in real-life DFA transition tables.

4.3 SC-FA for pattern matching

In pattern matching, DFA can be regarded as a hierarchical data structure: the initial state is viewed as the root node; all states next to the initial one are viewed as child nodes of the root. The significant difference between DFA and decision tree is that DFA transition may point to upper nodes (previous states). These backward transitions implicate the next and the preceding space will share the same partition pattern.

The proposed structural compression method is extended to DFA based pattern matching by taking the following steps. Firstly, the DFA graph is converted into a 256-stride tree by adding relay states. Then an offset encoding technique is employed to reduce the number of relay states by exploring the statistical distribution of relay state transitions. After that, the SC-FA data structure that eliminates the redundancies in DFA transition tables is obtained.

***STEP*-1**: *Introduction of relay states*. Each transition in the DFA graph belongs to either a descent transition (solid arrow in Figure 5) or a relay transition (dashed arrow in Figure 5). The existence of relay transitions

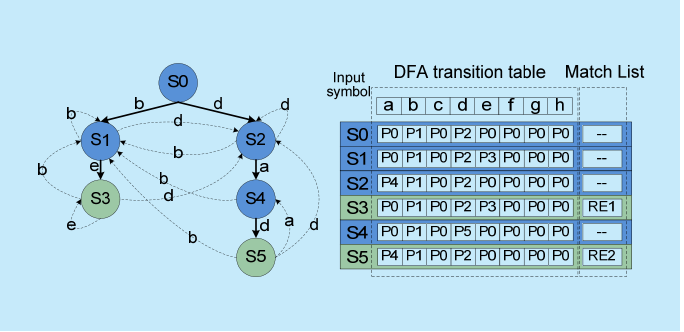

Fig. 5 *DFA graph and transitions for /.*be+/ and /.*dad/*

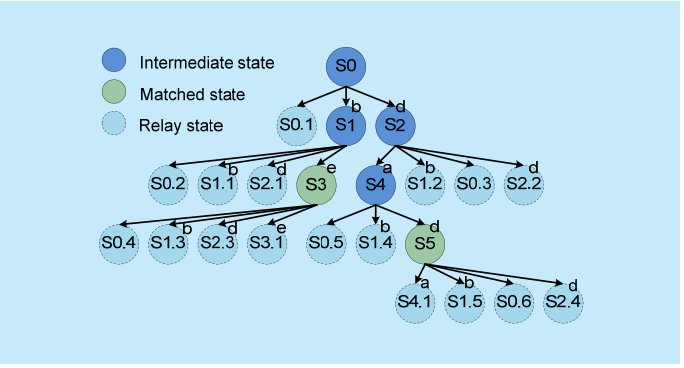

Fig. 6 *Adding relay states in the DFA graph*

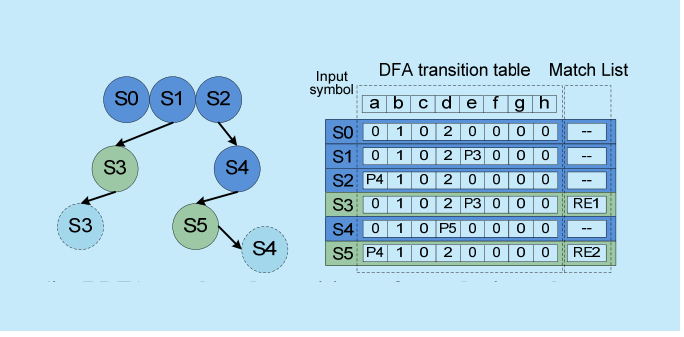

Fig. 7 *DFA graph and transitions after reducing relay states*

makes it impossible to use the base plus offset addressing scheme to extract partition information, because child states are stored in non-consecutive memory space. To support incremental addressing of child states, the additional relay state is introduced. Each relay state is a state replication corresponding to its relay transition. After adding relay states, all relay transitions can be replaced by descent transitions, and the DFA graph turns into a tree-like structure. Each state is similar to an

internal node of the packet classification tree, and transition pointers are analogous to the pointer array. Figure 6 shows the DFA graph after adding relay states. In this figure, it could be observed that all relay transitions have been removed by adding relay states, i.e., ***Si.j*** indicates the j^{th} duplicates of ***Si***. Because each relay state is identical to its corresponding state, the search of the DFA graph maintains the property of one character per state traversal. For example, given the input byte stream "*babedad*" and the initial state ***S0***, it will take the following steps to lookup the DFA graph in Figure 6:

$$S0 \xrightarrow{b} S1 \xrightarrow{a} S0.2 \xrightarrow{b} S1 \xrightarrow{e} S3(RE1) \xrightarrow{d} S2.3 \xrightarrow{a} S4 \xrightarrow{d} S5(RE2)$$

STEP-2: *Reduction of relay states*. To achieve a better compression ratio, the number of relay states is reduced before applying the structural compression. Based on the observation that *most relay transitions point to only a small number of states*, i.e., most relay states are replication of only a few states, an index encoding strategy could be introduced to significantly reduce the number of relay states. This encoding method takes the following steps:

- Reorder all DFA states according to the access frequency in descent order. This frequency can be obtained by counting the total number of a certain transition in original transition tables.
- Store the first ***M*** states in consecutive memory. Thus, these states can be directly accessed when given an index of the value, i.e., 0 ~ ***M*** - 1.
- Replace state transitions that point to the first ***M*** states with the state index.

For example, Figure 5 reveals that ***S0***, ***S1*** and ***S2*** are the top three most frequently visited states, which means ***M*** = 3. If the state index is employed to access them, their corresponding relay states can be removed from Figure 6. The resulting DFA graph and transition table are shown in Figure 7. Note that, because ***S*3** has only one child state, the transition pointer ***P*3**, rather than ***P*3.1**, can be directly used to relay state ***S*3.1**. Thus, the relay state ***S*3.1** can be removed in the transition table. Similarly, ***S*4.1** can be also eliminated, as it has the only child state of ***S*5**.

STEP-3: *Elimination of structural redundancy.* After transforming the original DFA and reducing the relay state overhead, the structural compression is employed to build the final DFA graph. As the indices of the first ***M*** states are used for *direct* accessing, the corresponding values in the offset array are the real global state IDs, rather than the offsets to the children array base. Except for these states, other decent states are viewed as siblings, and the corresponding values in the offset array vary from ***M*** ~ ***M*** + 255. In most cases, the number of decent states is small, so the offsets are also in a small range as well. After that, we also obtain the similar observation compared with the real-life packet classification trees: *The number of unique bitmaps and the number of unique offset lists are both small compared to the one of DFA states*. Thus, the unique bitmaps and offsets are extracted from all DFA states and shared. Besides, the pointer in each state can be replaced with its global state ID, and all states are shaped with the same size. Figure 8 shows the final SC-FA data structure.

To search the SC-FA data structure, we start from the initial state and traverse in DFA graph through multiple transitions until input symbols are all processed. For example, it takes the following steps to match the input character stream "*babedad*", by searching the SC-FA shown in Figure 8. At the initial state, as the first input character is 'b', the procedure falls in the 2nd unit-space. It counts the number of 1s of the first 2 bits of ***bmp0***, and gets the result 1. Because we regard ***S0***, ***S1*** and ***S2*** as three most visited states and the value of ***off0***[***1***], i.e., 1, is smaller than ***M***, it directly reads the next state whose ID is 1, i.e., ***S1***. Similar processing is performed until the current input character is 'e', and the current state is traversed to ***S1***. At ***S1***, as the input character is 'e', the procedure falls in the 5th unit-space. It counts the number of 1s of the first 5 bits of ***bmp1***, and gets the result 4. Because the value of ***off1***[4], i.e., 3, is not smaller than ***M***, it read the next state

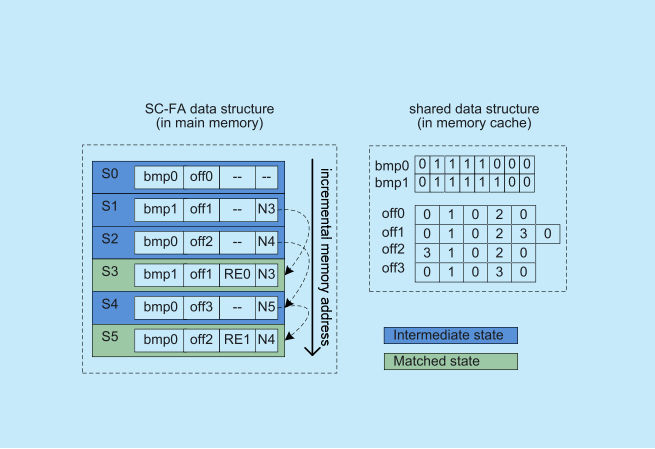

Fig. 8 *SC-FA of pattern matching in Fig. 5*

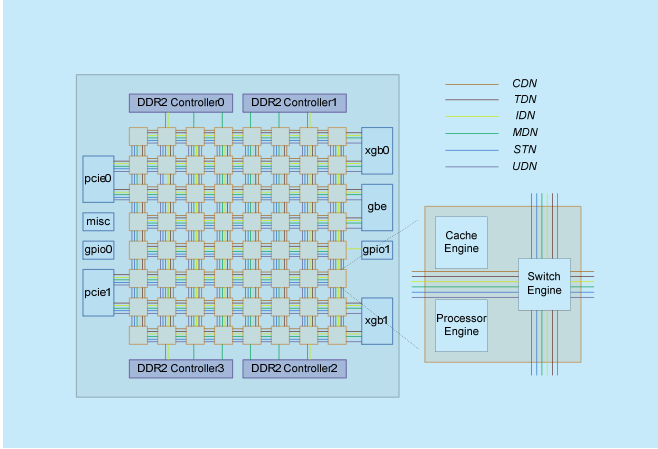

Fig. 9 *Tilera TILEPro64 architecture [12]*

whose ID is ***N3*** + ***off*1**[4] - M, i.e., ***S*3**. At ***S*3**, the procedure can find the matched ***RE*0**. The procedure goes through these two types of state addressing until all inputs are processed.

V. Many-Core Optimization and Performance Evaluation

5.1 Many-core platform

Among network security devices, high-end products may leverage ASIC or FPGA chips to perform critical packet processing. It can achieve extreme high processing speed, but lacks of flexibility to extend its functionality. As a result, most commodity network security devices are built on many-core network processing platforms, which are developed with various acceleration engines that are optimized for packet manipulation specifically, such as DFA thread engine for packet payload inspection, ingress/egress packet processor for high-speed packet parser and packet order unit for flow-level order preserving. Besides, these processors also have the specific instruction sets for security and compression processing, such as cryptographic and SIMD instructions.

Tilera TILEmpower appliance [11] is a many-core network processing platform. It can perform high-speed packet processing with low clock rate and low power consumption. It includes a Tilera TILEPro64 processor, providing Linux programming environment and optimized dataplane processing library. Figure 9 shows the high-level architecture of Tilera TILEPro64. The processor has 64 full-meshed processing tiles. The L2 cache in all processing cores can be configured to be accessed from other processing cores in various forms. Thus, all L2 caches can form a distributed L3 cache, which is able to reduce the access latency of main memory. All 64 cores are interconnected by six inter-tile on-chip networks, and three of them are software programmable for low-latency communication among processing cores. It also has four on-chip memory controllers that can address up to 16GB shared memory. All controllers can be configured for independent access or be striped for system controlled load balance in memory intensive scenarios. Two 10Gbps XAUIs are attached as network IO interfaces.

5.2 SC-Tree / SC-FA optimization

In high-speed processing systems, the data ought to be stored / loaded at alignment address and be processed in processor word length. According to the complexity of different policies, SC-Tree / SC-FA are able to limit the bits allocated for different usage within one word. It can be observed that each internal node has four different fields: the cutting dimension, the bitmap index, the

offset index and its first child index, which are determined by the dimension number of rules, the number of bitmap, the number of offset and the number of tree nodes, respectively. Based on the statistics analysis of public available 5-dimensional policies, these four types of information can be stored in one 64bit word with each field of 3bits, 12 bits, 9 bits and 40 bits, respectively.

Table II *Partition patterns in decision trees*

rules	total	uni bmp	uni off	agg off
ACL_1K	2548	640	64	29
ACL_5K	9547	1671	266	136
ACL_10K	27543	2705	369	200
FW_1K	304185	703	109	49
FW_5K	1585.3K	2612	293	128
FW_10K	4818.4K	3713	499	253
IPC_1K	75931	776	71	31
IPC_5K	808348	2258	237	111
IPC_10K	2122.1K	3968	331	153

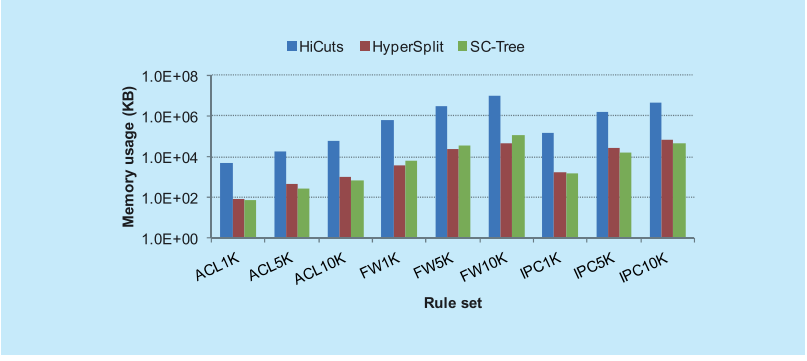

Fig. 10 *Memory usage of HiCuts, HyperSplit and SC-Tree*

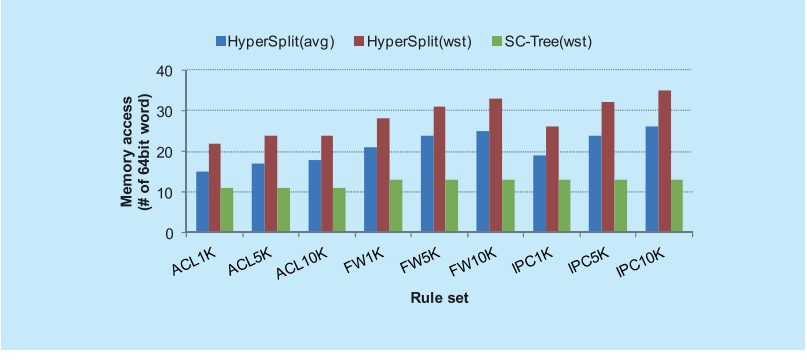

Fig. 11 *Memory access of HyperSplit and SC-Tree*

Parallel optimization: The structural compression approach is deployed in data parallelism mode. All available cores execute the same processing logic of policy lookup, and received packets are distributed among these cores at flow granularity. Ideally, the system throughput could achieve the linear speedup along with the increment of processing core number.

Cache optimization: The structural compression approach can guarantee the deterministic search speed among all policies, but it needs two extra memory accesses to fetch the bitmap and offset. To achieve high system throughput, the cache scheme of bitmaps and offsets are configured differently from the one of tree nodes. In TILEPro64 processor, each physical memory address could be configured with or without coherence guarantee. In coherence manner, the data structures could be cached in the L2 cache of all tiles. Besides, to avoid the jitter of processing speed, the coherence scheme could be configured as *hash-for-home*, where the coherence maintenance of one memory page is evenly distributed among all processing tiles. In incoherence manner, the data structures are locally cached, which could not leverage the cache of other tiles. For SC-Tree / SC-FA optimization, the bitmaps and offsets are configured in coherence manner with *hash-for-home* scheme. And they are also set with high priority to prevent cache eviction. The tree nodes are configured in incoherence manner, as its randomly accessed behavior is difficult to benefit from caching.

Memory optimization: The TILEPro64 processor supports huge page. And the size of one huge page is 16MB, against 4KB for the one of default memory page. This optimization could significantly reduce the TLB misses. All data structures of SC-Tree / SC-FA employ huge pages for storage. Besides, the packet buffer is also configured to use huge pages. The packet buffer is allocated on the memory controller which is the nearest one to the network IO interfaces, i.e., the 1st and

2nd controllers, to reduce the transmit latency. As the search procedure of SC-Tree / SC-FA consumes large volume of memory bandwidth, all controllers are configured in strip manner. Thus, the memory accesses of data structures are evenly scattered to achieve better balance of the memory IO pressure.

Instruction optimization: The structural compression approach heavily relies on the function of counting '1' in one bitmap. In generic software implementation, the function counts the number in loop manner, which needs multiple CPU cycles. The TILEPro64 processor provides multiple instructions for bit manipulation. The *pcnt* instruction is used for hardware acceleration, which could accomplish the counting of one word in single CPU cycle.

5.3 Performance evaluation

5.3.1 Evaluation data sets

The SC-Tree rule sets contain three types of rules: Access Control List (ACL), Firewall (FW) and IP Chain (IPC), generated by ClassBench [13]. Each type of rule sets includes the scales of 1K, 5K and 10K rules.

The SC-FA signature sets include two regular expression sets (snort24 and snort40) and two string sets (short8 and short120) from Snort [14], one regular expression set (bro217) from Bro [15] and two regular expression sets (linux13 and linux30) from L7-filter [16].

5.3.2 SC-Tree evaluation

5.3.2.1. Partition redundancy in decision trees
The number of search space partition patterns stored in data structures is measured to illustrate the partition redundancy in packet classification trees. The HiCuts algorithm is employed to build original decision tree. According to the requirement of structural compression technique, 256-stride cutting is conducted at each internal node. And the number of rules at leaf node is set to 1, which forbids the linear-search at leaf node. Table 2 shows the number of partition patterns stored in HiCuts-Tree and SC-Tree. As each internal node of HiCuts-Tree stores the partition information, the number of partition patterns equals to the one of internal tree nodes in HiCuts-Tree, which is shown in *total* field of Table 2. According to the complexity of search space on different rule sets, the number of partition patterns in HiCuts-Tree varies in a large range. After extracting bitmaps and offsets from internal nodes, the number of unique bitmaps (*uni bmp* in Table 2) is reduced up to 0.1% of the one of partition patterns in HiCuts-Tree. And the number of unique offsets (*uni off* in Table 2) is reduced even more. Besides, if aggregating offsets that share the same prefix, the number of unique offsets (*agg off* in Table 2) is further reduced by about 50%.

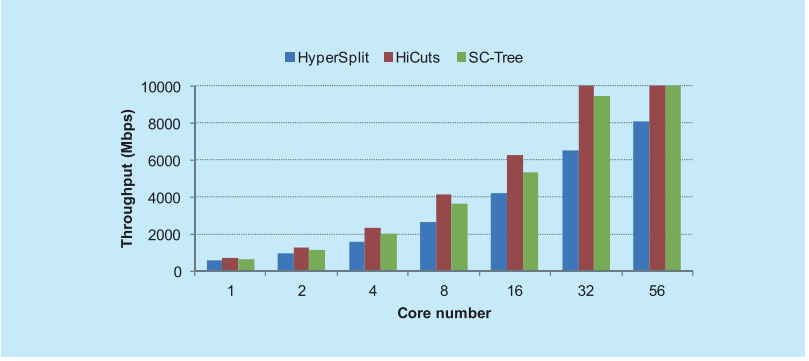

Fig. 12 *Throughput on different core number*

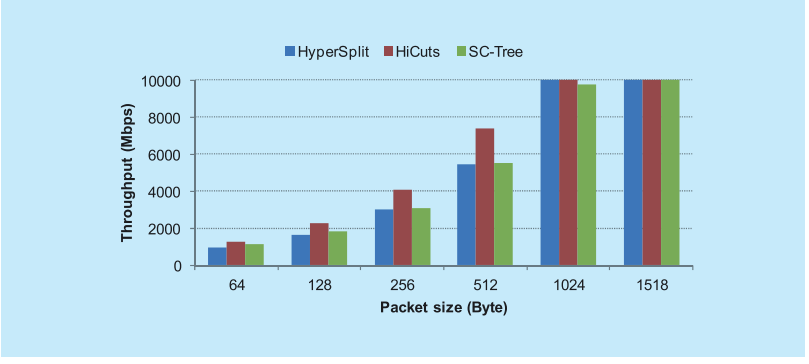

Fig. 13 *Throughput on different packet size*

5.3.2.2. Memory usage and access
The SC-Tree is compared with HiCuts and

HyperSplit [17] in terms of both memory size and memory access. HyperSplit is a typical algorithm of rule-based space decomposition with low memory requirement. Figure 10 shows the memory size on different policies. In this figure, the memory usage of SC-Tree can be reduced by up to 99% compared to the one of HiCuts-Tree. The memory size of SC-Tree is comparable with the one of HyperSplit on all rule sets. Figure 11 shows the memory access of both HyperSplit and SC-Tree. In equal-sized space decomposition algorithms, i.e., HiCuts and SC-Tree in our test, the memory access time is well guaranteed if the fixed-stride cutting strategy is adopted. Although SC-Tree needs multi-table lookups, the bitmaps and offset lists are relatively small to be stored in CPU L2 or L3 cache, which avoids directly fetching data from DRAM with much higher latency. On the contrary, the memory access time of HyperSplit increases when the policy scale or the complexity of search space grow.

Table III *Partition patterns in DFAs*

rules	total	uni bmp	uni/agg off	uni/agg off 256
snort24	8335	1595	828/795	2352/2316
snort40	19019	4494	566/289	2173/1505
linux13	4871	225	103/79	657/578
linux30	43547	2782	944/773	3648/3185
short8	5662	44	39/38	1278/1278
short120	56280	112	246/246	5473/5473
bro217	6533	73	116/1111	2235/2229

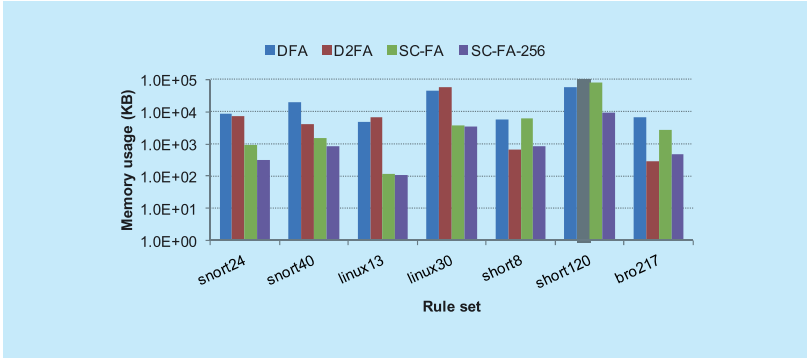

Fig. 14 *Memory usage of DFA, D²FA and SC-FA*

Fig. 15 *Throughput of DFA and SC-FA*

5.3.2.3. System throughput

Figure 12 and Figure 13 show the throughput of HyperSplit, HiCuts and SC-Tree using ACL10K policy, which are evaluated on the Tilera TILEPro64 platform. Input packets are generated by SmartBits. Both figures show the worst-case throughput of these algorithms on different core numbers and packet sizes, respectively. In Figure 12, both 256-stride HiCuts and SC-Tree can achieve 10Gbps line-rate throughput for 64-byte packets using all available tiles. However, the throughput of HyperSplit is bounded by memory bandwidth, as it requires twice memory access time of SC-Tree in worst case. Figure 13 demonstrates that both HyperSplit and 256-stride HiCuts could achieve 10Gbps line-rate throughput for packets whose length is above 1024 bytes, using two processing tiles. Moreover, SC-Tree reaches 10Gbps line-rate throughput when classifying 1518-byte packets with the same number of processing tiles, as it requires multiple bitmap calculations during each packet classification.

5.3.3 SC-FA evaluation

5.3.3.1. Partition redundancy in DFA

Similar to the measurement of partition redundancy in decision trees, the number of partition patterns in SC-FA is compared with the one of states in classic DFA to illustrate the partition redundancy in DFA based pattern matching algorithms. In Table 3, the total filed indicates the number of DFA states of different signature sets. After introducing

relay states and extracting partition patterns from all states, it is observed that the numbers of unique bitmaps and offsets are both much smaller than the one of DFA states. If the IDs of first 256 DFA states are encoded in offset arrays, the numbers of unique offsets (uni/agg off 256 in Table 3) are mostly larger than the ones of unique bitmaps (uni bmp in Table 3). Besides, unique offsets can hardly be aggregated, especially for the two string sets. The main reason of the above difference is that in packet classification, most subspaces have the same sub-rule sets but different unit-space aggregation patterns. While in pattern matching, most DFA transitions have the same transition patterns but different next states for the same symbol.

5.3.3.2. Memory usage

In Figure 14, SC-FA, D^2FA and DFA are compared in terms of memory usage. Two ID encoding schemes of SC-FA are employed for comparison. After structural compression, SC-FA can achieve 80% compression ratio and outperforms D^2FA in most cases. Specifically, the SC-FAs without ID encoding take less memory usage than DFAs, except for the two string sets from Snort. Besides, the SC-FAs with the ID encoding of first 256 DFA states can further reduce the memory requirement, except for the signature sets from L7-filter. And the D^2FA fails to process the short120 signature set due to the large number of state transition. The first exception indicates that introducing relay states on string sets imposes great state space overhead. The main reason is that the multi-string DFA usually has high-dense distinct transitions, which is quite different from the DFA constructed form regular expressions. The second exception is mainly due to the DFA states of L7-filter sets are clustered into multiple groups, and the reduction of relay states cannot benefit from state ID encoding.

5.3.3.3. System throughput

Figure 15 depicts the comparison of system throughput between DFA and SC-FA on the Tilera TILEPro64 many-core platform. All processing tiles are employed to process 1518-byte packets. The evaluation result on random packet payload reveals that the SC-FA can achieve comparable processing speed against the classic DFA. For linux13 set, SC-FA has high system throughput, i.e., 3.4Gbps processing speed, because the compressed data structure is so small that could be stored within L2 cache. Besides, the throughputs are nearly identical on the signature sets of short8 and bro217. For other evaluation sets, the throughputs are only degenerated about 10% on average. If employing the state ID encoding of SC-FA, the throughputs are still around 1Gbps. The speed gap mainly results from the increase of unique offset lists introducing more extra cache coherence operations among multiple tiles.

VI. Conclusions

The rapid growth of network applications requires network security devices to perform high-performance lookup of complex security policies. This paper focuses on the efficient implementation of both packet classification and pattern matching algorithms on many-core network processing platforms. Based on the decoupling of addressing and partition information, the structural compression approach homogenizes these two types of search data structures. It achieves above 90% compression ratio, and also guarantees the processing speed. Evaluation shows the approach reaches multi-Gbps packet inspection speed on Tilera TILEPro64 many-core platform. To encourage the innovation in this area, the source code of SC-Tree has been publicly available at [18].

Acknowledgements

The authors would like to thank the reviewers for their detailed reviews and constructive comments, which have helped improve the quality of this paper.

References

[1] GUPTA P, MCKEOWN N. Classifying Packets with Hierarchical Intelligent Cuttings [J]. IEEE Micro, 2000, 20(1): 34-41.

[2] SINGH S, BABOESCU F, VARGHESE G, WANG J. Packet Classification using Multidimensional Cutting [C]// Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03). New York, USA: ACM Press, 2003: 213-224.

[3] KUMAR S, DHARMAPURIKAR S, YU F, CROWLEY P, TURNER J. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection [C]// Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '06). New York, USA: ACM Press, 2006: 339-350.

[4] BECCHI M, CROWLEY P. An Improved Algorithm to Accelerate Regular Expression Evaluation [C]// Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '07). New York, USA: ACM Press, 2007: 145-154.

[5] Tilera Corp. TILEPro Processor Family. http://www.tilera.com/products/processors/TILEPro_Family

[6] VAMANAN B, VOSKUILEN G, VIJAYKUMAR T. EffiCuts: Optimizing Packet Classification for Memory and Throughput [C]// Proceedings of the 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '10). New York, USA: ACM Press, 2010: 207-218.

[7] LIU D, HUA B, HU X, TANG X. High-performance Packet Classification Algorithm for Many-core and Multithreaded Network Processor [C]// Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06). New York, USA: ACM Press, 2006: 334-344.

[8] QI Y, WANG K, FONG J, XUE Y, LI J, JIANG W, PRASANNA V. FEACAN: Front-End Acceleration for Content-Aware Network Processing [C]// Proceedings of the 33rd Annual IEEE International Conference on Computer Communications (INFOCOM '11). Washington DC, USA: IEEE Press, 2011: 2114-2122.

[9] MEINERS C, PATEL J, NORIGE E, TORNG E, LIU A. Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems [C]// Proceedings of the 19th USENIX conference on Security (USENIX Security'10). Berkeley, CA, USA: USENIX Association, 2010: 8-8.

[10] SMITH R, ESTAN C, JHA S, KONG S. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata [C]// Proceedings of the 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '08). New York, USA: ACM Press, 2008: 207-218.

[11] Tilera Corp. UG305 TILEmpower Appliance User's Guide, 2011.

[12] Tilera Corp. UG101 Tile Processor User Architecture Manual, 2011.

[13] ClassBench. http://www.arl.wustl.edu/classbench/

[14] Snort. http://www.snort.org/

[15] Bro. http://www.bro-ids.org/

[16] Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/

[17] QI Y, XU L, YANG B, XUE Y, LI J. Packet Classification Algorithms: From Theory to Practice [C]// Proceedings of the 31rd Annual IEEE International Conference on Computer Communications (INFOCOM '09). Washington DC, USA: IEEE Press, 2009: 648-656.

[18] SC-Tree. http://security.riit.tsinghua.edu.cn/share/sc_tree.tar.gz

Biographies

WANG Xiang, (the corresponding author, email: xiang-wang11@mails.tsinghua.edu.cn), received the B.S. degree from the School of Telecommunication Engineering, Xidian University, Xi'an, Shaanxi, China, in 2007, and the M.S. degree from the School of Software Engineering, University of Science and Technology of China, Hefei, Anhui, China, in 2010. He is working toward the Ph.D. degree at the Department of Automation, Tsinghua University, China. He is currently a research assistant in the Network Security Laboratory. His research interests include algorithmic, optimization, and performance issues in computer networking and architectures.

QI Yaxuan, received the B.S. degree, M.S. degree and Ph.D. degree from the Department of Automation, Tsinghua University, China, in 2002, 2005, 2011, respectively.

WANG Kai, received the B.S. degree from the Department of Electronic Science and Engineering, Nanjing University, Nanjing, Jiangsu, China, in 2009. He is working toward the M.S. and Ph.D. degrees at the Department of Automation, Tsinghua University, China. He is currently a research assistant in the Network Security Laboratory. His research interests include network security and algorithmic, software-defined networking.

XUE Yibo, received the B.S. and M.S. degrees in computer science from Harbin Institute of Technology, Heilongjiang, China, in 1989 and 1992, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology of Chinese Academy of Sciences, Beijing, China, in 1995. He is a member of the IEEE and ACM, a senior member of the China Computer Federation. His research interests include computer network and information security, computer architecture, and parallel computing. He has published more than 150 papers, and is inventor of more than 40 patents. He is currently a vice director at the Centre for Microprocessor and SOC Technology and a professor at the Research Institute of Information Technology, Tsinghua University.

LI Jun, received the Ph.D. degree in Computer Science from New Jersey Institute of Technology (NJIT), and MS and BS degrees in Control and Information, respectively, from Department of Automation, Tsinghua University. He is currently Dean of the Research Institute of Information Technology, Tsinghua University, Beijing, China. He is also Executive Deputy Director of the Tsinghua National Laboratory for Information Science and Technology, Beijing, China. Before rejoined Tsinghua in 2003, he held executive positions at ServGate Technologies, which he co-founded in 1999. Prior to that, he was senior software engineer at EXAR and TeraLogic. In between of his MS and Ph.D. studies, he was an assistant professor then lecturer in the Department of Automation, Tsinghua University. His research interests include networking and network security, pattern recognition and image processing.