High Throughput and Memory Efficient Multi-Match Packet Classification Based on Distributed and Pipelined Hash Tables

Yang Xu, *Member, IEEE,* Zhaobo Liu, Zhuoyuan Zhang, and H. Jonathan Chao, *Fellow, IEEE*

***Abstract*—The emergence of new network applications, such as the network intrusion detection system and packet-level accounting, requires packet classification to report all matched rules instead of only the best matched rule. Although several schemes have been proposed recently to address the multi-match packet classification problem, most of them require either huge memory or expensive Ternary Content Addressable Memory (TCAM) to store the intermediate data structure, or they suffer from steep performance degradation under certain types of classifiers. In this paper, we decompose the operation of multi-match packet classification from the complicated multi-dimensional search to several single-dimensional searches, and present an asynchronous pipeline architecture based on a signature tree structure to combine the intermediate results returned from single-dimensional searches. By spreading edges of the signature tree across multiple hash tables at different stages, the pipeline can achieve a high throughput via the inter-stage parallel access to hash tables. To exploit further intra-stage parallelism, two edge-grouping algorithms are designed to evenly divide the edges associated with each stage into multiple work-conserving hash tables. To avoid collisions involved in hash table lookup, a hybrid perfect hash table construction scheme is proposed. Extensive simulation using realistic classifiers and traffic traces shows that the proposed pipeline architecture outperforms HyperCuts and B2PC schemes in classification speed by at least one order of magnitude, while having a similar storage requirement. Particularly, with different types of classifiers of 4K rules, the proposed pipeline architecture is able to achieve a throughput between 26.8Gbps and 93.1Gbps using perfect hash tables.**

***Index Terms*—Packet Classification, Signature Tree, TCAM, Hash Table.**

I. Introduction

As the Internet continues to grow rapidly, packet classification has become a major bottleneck of high-speed routers. Most traditional network applications require packet classification to return the best or highest-priority matched rule. However, with the emergence of new network applications like Network Intrusion Detection System (NIDS), packet-level accounting [1], and load-balancing, packet classification is required to report all matched rules, not only the best matched rule. Packet classification with this capability is called multi-match packet classification [1] [2] [3] [4] [5], to distinguish it from the conventional best-match packet classification.

Yang Xu, Zhaobo Liu, Zhuoyuan Zhang, H. Jonathan Chao are with the Department of Electrical and Computer Engineering, Polytechnic Institute of New York University, New York, 11201 USA.

This work was presented in part at the fifth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2009), Princeton, New Jersey, October 2009.

Typical NIDS systems, like SNORT [6], use multi-match packet classification as a pre-processing module to filter out benign network traffic and thereby reduce the rate of suspect traffic arriving at the content matching module [7] [8], which is more complicated than packet classification, and usually can not run at the line rate in the worst case situation. As a pre-processing module, packet classification has to check every incoming packet by comparing fields of the packet header against rules defined in a classifier. To avoid slowing down the performance of the NIDS system, packet classification should run at the line rate in spite of the classifiers and traffic patterns.

Many schemes have been proposed in literature aiming at optimizing the performance of packet classification in terms of classification speed and storage cost; however, most of them focus on only the best-match packet classification [9] [10] [11]. Although some of them could also be used for multi-match packet classification, they suffer from either a huge memory requirement or steep performance degradation under certain types of classifiers [12] [13]. Ternary Content Addressable Memory (TCAM) is well-known for its parallel search capability and constant processing speed, and it is widely used in IP route lookup and best-match packet classification. Due to the limitation of its native circuit structure, TCAM can only return the first matching entry, and therefore can not be directly used in multi-match packet classification. To enable the multi-match packet classification on TCAM, some research works published recently [2] [3] [4] [5] propose to add redundant intersection rules in TCAM. However, the introduction of redundant intersection rules further increases the already high implementation cost of the TCAM system.

The objective of this paper is to design a high throughput and memory efficient multi-match packet classification scheme without using TCAMs. Given the fact that a single-dimensional search is much simpler and has already been well studied, we decompose the complex multi-match packet classification into two steps. In the first step, single-dimensional searches are performed in parallel to return matched fields in each dimension. In the second step, a well-designed pipeline architecture combines the results from single-dimensional searches to find all matched rules. Simulation results show that the proposed pipeline architecture performs very well under all tested classifiers, and is able to classify one packet within every 2-10 time slots, where one time slot is defined as the time for one memory access. Our main contributions in this paper are summarized as follows:

1) We model the multi-match packet classification as a

concatenated multi-string matching problem, which can be solved by traversing a signature tree structure.
2) We propose an asynchronous pipeline architecture to accelerate the traversal of the signature tree. By distributing edges of the signature tree into hash tables at different stages, the proposed pipeline can achieve a very high throughput.
3) We propose two edge-grouping algorithms to partition the hash table at each stage of the pipeline into multiple work-conserving hash tables, so that the intra-stage parallelism can be exploited. By taking advantage of the properties of the signature tree, the proposed edge-grouping algorithms perform well in solving the location problem, overhead minimization problem, and balancing problem involved in the process of hash table partition.
4) We propose a hybrid perfect hash table construction scheme, which can build perfect hash tables for each stage of the pipeline structure, leading to an improved performance in both classification speed and storage complexity.

The rest of the paper is organized as follows. Section II reviews the related work. Section III formally defines the multi-match packet classification problem, and presents terms to be used in the paper. Section IV introduces the concept of signature tree, based on which Section V proposes an asynchronous pipeline architecture. Section VI presents two edge-grouping algorithms which are used to exploit intra-stage parallel query. Section VII presents the hybrid perfect hashing scheme. In Section VIII, we discuss implementation issues and present the experimental results. Finally, Section IX concludes the paper.

II. Related Work

Many schemes have been proposed in literature to address the best-match packet classification problem, such as Trie-based schemes [9] [14], decision tree-based schemes [11] [12] [15], TCAM-based schemes [16]–[24], and two-stage schemes [14] [13] [19] [25].

Due to the high power consumption of TCAM, some schemes are proposed to reduce it using one of the following two ideas: (1) reducing the TCAM entries required to represent a classifier by using range encoding [19] [20] [21] or logic optimization [22] [23], or (2) selectively activating part of the TCAM blocks when performing a classification [24]. Although these schemes reduce the power consumption of TCAM, they can not be directly applied to multi-match packet classification. To enable the multi-match packet classification on TCAM, [1] proposes several schemes that allow TCAM to return all matched entries by searching the TCAM multiple times after adding a discriminator field in TCAM. Consequently, the power consumption and processing time increase linearly when the number of entries matching a packet increases. According to the results in [1] based on 112 Access Control Lists (ACLs) and the SNORT rule set, the average number of entries that one packet can potentially match is about 4 to 5. So the power consumption and processing time of multi-match packet classification can be 5 times higher than that of the best-match packet classification. Some research works published

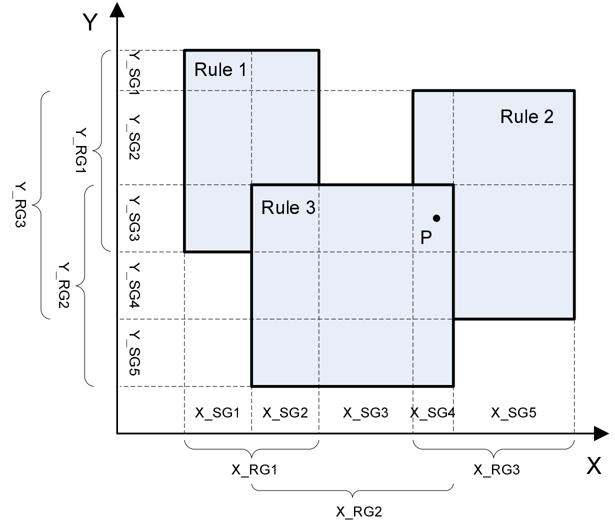


Fig. 1. Segment encoding vs. range encoding.

recently [2] [3] [4] [5] propose to add redundant intersection rules in TCAM to support multi-match packet classification. However, the introduction of redundant intersection rules further increases the already high implementation cost of the TCAM system.

In this paper, we focus on the two-stage schemes, in which the multi-dimensional search of packet classification is first decomposed into several single-dimensional searches, and then the intermediate results of single-dimensional searches are combined to get the final matched rule. To facilitate the combination operation, each field of rules in the two-stage schemes is usually encoded as either a range ID or several segment IDs. Consider the classifier shown in Fig.1, which has three 2-dimensional rules, each represented by a rectangle. Ranges are defined as the projections of the rectangles along a certain axis. For example, the projections of rules R1, R2, and R3 along axis X form three ranges denoted by X_RG1, X_RG3, and X_RG2, respectively. In contrast, segments are the intervals divided by the boundaries of projections.

With the segment encoding method, each rule is represented by multiple segment ID combinations, which may cause a serious storage explosion problem [14] [13]. Several schemes [19] [25] have been proposed to address the storage explosion problem by using TCAM and specially designed encoding schemes. However, the use of TCAM increases the power consumption and implementation cost, and more importantly, they can only be used for the best-match packet classification.

With the range encoding method, the representation of each rule requires only one range ID combination, and therefore the storage explosion problem involved in the segment encoding is avoided. The low storage requirement comes at a price of slow query speed , which prevents the range encoding method from being used in practical systems. To the best of our knowledge, the only published two-stage classification scheme using range encoding is B2PC [26], which uses multiple Bloom Filters to accelerate the validation of range ID combinations. In order to avoid the slow exhaustive validation, B2PC examines range ID combinations according to a predetermined sequence, and re-

turns only the first matched range ID combination, which may not always correspond to the highest-priority matched rule due to the inherent limitation of the B2PC scheme. Furthermore, B2PC can not support multi-match packet classification.

III. Problem Statement

A classifier *C* is a set of *N* rules, sorted in descending order of priorities. The priorities of rules are usually defined by their rule IDs, where a smaller rule ID means a higher priority. Each rule includes *d* fields, each of which represents a range of a certain dimension. From a geometric point of view, each rule represents a hyper-rectangle in the *d*-dimensional space. Since each packet header corresponds to a point *P* in the *d*-dimensional space, the problem of conventional best-match packet classification is equivalent to finding the highest-priority hyper-rectangle enclosing point *P*, while the problem of multi-match packet classification is equivalent to finding all hyper-rectangles enclosing point *P*.

In order to perform the multi-match packet classification efficiently, given a classifier, we convert it to an encoded counterpart by assigning each distinct range a unique ID on each dimension. Given the classifier in Table I, its encoded counterpart is shown in Table II, in which f_{ij} is the ID of the jth unique range appearing on the ith dimension of the classifier.

TABLE I
A classifier with seven rules

Rule	Src IP	Dest IP	Src Port	Dest Port	Prot
r_1	128.238.147.3	169.229.16.*	135	*	TCP
r_2	128.238.147.3	169.229.16.*	<1024	80	UDP
r_3	128.238.147.3	169.229.16.*	*	21	TCP
r_4	128.238.147.3	169.229.16.*	*	21	*
r_5	169.229. 4.*	128.238.147.3	<1024	<1024	TCP
r_6	128.238.147.3	169.229.4.*	110	80	TCP
r_7	169.229.4.*	*	*	21	TCP

TABLE II
The classifier after range encoding

Rule	Src IP	Dest IP	Src Port	Dest Port	Protocol
r_1	f_{11}	f_{21}	f_{31}	f_{41}	f_{51}
r_2	f_{11}	f_{21}	f_{32}	f_{42}	f_{52}
r_3	f_{11}	f_{21}	f_{33}	f_{43}	f_{51}
r_4	f_{11}	f_{21}	f_{33}	f_{43}	f_{53}
r_5	f_{12}	f_{22}	f_{32}	f_{44}	f_{51}
r_6	f_{11}	f_{23}	f_{34}	f_{42}	f_{51}
r_7	f_{12}	f_{24}	f_{33}	f_{43}	f_{51}

TABLE III
A packet to be classified

Src IP	Dest IP	Src Port	Dest Port	Protocol
128.238.147.3	169.229.16.2	135	21	TCP

Given a packet header and an encoded classifier with *d* dimensions, the multi-match packet classification scheme proposed in this paper consists of two steps. In the first step, *d* relevant fields of the packet header are each sent to a single-dimensional search engine, where either prefix-based matching or range-based matching will be performed to return all matched range IDs. Consider a packet header given in Table III: the range IDs returned from five single-dimensional search engines are shown in Table IV, and can form $1 \times 2 \times 3 \times 3 \times 2 = 36$ different range ID combinations. Since we have no idea in advance of which combinations among the 36 appear in the encoded classifier, we have to examine all 36 combinations, without exception, in the second step to return all valid combinations. Since the single-dimensional search problem has been well addressed in literature [27], in this paper we focus on only the second step. In the remainder of the paper, packet classification will specifically refer to this second step unless special notation is given.

TABLE IV
Range IDs returned by single-dimensional searches

Src IP	Dest IP	Src Port	Dest Port	Protocol
f_{11}	f_{21}	f_{31}	f_{41}	f_{51}
	f_{24}	f_{32}	f_{43}	f_{53}
		f_{33}	f_{44}	

If we view each range ID as a character, the multi-match packet classification problem could be modeled as a concatenated multi-string matching problem. In this problem, the encoded classifier could be regarded as a set of strings with *d* characters. From the encoded classifier, we can get *d* *universal character sets*, each of which includes characters in one column of the encoded classifier. The set of range IDs returned by each single-dimensional search engine is called a *matching character set*, which is a subset of the corresponding universal character set. **The concatenated multi-string matching problem is to identify all strings in the encoded classifier which could be constructed by concatenating one character from each of *d* matching character sets.** The main challenge of the concatenated multi-string matching problem is to examine a large number of concatenated strings at an extremely high speed to meet the requirement of high-speed routers.

IV. Signature Tree

To facilitate the operation of concatenated multi-string matching, we present a data structure named signature tree to store strings in the encoded classifier. Fig.2 shows a signature tree corresponding to the encoded classifier in Table II. Each edge of the tree represents a character, and each node represents a prefix of strings in the encoded classifier. The ID of each leaf node represents the ID of a rule in the encoded classifier.

The concatenated multi-string matching can be performed by traversing the signature tree according to the inputs of *d* matching character sets. If any of the *d* matching character sets is empty, the result will be NULL. Otherwise, the matching is performed as follows. At the beginning, only the root node is active. The outgoing edges of the root node are examined against the characters in the first matching character set. Each time when a match is found, the corresponding node (at level one) pointed by the matched edge will be activated. After the

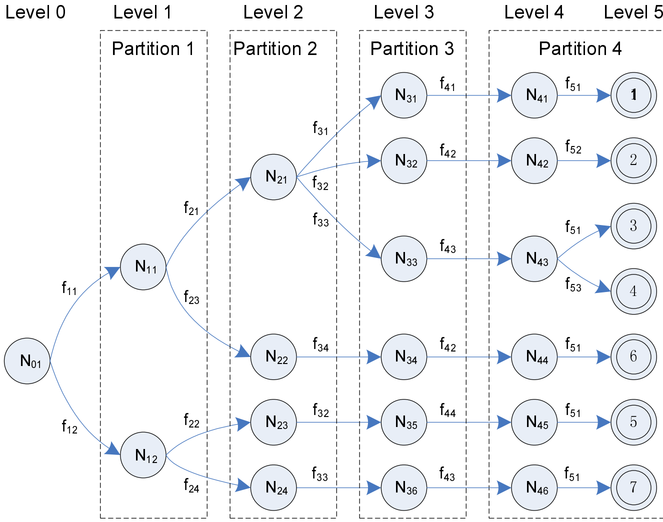


Fig. 2. An example of signature tree.

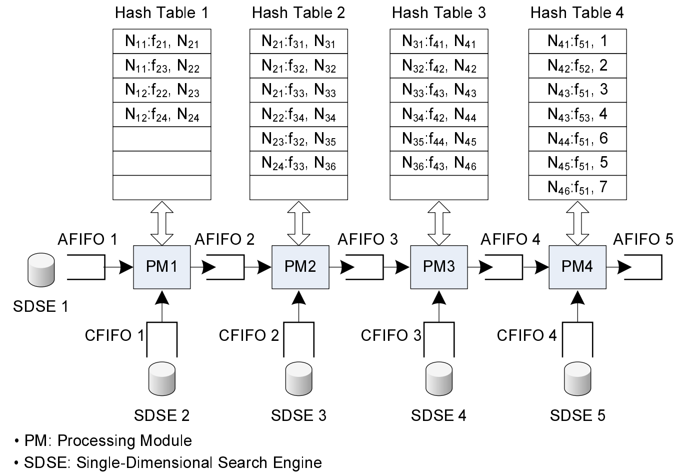


Fig. 3. Pipelining architecture for packet classification.

examination, the root node is deactivated, and one or multiple nodes (if at least one matching edge is found) at level one become active. Then the active nodes at level one will be examined one by one against the characters in the second matching character set. A similar procedure will repeat to examine characters in the remaining matching character sets. In this procedure, active nodes move from low levels to high levels of the signature tree, and eventually IDs of the active leaf nodes represent the matched rules.

The traversal complexity of the signature tree depends on many factors, including the size of each matching character set, the number of active nodes at each level when the signature tree is being traversed, as well as the implementation method of the signature tree. One way of implementing the signature tree is to store each node as a whole data structure, and connect parent and child nodes together by points in the parent nodes. However, our analysis on the real classifiers shows that the numbers of outgoing edges of nodes have a very large deviation (due to the inherent non-uniform distribution of field values in classifiers), which makes the design of a compact node structure a very challenging task. Even if we came up with a compact node structure using the pointer compressing scheme [28], incremental updates and query operations on the signature tree would become extremely difficult. Therefore in this paper, rather than storing each node as a whole structure, we break up the node and store edges directly in a hash table. More specifically, each edge on the signature tree takes one entry of the hash table in the form of <source node ID:character, destined node ID>. Here, "source node ID : character" means the concatenation of "source node ID" and "character" in binary mode, and works as the key of the hash function, while "destined node ID" is the result we hope to get from the hash table access.

Given the fact that the memory access speed is much slower than logic, the processing speed of a networking algorithm is usually determined by the number of memory accesses (references) required for processing a packet [29]. The memory accesses that occur during the signature tree traversal are caused by the hash table lookup. Therefore given a certain hash table implementation, the number of times that the hash table must be accessed to classify a packet is a good indicator for the processing speed of the signature tree based packet classification. In the following sections, we divide the hash table into multiple partitions to exploit parallel hash table access and thus improve the performance. Here, we introduce two properties about the universal character set and the signature tree, which will be used later.

Property 1. *Characters in each universal character set can be encoded as any bit strings as long as there are no two characters with the same encoding.*

Property 2. *Nodes on the signature tree can be given any IDs, as long as there are no two nodes with the same IDs at the same level.*

V. Asynchronous Pipeline Architecture

To improve the traversal speed of the signature tree, we separate the signature tree into d-1 partitions, and store edges of each partition into an individual hash table. More specifically, the outgoing edges of level-i nodes (i=1,...,d-1) are stored in hash table i. An example of the signature tree after partition is shown in Fig.2, which has four partitions. The corresponding hash tables of these four partitions are shown in Fig.3. It's worth noting that outgoing edges of the root node are not stored. This is because the root node is the only node at level 0, and each of its outgoing edges corresponds to exactly one character in the first universal character set. According to property 1, we can encode each character of the first universal character set using the ID of the corresponding destined level-1 node. For instance, in Fig.2 we can let f_{11}=N_{11} and f_{12}=N_{12}. So given the first matching character set, we can immediately get the IDs of active nodes at level 1.

For a d-dimensional packet classification application, we propose an asynchronous pipeline architecture with d-1 stages. Fig.3 gives an example of the proposed pipeline architecture with d=5. It includes d-1 processing modules (PM). Each PM is attached with an input Character FIFO (CFIFO), an input Active node FIFO (AFIFO), an output AFIFO, and a hash

S	E	Character/Node ID

Fig. 4. The format of entries in CFIFO/AFIFO.

table. Each CFIFO supplies the connected PM with a set of matching characters returned by the single-dimensional search engine. Each AFIFO delivers active node IDs between adjacent PMs. Each hash table stores edges in a certain partition of the signature tree.

Since each packet may have multiple matching characters/active nodes at a stage of the pipeline, two bits in each entry of CFIFO/AFIFO are used to indicate the ownership of matching characters/active nodes, as shown in Fig.4. An "S" bit set to 1 means that the entry is the first matching character/active node of a packet, while an "E" bit set to 1 means that the entry is the last matching character/active node of a packet. If both "S" and "E" bits are set to 1, it means that the entry is the only matching character/active node of a packet.

When a packet is going to be classified, the d relevant fields of the packet header are first sent to d single-dimensional search engines. Each search engine returns a set of matching characters representing the matched ranges on the corresponding dimension to the attached CFIFO (the first search engine returns matching characters to the attached AFIFO 1). If no matching character is found, a NULL character encoded as all "0" is returned.

In the pipeline, all PMs work in exactly the same way, therefore we focus on a certain PM i (i=1,...,d-1) and consider the procedure that a packet P is being processed at PM i.

Suppose that packet P has x active node IDs in AFIFO i, which are denoted by n_1, n_2, $\ldots$, n_x, and y matching characters in CFIFO i, which are denoted by c_1, c_2, $\ldots$, c_y. The processing of packet P at PM i consists of the processing of x active node IDs. In the processing of each active node ID, say n_j, PM i takes out the matching character c_1, c_2, $\ldots$, c_y from the attached CFIFO, and concatenates each of them (if the character is not NULL) to n_j to form y hash keys and access the attached hash table for y times. Results from the hash table indicate the IDs of n_j's child nodes, which will be pushed into the output AFIFO when it is not full. If the output AFIFO is currently full, the push-in operation along with the operation of PM i will be suspended until one slot of the output AFIFO becomes available.

During the processing of packet P, if PM i can not find any match in the hash table, or if the matching character of the packet is NULL, PM i will push a "NULL" node ID encoded as all "0" into the output AFIFO to notify the downstream PMs that the packet does not match any rule.

The number of hash table accesses required by PM i to process packet P is equal to the product of the active node number and the matching character number of packet P; i.e., $x \cdot y$ in this case, if we omit the overhead caused by the hash collision.

VI. Intra-Stage Parallel Query

The asynchronous pipeline architecture introduced above deploys one hash table at each stage. The processing of each active node ID at each PM may involve multiple accessing of the hash table. To accelerate the processing of each active node ID, we plan to further partition the hash table at each stage to exploit intra-stage parallelism. After the intra-stage partition, each PM might be associated with multiple hash tables, which can be accessed in parallel. For easy control and avoidance of the packet out-of-sequence, each PM will process active node IDs in the strict serial way. That is, if there is an active node ID currently being processed (some hash tables are therefore occupied), the processing of the next active node ID cannot be started, even if there are hash tables available to use.

Before introducing schemes for the intra-stage hash table partition, we present several concepts, among which the concept of independent range set is similar to but not exactly the same as the concept of independent rule set proposed by Sun et al. in [30].

Definition 1. *Independent ranges*. Let $f_1$and $f_2 (f_1 \neq f_2)$be two ranges on a dimension. $f_1$is called independent of f_2 if $f_1 \bigcap f_2 = \phi$.

Definition 2. *Independent range set*. Let T be a set of ranges. T is called an independent range set if any two ranges in T are independent.

Definition 3. *Independent characters*. Let c_1 and c_2 be two characters associated with range f_1 and f_2, respectively. c_1 is called independent of c_2 if f_1 is independent of f_2.

Definition 4. *Independent character set*. Let U be a set of characters. U is called an independent character set if any two characters in U are independent.

Definition 5. *Independent edges*. Suppose $e_1=<s_1:c_1, d_1>$ and $e_2=<s_2:c_2, d_2>$ are two edges in a certain partition of the signature tree. $e_1$is called dependent on e_2 if $s_1=s_2$ and c_1 is dependent on c_2; otherwise, $e_1$is called independent of e_2.

Definition 6. *Independent edge set*. Let E be a set of edges in a certain partition of the signature tree. $E$is called an independent edge set if any two edges in E are independent.

Definition 7. *Work-conserving hash tables*. Suppose we have M hash tables associated with PM i of the pipeline, where an active node ID, say nid_1, is being processed. We say these hash tables are work-conserving for processing nid_1 if no hash table is left idle when there are matching characters associated with nid_1 waiting for query; in other words, we can always find a free hash table in which an un-queried edge [1] of nid_1 is stored if not all hash tables are occupied. Hash tables associated with PM i are called work-conserving hash tables if they are work-conserving for processing any active node IDs.

A. Edge Grouping

The main objective of the intra-stage hash table partition is to guarantee the work-conserving property of the partitioned hash tables, so that the processing throughput of PM can

[1] An un-queried edge of an active node ID could be either a real edge or an unreal edge on the signature tree. The query for an unreal edge will cause a return of search failure.

be maximized and made more predictable. Given M work-conserving hash tables and y matching characters, the processing of each active node ID can be finished within $\lceil y/M \rceil$ parallel hash accesses.

Suppose we want to partition the original hash table associated with PM i into M work-conserving hash tables. The most straightforward way is to divide edges of the original hash table into M independent edge sets, and then store each of them in an individual hash table. This way, we can guarantee the work-conserving property of the partitioned hash tables, because edges to be queried for an active node ID must be dependent on each other, and stored in different hash tables.

However, since M is a user-specified parameter, M hash tables may not be sufficient to avoid the dependency among all edges. Therefore, instead of dividing edges of the original hash table into M independent sets, we divide them into M+1 sets denoted by $G_k(k = 1, .., M+1)$, among which the first M sets are all independent edge sets, and the last set is a residual edge set, which stores edges that do not fit into the first M sets. The above action is called *edge-grouping*. We call edges in the independent edge sets regular edges, and edges in the residual edge set residual edges.

Given the M+1 edge sets after the edge-grouping, we can store edges of each independent edge set in an individual hash table, while duplicate edges of the residual edge set into all M hash tables. When an active node is being processed, we first query its regular edges, and then its residual edges. It's easily seen that no hash table would be left idle if there were an un-queried edge. Therefore, the work-conserving property of the partitioned hash tables is guaranteed.

Actually, the problem of edge-grouping itself is not difficult. The main challenge comes from the following three questions.

1) Given an edge (real or unreal), how can we locate the partitioned hash table in which the edge is stored?
2) How can we minimize the overhead caused by the redundancy of residual edges?
3) How can we balance the sizes of partitioned hash tables?

We name these three problems, respectively, as the location problem, the overhead minimization problem, and the balance problem, and present two edge-grouping schemes to deal with them.

B. Character-Based Edge-Grouping

The first edge-grouping scheme is named *character-based edge-grouping* (CB_EG). Its basic idea is to divide edges according to their associated characters, and then embed the grouping information in the encodings of characters. More specifically, we reserve the first $\lceil \log_2(M+1) \rceil$ bit of each character for the *locating prefix*, whose value is between 0 and M. If the locating prefix of a character is 0, edges labeled with the character are residual edges, and thus can be found in any partitioned hash tables. Otherwise, edges labeled with the character are regular edges and can only be found in the partitioned hash table indexed by the locating prefix. The location problem of edge-grouping is thereby solved.

To address the overhead minimization problem and the balance problem, we model the CB_EG scheme as a *weighted character grouping* (WCG) problem.

TABLE V
THE WEIGHTED CHARACTER GROUPING PROBLEM

Subject to.

$$U_k \subseteq U \ \ (k = 1, ..., M+1); \quad (1)$$

$$\bigcup_k U_k = U \quad (2)$$

$$U_{k1} \bigcap U_{k2} = \phi \ \ (k1,\ k2 = 1, ..., M+1 \ \&\ k1 \neq k2) \quad (3)$$

$$L(c_1, c_2) := \begin{cases} 1 & c_1 \text{ is dependent on } c_2 \ (c_1, c_2 \in U) \\ 0 & c_1 \text{ is independent of } c_2 \ (c_1, c_2 \in U) \end{cases} \quad (4)$$

$$L(c_1, c_2) = 0 \ \ (\forall c_1, c_2 \in U_k\ ; k = 1, ..., M) \quad (5)$$

$$W(U_k) := \sum_{c \in U_k} W(c) \quad (6)$$

Objective.

$$\text{Minimize: } \underset{k=1,...,M}{Max}(W(U_k) + W(U_{M+1})) \quad (7)$$

Let U be the universal character set associated with PM i. Let U_k (k=1, ..., M+1) be M+1 non-overlapping character sets divided from U.

Let c be an arbitrary character in U, and $W(c)$ be its weight function, meaning the number of edges labeled with c in the original hash table.

Let $W(U_k)$ (k=1, ..., M+1) be the weight of character set U_k.

Let $L()$ be the dependence indicator. $\forall c_1$, $c_2 \in U$, if c_1 is dependent on c_2, $L(c_1, c_2)$:=1; otherwise, $L(c_1, c_2)$:=0.

The WCG problem is formally described in Table V. The WCG problem is to find a valid configuration of U_k (k=1, ..., M+1) to achieve the given objective. We have proved that the WCG problem is an NP-hard problem, the proof of which is provided in the appendix. Thus, we propose the greedy algorithm in Algorithm 1 to solve the WCG problem.

According to M+1 character sets returned by the greedy WCG algorithm, we assign each character a locating prefix, and divide edges of the original hash table into M+1 edge sets. The principle is that for each character in U_k (k=1, ..., M), we let k be its locating prefix and allocate its associated edges to edge set G_k; for each character in U_{M+1}, we let 0 be its locating prefix, and allocate its associated edges to edge set G_{M+1}. After that, we can get M partitioned hash tables by allocating edges of G_k (k=1, ..., M) to hash table k, and duplicating edges of G_{M+1} in every hash table.

Let's consider an example, which partitions the last hash table of Fig.3 into two work-conserving hash tables with the CB_EG scheme. First of all, we get the universal character set associated with PM 4. It has three characters: f_{51}, f_{52}, and f_{53}, whose weights are 5, 1, and 1, respectively. With the greedy WCG algorithm, the three characters are divided into two independent character sets (U_1 and U_2) and one residual character set (U_3), among which U_1 contains f_{51}, U_2 contains f_{52}, and U_3 contains f_{53}. Therefore, edges labeled

Algorithm 1 Greedy algorithm for WCG problem

1: **Input:**
2: U : the universal character set;
3: M : the number of independent character sets;
4: $W(c)$: the weight of character c;
5: **Output:**
6: independent character sets U_1, …, U_M;
7: residual character set U_{M+1};
8:
9: $U_k := \phi$ $(k = 1, ..., M + 1)$;
10: $W(U_k) := 0$ $(k = 1, ..., M)$; //initialize the weight of set U_k
11: Sort U in decreasing order of the character weight.
12: If U is empty, return (U_k $(k = 1, \ldots, M + 1)$);
13: From U select the character c with the largest weight;
14: Select the set $U^{'}$ with the smallest weight among sets U_1, …, U_M whose characters are all independent of c. If there is more than one such set, select the first one. If no such set is found, put c into set U_{M+1}, remove c from set U, and go to step 12;
15: Put c into set $U^{'}$; remove c from set U; $W(U^{'}) + = W(c)$; Go to step 12.

with character f_{51} and f_{52} are allocated to the first partitioned hash table and the second partitioned hash table respectively, while edges labeled with character f_{53} are duplicated in both partitioned hash tables. The final partition result is shown in Fig.5(a).

We use *PE* to denote the partition efficiency of a hash table partition and define it in (8).

$$PE = \frac{\text{\# of edges in the original hash table}}{M \times \text{\# of edges in the largest partitioned hash table}} \tag{8}$$

In the example above, the partition efficiency is only 58.3% for two reasons: first is the redundancy caused by the residual edge <N_{43}:f_{53}, 4>; second reason is the extreme imbalance between two partitioned hash tables. This imbalance is caused by the inherent property of the CB_EG scheme, which has to allocate edges labeled with the same character into the same hash table. In the last hash table of Fig.3, five out of seven edges are labeled with the same character f_{51}. According to the CB_EG scheme, these five edges have to be allocated to the same hash table, which results in the imbalance between partitioned hash tables.

As a matter of fact, in real classifiers, the second reason degrades the partition efficiency more severely than the first. This is because many fields of real classifiers have very imbalanced distributions of field values. For instance, the transport-layer protocol field of real classifiers is restricted to a small set of field values, such as TCP, UDP, ICMP, etc. Most entries, say 80%, of real classifiers, are associated with the TCP protocol. With the CB_EG scheme, edges labeled with TCP have to be allocated to the same hash table, which may cause an extreme imbalance of hash tables, and thus result in low partition efficiency.

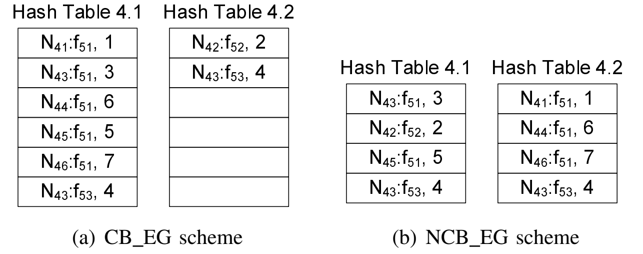


Fig. 5. Two partitioned hash tables from the last hash table in Fig.3.

C. Node-Character-Based Edge-Grouping

The second edge-grouping scheme is named Node-Character-Based Edge-Grouping (NCB_EG), which divides edges not only based on their labeled characters, but also based on the IDs of their source nodes.

According to property 2, the ID of each node on the signature tree can be assigned to any values as long as there are no two nodes at the same level assigned the same ID. With this property, the NCB_EG scheme stores the grouping information of each edge in both the encoding of the edge's associated character and the ID of the edge's source node. More specifically, the NCB_EG scheme reserves the first $\lceil \log_2(M+1) \rceil$ bits of each character for the *locating prefix*, and the first $\lceil \log_2 M \rceil$ bits of each node ID for the *shifting prefix*. Given an arbitrary edge <s_1:c_1, d_1>, suppose the locating prefix of c_1 is *loc*, and the shifting prefix of s_1 is *sft*. If *loc* equals 0, the edge is a residual edge, and can be found in any partitioned hash tables. Otherwise, the edge is a regular edge and can only be found in the partitioned hash table indexed by (*sft*+*loc*-1) mod *M*+1.

In order to locate the partitioned hash table in which a given edge is stored using the above principle, we have to divide edges into different edge sets following the same principle. In contrast to CB_EG, the NCB_EG scheme solves the overhead minimization problem and the balance problem in two different steps: the Locating Prefix Assignment (LPA) and the Shift Prefix Assignment (SPA).

The overhead minimization problem is solved in the LPA step, in which the universal character set associated with PM *i* is divided into *M* independent character sets and one residual character set. Each character is assigned a locating prefix ranging from 0 to *M* according to the character set it is allocated to. The LPA step can also be described by the WCG problem given in Table V, with only the objective changed from (7) to (9).

$$\text{Minimize: } W(U_{M+1}) \tag{9}$$

We can not find a polynomial time optimal algorithm to solve the WCG problem with the objective in (9); therefore, we use the greedy WCG algorithm given in Algorithm 1 to solve it.

The purpose of the SPA step is to balance the sizes of independent edge sets. This is achieved by assigning a shift prefix to each node to adjust the edge sets to which the outgoing edges of the node are allocated. A heuristic algorithm for the shift prefix assignment is given in Algorithm 2.

Algorithm 2 Shift Prefix Assignment Algorithm

1: **Input:**
2: M: the number of independent character sets;
3: independent character set U_1, …, U_M;
4: residual character set U_{M+1};
5: S: the set of nodes at level i of the signature tree;
6: E: the set of outgoing edges of nodes in S;
7: **Output:**
8: shift prefixes of nodes in S;
9: independent edge sets G_1, …, G_M;
10: residual edge set G_{M+1};
11:
12: $G_k := \phi$ $(k = 1, ..., M + 1)$;
13: Sort nodes of S in decreasing order of the number of outgoing edges;
14: **for** each node n in S **do**
15: Divide the outgoing edges of n into $M + 1$ sets. The principle is that for characters in U_k $(k = 1, \ldots, M + 1)$, put their associated outgoing edges to Z_k;
16: Select the largest edge set Z_t among Z_k $(k = 1, \ldots, M)$; if there are multiple largest edge sets, select the first one;
17: Select the smallest edge set G_v among G_k $(k = 1, \ldots, M)$; if there are multiple smallest edge sets, select the first one;
18: Let $p := (v - t) \bmod M$, and set the shift prefix of n as p; //align Z_t to G_v to achieve balance among G_k $(k = 1, \ldots, M)$
19: **for** each set Z_k $(k = 1, \ldots, M)$ **do**
20: Move edges from set Z_k to set $G_{(k+p-1) \bmod M+1}$;
21: **end for**
22: Move edges from set Z_{M+1} to set G_{M+1};
23: **end for**

Consider using the NCB_EG scheme to partition the last hash table of Fig.3 into two work-conserving hash tables. The LPA step of NCB_EG is same as that for CB_EG. With the greedy WCG algorithm, we can get two independent character sets (U_1 and U_2) and one residual character set (U_3); among which U_1 contains f_{51}, U_2 contains f_{52}, and U_3 contains f_{53}. Therefore the locating prefixes of f_{51}, f_{52}, and f_{53} are 1, 2, and 0, respectively. Then the SPA algorithm is used to assign each level-4 node of the signature tree a shift prefix. Since Node N_{43} has two outgoing edges, while other nodes have only one, it will first be assigned a shift prefix. Since all independent edge sets (G_1 and G_2) are empty at the beginning, we assign a shift prefix of 0 to N_{43}. Based on the shift prefix on the N_{43}, and the locating prefix on characters, regular edge <N_{43}:f_{51}, 3> is allocated to G_1, and residual edge <N_{43}:f_{53}, 4> is allocated to G_3. N_{41} is the second node to be assigned a shift prefix. In order to balance the sizes of G_1 and G_2, the shift prefix of N_{41} is set to 1, so that the edge <N_{41}:f_{51}, 1> is allocated to G_2 according to the locating prefix of f_{51}. Similarly, N_{42}, N_{44}, N_{45}, and N_{46} will be each assigned a shift prefix, and their outgoing edges are allocated to the corresponding edge sets. The final partitioned hash tables, after the edge-grouping, are shown in Fig.5(b), where the residual edge <N_{43}:f_{53}, 4> is duplicated in both hash tables.

In this example, the hash table partition efficiency is 7/8=87.5%, which is higher than that with the CB_EG scheme. The main reason for this improved partition efficiency is that the NCB_EG scheme is capable of spreading edges labeled with character f_{51} into different hash tables, so that a better balance among partitioned hash tables is achieved.

VII. Perfect Hash Table Construction

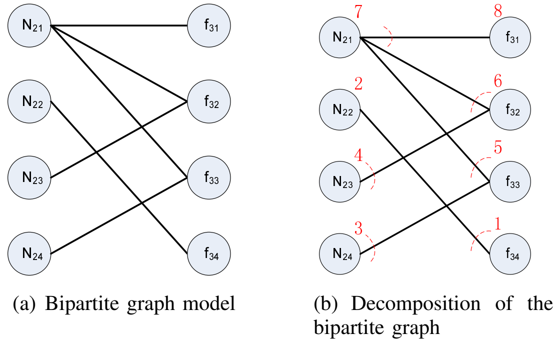
One major problem involved in the hash table design is hash collision, which may increase the number of memory accesses involved in each hash table lookup and slow down the packet classification speed. Therefore, a perfect hash table with the guarantee of no hash collision is desirable to support high-speed packet classification. Although there are many perfect hashing and alternative algorithms available in literature, most of them require a complicated procedure to generate the hash index (e.g., traversing a tree-like structure) [31], or need more than one memory access in the worst case to find the correct location storing the hash key among multiple potential locations [32], [33].

In this section, we present a hybrid perfect hash table construction scheme, which uses an arbitrary uniform hash function and guarantees that the searched hash key can be located with one memory access. For a given level of the signature tree, the proposed perfect hash table construction scheme can be used to create a single hash table (as in Fig.3) or multiple partitioned hash tables (as in Fig.5). For simplicity, we consider the case that each level of the signature tree has one single hash table. The main idea behind the scheme is that if a hash collision occurs when placement of a new key into the hash table is attempted, the collision might be avoided if the value of the key can be changed.

The hash key in the hash tables in Fig.3 is the concatenation of "source node ID" and "character". According to properties 1 and 2 summarized in Section IV, we can rename either the "source node" or the "character" to change the value of a hash key and achieve a collision-free placement. When a source node at a stage is renamed to avoid collision, we also need to update the destination node ID of its pointing edge at the previous stage. Since the change of destination node ID field won't affect the hash entry location, this process only requires minor modification in the hash table at the previous stage.

The main challenge involved in this process is that when a node or character is renamed, some existing hash keys in the hash table might have to be re-placed if they are associated with the renamed node or character. We should avoid fluctuation of the hash table; i.e., hash keys constantly being added/removed from the hash table.

The hybrid perfect hash table construction consists of three steps: in the first step, we convert the edges in the partition into an equivalent bipartite graph model; in the second step, we decompose the bipartite graph into groups of edges, and associate each group with either a node vertex or a character vertex; in the third step, we place these edges into the perfect hash table. Edges in the same group will be placed into the hash table as a whole.



(a) Bipartite graph model

(b) Decomposition of the bipartite graph

Fig. 6. Bipartite graph model associated with partition 2 of Fig.2. In Fig(b), the red-colored number close to each vertex indicates the sequence in which the corresponding vertex is carved from the graph.

1) Bipartite Graph Decomposition

Given a certain partition of the signature tree (for example, partition 2 in Fig.2), we build a bipartite graph (as shown in Fig.6(a)), in which each vertex on the left side represents a node in the partition (called node vertex), each vertex on the right side represents a character (called character vertex) in the partition, and each edge represents an edge in the partition.

2) Bipartite Graph Decomposition

On the bipartite graph, we do a least-connected-vertex-first traversal. In each iteration, we select the least connected vertex (randomly select one if there are many), and carve the vertex as well as its connected edges out from the graph. The edges that are carved together will be regarded as being in the same group, and they will be associated with the vertex that is carved together with them. This recursive process repeats itself until the bipartite graph becomes empty.

Given the bipartite graph in Fig.6(a), the process of decomposition is shown in Fig.6(b). The first vertex that is carved from the bipartite graph is character vertex "f_{34}", since it is one of the vertices with the least connected edges. The edge connected to character vertex "f_{34}" is also carved out from the bipartite graph and associated with vertex "f_{34}". After the first carving, node vertex "N_{22}" becomes the vertex with the least connected edges and will be carved from the bipartite graph. The edge group associated with "N_{22}" is empty since "N_{22}" has no edge connected at the moment when it is carved. The above process continues, and after the entire decomposition, each vertex will be associated with a group of edges (some groups can be empty as those associated with "N_{22}" and "f_{31}").

3) Perfect Hash Table Construction

Then we start to add carved edges into the hash table in reverse order of the decomposition. In other words, edges removed from the bipartite graph first are the last placed into the hash table. The edges that were carved out from the bipartite graph together will be placed into the hash table as a group. Any collision occurring during the placement of a group will cause the renaming of the associated dimension (either the source node ID or character) and the re-placement of the entire group. The reason that we place the groups of edges in reverse order of the decomposition is because the edge groups removed early from the bipartite graph are all very small (most of them have only one edge). So placing them into the hash table at the end can increase the success probability when the hash table becomes full.

During the placement of a group of edges, the name of their associated vertex will also be decided. Once a group of edges is successfully stored into the hash table, the name of its associated vertex is settled and will not be changed again.

It should be noted that the first group placed into the hash table is always empty. Consider the decomposition result of Fig.6(b). The group associated with "f_{31}" will be the first placed into the hash table, which is empty. Therefore we can label "f_{31}" as any value. Then the second group is the one associated with node "N_{21}", which has only one edge $< N_{21} : f_{31} >$. Since the edge is the first edge placed into the hash table, there is no collision and "N_{21}" can be assigned any value. The third group to be placed is the group associated with "f_{32}", which has only one edge $< N_{21} : f_{32} >$. If there is a collision when $< N_{21} : f_{32} >$ is placed, we will rename the vertex associated with it, which is "f_{32}" in this case, to resolve the collision. Note that the other dimension of the edge ("N_{21}") has its name already settled prior to this placement and will not be renamed to resolve the collision. The above process repeats itself until all edge groups are placed into the hash table without collision.

It is easy to see that with the hybrid perfect hash table construction solution, all edges are broken into many small independent groups. A collision occurring during the placement of a group will not affect the groups that had already been placed into the hash table, so the process will converge, and a perfect hash table can be constructed. It is worth mentioning that the hash function used in the hybrid perfect hash table construction doesn't have to be invertible. An arbitrary universal hash function can be used here.

VIII. Implementation Issues And Performance Evaluation

A. Scalability and Incremental Update

The proposed pipeline architecture supports an arbitrary number of dimensions. To add/delete a dimension, we only need to add/remove a PM along with its associated single-dimensional search engine, CFIFO, AFIFO, and hash tables.

The pipeline architecture also supports incremental updates of rules. To add/remove a rule, we traverse the signature tree along the path representing the rule, and add/remove the corresponding edges in the hash tables. Considering the overhead involved in perfect hash table update, we would suggest to use conventional hashing schemes to build the hash tables if the classifier is expected to be updated. Since the complexity of insertion/removal operation in hash table is $O(1)$, the pipeline architecture has a very low complexity for incremental update.

B. Storage Complexity

Suppose the maximum number of rules supported by the proposed pipeline architecture is N, the maximum number of hash tables used at each stage is M, and the number of total

dimensions is d. The storage requirement of the pipeline architecture mainly comes from two parts: (1) d single-dimensional search engines; (2) hash tables and AFIFOs/CFIFOs in the pipeline.

The storage requirement of each single-dimensional search engine depends greatly on the type of field associated with the dimension. For instance, if the type of field is transport-layer protocol, a 256-entry table could be used as the search engine, which requires no more than 1 Kbytes of memory. If the type of field is source/destination IP address, an IP lookup engine must be the single-dimensional search engine, which might require 70~100 Kbytes of memory [27].

For hash tables in the pipeline, we use two different hash schemes. The first is the conventional hashing scheme, which uses a simple linear probing scheme to resolve the hash collision [34]. In order to achieve a low hash collision rate, the conventional hash scheme uses a relatively low load factor $LF = 0.5$. The second is the hybrid perfect hashing scheme presented in Section VII, which can easily build a perfect hash table with a high load factor $LF = 0.8$.

The storage requirement for hash tables at each stage (H) is determined by the number of edges associated with that stage (T), the number of bits used to represent each edge (B), the load factor of hash tables, and the partition efficiency (PE) when multiple hash tables are used. H can be represented by (10):

$$H = T \times B \times \frac{1}{PE} \times \frac{1}{LF} \text{ (bits)} \tag{10}$$

Since each edge e_1 is represented by $< s_1 : c_1, d_1 >$; where s_1 is the ID of the source node of e_1, c_1 is the character labeled on e_1, and d_1 is the ID of the destination node of e_1; the number of bits required to represent each edge is equal to the sum of the numbers of bits used for representing s_1, c_1, and d_1. It is easily seen that the number of nodes at each level of the signature tree is no more than N; therefore s_1 and d_1 can each be represented by $\lceil \log_2 M \rceil + \lceil \log_2 N \rceil$ bits, where the first $\lceil \log_2 M \rceil$ bits are the shift prefix, and the last $\lceil \log_2 N \rceil$ bits are used to uniquely identify the node at each level of the signature tree. The number of characters in the universal character set of each dimension is equal to the number of unique ranges on that dimension. It is easy to see that the unique range on each dimension is no more than N. Therefore, c_1 could be encoded as $\lceil \log_2 (M+1) \rceil + \lceil \log_2 N \rceil$ bits, where the first $\lceil \log_2 (M+1) \rceil$ bits are the locating prefix, and the last $\lceil \log_2 N \rceil$ bits are used to uniquely identify the character (range) on the dimension. To sum up, the number of bits used for representing each edge could be obtained in (11):

$$B \leq 3 \lceil \log_2 N \rceil + 2 \lceil \log_2 M \rceil + \lceil \log_2 (M+1) \rceil \tag{11}$$

The number of edges to be stored at each stage of the pipeline is bounded by N; therefore $T \leq N$. If we assume the hash table partition efficiency is 1 (Shortly, we will show that the partition efficiency of NCB_EG scheme is close to 1), and substitute it along with $LF = 0.5$, $T \leq N$ and (11) into (12), we can get the total storage requirement of the hash tables at each stage as in (12):

$$H \leq 6N \lceil \log_2 N \rceil + 6N \lceil \log_2 M \rceil \text{ (bits)} \leq N \log_2 N \text{ (bytes)} \tag{12}$$

Since there are $d-1$ stages in the pipeline, the total memory requirement is bounded by $N(d-1)\log_2 N$ bytes. The actual memory requirement is much less that this number since the number of edges in the first few stages is much less than N. In addition, by using the perfect hash table with a high load factor $LF = 0.8$, the memory requirement can be further reduced.

Regarding AFIFOs and CFIFOs, we will later show that each AFIFO/CFIFO needs only a small piece of memory, say 8 entries, to achieve a good enough performance. If $d = 5$, the total storage requirement for 5 AFIFOs and 4 CFIFOs is less than 200 bytes, which is insignificant compared to the storage requirement of hash tables.

As a whole, the total storage requirement of the pipeline architecture, excluding the single dimensional search engines, is bounded by $N(d-1)\log_2 N$ bytes. If we substitute $N = 4K$, $d = 5$ in it, the storage requirement is bounded by 192 Kbytes, which makes our pipeline architecture among the most compact packet classification schemes proposed in literature, even if we count in the memory required by the single dimensional search engines. The actual memory requirement of the proposed pipeline will be presented in the next subsection using classifiers created by ClassBench [35].

C. *Performance Evaluation*

Three packet classification schemes are used in the evaluation: the proposed pipeline architecture, B2PC [26], and HyperCuts [12]. We implemented the first two schemes in C++ and obtained the HyperCuts implementation from [36]. Since the processing speed of a networking algorithm is usually determined by the number of memory accesses required for processing a packet, we count and compare the average number of memory accesses required to process a packet in different schemes. In the paper, we define one time slot as the time required for one memory access.

To evaluate the performance of the proposed pipeline architecture, we use ClassBench tool suites developed by Taylor to generate classifiers and traffic traces [35]. Three types of classifiers are used in the evaluation: Access Control Lists (ACL), Firewalls (FW), and IP Chains (IPC). We generate two classifiers for each type using the provided filter seed files, and name them ACL1, ACL2, FW1, FW2, IPC1, and IPC2, each of which has five dimensions and about 4K rules [2].

We first evaluate the partition efficiency. Table VI shows the partition efficiencies of CB_EG and NCB_EG under classifiers ACL1, FW1, and IPC1 with a different number of partitioned hash tables (M) at each stage. Apparently, NCB_EG always outperforms CB_EG, and can achieve a partition efficiency higher than 90% in most situations. The only exception is the *Destination IP* field, where the partition efficiency achieved by NCB_EG ranges from 67% to 96%. The reason for this relatively low percentage is that level 1 of the signature tree,

[2]The generated rules are slightly less than 4K because of the existence of redundant rules. [35]

TABLE VI
THE PARTITION EFFICIENCY OF CB_EG AND NCB_EG AT DIFFERENT STAGES OF THE PIPELINE WITH DIFFERENT CLASSIFIERS

	M	Dest IP			Src Port			Dest Port			Protocol		
		# edges	CB_EG	NCB_EG	# edges	CB_EG	NCB_EG	# edges	CB_EG	NCB_EG	# edges	CN_EG	NCB_EG
ACL1	2	1568	93.44%	96.43%	1568	50.00%	100.00%	3273	65.83%	95.65%	3429	54.26%	99.97%
	3	1568	88.59%	93.84%	1568	33.33%	99.94%	3273	78.66%	95.03%	3429	37.41%	100.00%
	4	1568	86.92%	93.11%	1568	25.00%	100.00%	3273	80.38%	94.05%	3429	28.37%	99.91%
FW1	2	2400	74.77%	90.50%	2625	68.65%	97.87%	3509	73.13%	93.42%	3601	76.58%	98.87%
	3	2400	93.24%	94.90%	2625	47.14%	99.89%	3509	91.45%	99.80%	3601	52.21%	97.75%
	4	2400	73.89%	85.47%	2625	35.36%	99.89%	3509	79.03%	99.80%	3601	41.51%	96.59%
IPC1	2	2691	52.17%	67.61%	2889	62.40%	98.07%	3488	77.79%	90.04%	3588	73.16%	100.00%
	3	2691	56.27%	69.27%	2889	42.50%	99.59%	3488	61.32%	97.70%	3588	83.06%	99.92%
	4	2691	65.32%	77.24%	2889	31.87%	99.21%	3488	46.81%	99.32%	3588	64.12%	99.89%

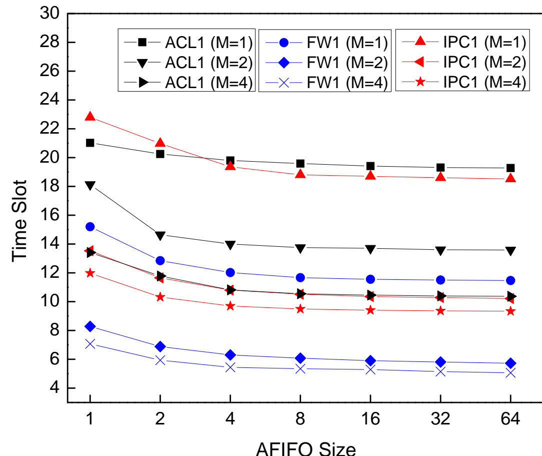


Fig. 7. Time slots for generating one result vs. AFIFO Size, when conventional hash scheme is used.

which is associated with the Destination IP field, has fewer nodes than the other levels. This small number of nodes at level 1 however each has a relatively large fan-out, which lowers the efficiency of the SPA algorithm and increases the imbalance between partitioned hash tables. Fortunately, the number of edges associated with the first stage of the pipeline is far less than that associated with other stages. Therefore, the relatively low partition efficiency would not increase too much of the storage requirement. In the remainder of this section, all simulations are conducted with the NCB_EG scheme.

In the proposed pipeline, when an AFIFO becomes full, the backpressure would prevent the upstream PM from processing new active node IDs; therefore the size of an AFIFO might affect the throughput of the pipeline to a certain extent. Fig.7 shows the relationship between AFIFO size and the average number of time slots needed for exporting one classification result when a conventional hash scheme is used. Curves in the figure show that the throughput of the pipeline is not sensitive to AFIFO size. When AFIFO size is larger than 8, the pipeline can achieve stable throughputs regardless of the classifier types or the value of M. Further increasing AFIFO size can not lead to significant throughput improvement. Therefore, in the remainder of our simulations, the sizes of AFIFOs are all set to 8 entries.

In Fig.8, we compare the proposed pipeline architecture with HyperCuts and the B2PC scheme in terms of the average time slots required for each classification operation. In the comparison, we use two different hash schemes to implement the distributed hash tables: the conventional hash scheme with $LF = 0.5$, and the proposed hybrid perfect hash scheme with $LF = 0.8$. Since the original B2PC scheme was designed to return only the most specific rule, we changed it to return all matched rules. The bucket size of HyperCuts is set to 16, and its space factor is set to 4 (optimized for speed). We suppose that each memory access of HyperCuts could read 64 bits. Based on the results in [27], we assume the single-dimensional search engines are able to return a search result in every 2.2 memory accesses (time slots).

Fig.8 shows that for IPC2 the proposed pipeline can complete one classification operation in every 3.79 time slots even when there is only one (conventional) hash table at each stage. If we use perfect hash tables to replace the conventional hash tables, the performance can be improved to 2.87 time slots for each classification operation. The reason that the pipeline with perfect hash tables performs better is that the conventional hash tables may introduce hash collisions during the lookup operations, thereby increasing the number of memory accesses for each operation. For IPC2, there is just a slight performance improvement (from 3.79 time slots/operation to 2.22 time slots/operation) when M increases from 2 to 4. This is because the packet classification speed has already reached the speed limitation of single-dimensional search engines. In contrast, for ACL1 the proposed pipeline architecture needs about 20 time slots to finish a packet classification when $M = 1$. The performance gets significantly better when M increases to 4, where the proposed pipeline can export one classification result in every 10.52 time slots with conventional hash tables and 6.27 time slots with perfect hash tables.

The proposed pipeline architecture has very strong robustness. It significantly outperforms HyperCuts and B2PC schemes for all tested classifiers. Although part of the performance improvement is gained from the parallelism of the pipeline (in fact, the B2PC scheme also employs many parallel bloom filters to accelerate its classification speed), the use of parallelism doesn't increase the overall storage cost thanks to the high partition efficiency provided by the NCB_EG scheme. For ACL2, FW1, and IPC1, the HyperCuts scheme requires more than 200 time slots on average to perform each packet classification. The reason for this slow processing speed is that these classifiers have lots of overlapping ranges/fields

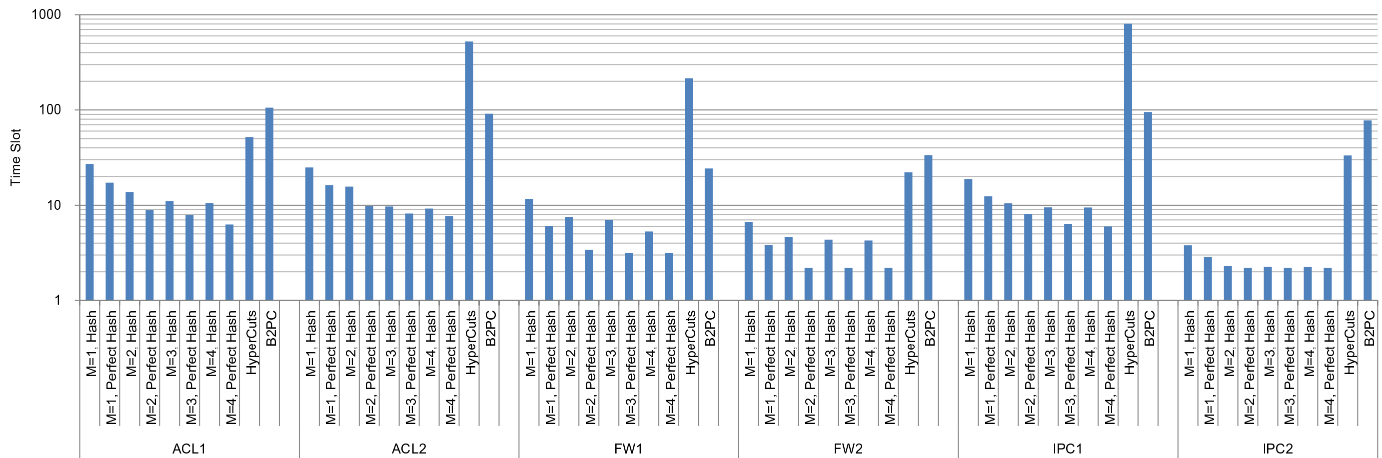


Fig. 8. Average time slots required for each classification operation under different schemes.

at source/destination IP address fields and source/destination port fields. The large number of overlapping ranges can cause a large number of rules replicated in leaves [12], which leads to a steep increase in the number of memory accesses. Although authors in [12] claimed that the performance of HyperCuts could be improved by using a pipeline, it is unclear yet what performance the pipelined-version of HyperCuts would achieve, since the last stage of the pipelined-version of HyperCuts still needs to search for a large number of replicated rules in leaf nodes.

From Fig.8, it is easy to see that using the perfect hash table can significantly improve the classification speed (by 30% $\sim$ 50% under most classifiers and settings). Table VII shows the hash table collision rate under different classifiers and settings of M when the conventional hash scheme is used to build hash tables with $LF = 0.5$.

TABLE VII
HASH TABLE COLLISION RATE WHEN CONVENTIONAL HASH SCHEME IS USED

Classifier	M=1	M=2	M=3	M=4
ACL1	0.31	0.20	0.12	0.12
ACL2	0.27	0.33	0.10	0.12
FW1	0.28	0.24	0.39	0.17
FW2	0.46	0.22	0.16	0.30
IPC1	0.31	0.22	0.18	0.20
IPC2	0.28	0.18	0.13	0.13

In Table VIII, we compare the storage cost of the proposed pipeline with those of HyperCuts and B2PC. The storage cost of the proposed pipeline consists of two major parts: (1) single dimensional search engines and (2) hash tables. According to the single-dimensional search proposed in [27], the storage requirement for a single-dimensional search engine is 78 KB, or 312 KB for four engines (the storage requirement of the single-dimensional search engine for the transport-layer protocol field is omitted here). The calculation of the storage cost of hash tables is based on formulas (10) and (11), where the load factors LF of conventional hash tables and perfect hash tables are configured as 0.5 and 0.8, respectively.

The small storage requirement allows the proposed pipeline to fit into a commodity FPGA, in which the hash tables could be implemented by on-chip SRAM. Suppose the on-chip SRAM access frequency is 400 MHz, and the smallest size of an IP packet is 64 bytes. From Fig.8, the proposed pipeline without perfect hash tables achieves the best performance under IPC2 (with 2.25 time slots/operation) and the worst performance under ACL1 (with 10.52 time slots/operation), which leads to a throughput between 19.5Gbps and 91Gbps. With the perfect hash tables, the proposed pipeline achieves the best performance under IPC2 and FW2 (with 2.2 time slots/operation) and the worst performance under ACL2 (with 7.65 time slots/operation), leading to an improved throughput of between 26.8Gbps and 93.1Gbps.

TABLE VIII
STORAGE REQUIREMENT REQUIRED BY DIFFERENT SCHEMES

Classifier	Proposed Pipeline (M=4) Convl Hash	Proposed Pipeline (M=4) Perfect Hash	HyperCuts	B2PC
ACL1	436K	390K	611K	540K
ACL2	470K	410K	214K	540K
FW1	464K	407K	3536K	540K
FW2	502K	431K	2766K	540K
IPC1	477K	415K	445K	540K
IPC2	516K	440K	1482K	540K

IX. CONCLUSION

In this paper, we model the multi-match packet classification as a concatenated multi-string matching problem, which can be solved by traversing a flat signature tree. To speed up the traversal of the signature tree, the edges of the signature tree are divided into different hash tables in both vertical and horizontal directions. These hash tables are then connected together by a pipeline architecture, and they work in parallel when packet classification operations are performed. A perfect hash table construction is also presented, which guarantees that each hash table lookup can be finished in exactly one memory access. Because of the large degree of parallelism and elaborately designed edge partition scheme, the proposed pipeline architecture is able to achieve an ultra-high packet classification speed with a very low storage requirement.

Simulation results show that the proposed pipeline architecture outperforms HyperCuts and B2PC schemes in classification speed by at least one order of magnitude with a storage requirement similar to that of the HyperCuts and B2PC schemes.

Appendix
Proof: The weighted character grouping (WCG) problem is an NP-hard Problem

We first introduce an Average-Division Problem, which has been proved in [37] to be an NP-Complete problem.

Given a finite set $S=\{1,2,\dots,N\}$, and a weight function $w:S\rightarrow Z$, we ask whether there is a subset $S^{'}\subseteq S$ satisfying

$$\sum_{i\in S'} w(i)=\frac{1}{2}\sum_{i\in S} w(i) \tag{13}$$

The Average-Division problem can be defined in a language:
AVG_DIV:=
$\{<S,w>$:
$S\subset N$,
w is a function from $N\rightarrow Z$,
there exists a subset $S^{'}\subseteq S$ such that $\sum_{i\in S'} w(i)=\frac{1}{2}\sum_{i\in S} w(i)\}$

To prove that the WCG problem is NP-hard, we first introduce the decision problem of WCG and then define it as a language. We will show that the language of AVG_DIV is reducible to the language of WCG in polynomial time.

The decision problem of the WCG problem is formally defined as follows:

Given a field set F, a set number M, a dependence indicator L, a weight function W, and a real number V, we ask whether there exists a character grouping scheme $\{G_1,...,G_{M+1}\}$ that satisfies the constraints (1) $\sim$ (6) defined in the WCG problem, as well as the new constraint:

$$\underset{k=1,\dots,M}{Max}(W(G_k)+W(G_{M+1}))\leq V \tag{14}$$

We define the WCG problem as a language:
$\{<F,M,W,V>$:
F: a set of fields; each field is defined in the form of $<a,b>$, where a and b are two real numbers satisfying $a\leq b$;
M: the number of sets;
W: a weight function from $F\rightarrow Z$;
V: a real number,
there exists a character grouping scheme $\{G_1,...,G_{M+1}\}$ satisfying the constraints above. $\}$

Now we will show that the AVG_DIV language can be reduced to WCG language in polynomial time.

Let $<S_1,w>$ be an instance of AVG_DIV. We construct an instance $<F,M,W,V>$ of WCG as follows:

Let F be the set of N independent fields; thus constraint (4) and (5) in Table V can be eliminated. For each $f_i\in F(i=1,...,N)$, let $W(f_i)=w(i)$. Let M be 2, and V be $\sum_{i\in S_1} w(i)/2$. Now the constraints can be re-constructed as follows:

$$G_1,G_2,G_3\subseteq F; \tag{15}$$

$$\bigcup G_k=F; \tag{16}$$

$$G_1\bigcap G_2=\phi,\ G_1\bigcap G_3=\phi,\ G_2\bigcap G_3=\phi; \tag{17}$$

$$W(G_k):=\sum_{f_i\in G_k} w(i)\ \ (k=1,2,3); \tag{18}$$

$$\underset{k=1,2}{Max}(W(G_k)+W(G_3))\leq\frac{1}{2}\sum_{i\in S_1} w(i). \tag{19}$$

It is easy to know that (19) is equivalent to

$$W(G_3)=0 \text{ and } W(G_1)=W(G_2)=\frac{1}{2}\sum_{i\in S_1} w(i) \tag{20}$$

Next we will show that $<S_1,w>\in$ AVG_DIV if and only if $<F,2,W,\sum_{i\in S_1} w(i)/2>\in$ WCG when $W(f_i)=w(i)$:

Suppose there exists an allocation scheme $\{G_1,G_2,G_3\}$ satisfying (15)-(20). Let $S_1^{'}=\{i|f_i\in G_1\}$. According to (18) and (20), $\sum_{i\in S_1'} w(i)=\frac{1}{2}\sum_{i\in S_1} w(i)$. Conversely, suppose there exists a set $S_1^{'}\subseteq S_1$ satisfying $\sum_{i\in S_1'} w(i)=\frac{1}{2}\sum_{i\in S_1} w(i)$. Let $G_1=\{f_i|i\in S_1^{'}\}$, $G_2=F-G_1$, and $G_3=\phi$. Apparently (15)-(20) are satisfied.

As we've seen, AVG_DIV can be reduced to WCG in polynomial time; thus WCG is an NP-hard problem. ∎

References

[1] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary cams," in *Proceedings of ACM SIGCOMM '05*, New York, NY, USA, 2005, pp. 193–204.

[2] M. Faezipour and M. Nourani, "Wire-speed tcam-based architectures for multimatch packet classification," *IEEE Trans. Comput.*, vol. 58, pp. 5–17, January 2009.

[3] ——, "Cam01-1: A customized tcam architecture for multi-match packet classification," in *IEEE Global Telecommunications Conference (GLOBECOM '06)*, dec. 2006, pp. 1 –5.

[4] F. Yu, R. H. Katz, and T. V. Lakshman, "Efficient multimatch packet classification and lookup with tcam," *IEEE Micro*, vol. 25, pp. 50–59, January 2005.

[5] F. Yu, T. V. Lakshman, M. A. Motoyama, and R. H. Katz, "Ssa: a power and memory efficient scheme to multi-match packet classification," in *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems (ANCS '05)*, New York, NY, USA, 2005, pp. 105–113.

[6] A free lightweight network intrusion detection system for UNIX and Windows. [Online]. Available: http://www.snort.org

[7] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems (ANCS '06)*, New York, NY, USA, 2006, pp. 81–92.

[8] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of ACM SIGCOMM '06*, New York, NY, USA, 2006, pp. 339–350.

[9] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24 –32, mar/apr 2001.

[10] H. J. Chao and B. Liu, *High Performance Switches and Routers*. Wiley-IEEE Press, 2007.

[11] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34 –41, jan/feb 2000.

[12] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '03)*, New York, NY, USA, 2003, pp. 213–224.

[13] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proceedings of ACM SIGCOMM '98*, New York, NY, USA, 1998, pp. 203–214.

[14] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *SIGCOMM Comput. Commun. Rev.*, vol. 28, pp. 191–202, October 1998.

[15] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput," in *Proceedings of the ACM SIGCOMM 2010 conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 207–218.

[16] H. Liu, "Efficient mapping of range classifier into ternary-cam," in *Proceedings of the 10th Symposium on High Performance Interconnects HOT Interconnects (HOTI '02)*, Washington, DC, USA, 2002, pp. 95–100.

[17] A. Liu, C. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in tcams," in *the 27th IEEE Conference on Computer Communications (INFOCOM 2008)*, april 2008, pp. 111 –115.

[18] B. Vamanan and T. N. Vijaykumar, "Treecam: decoupling updates and lookups in packet classification," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '11. New York, NY, USA: ACM, 2011, pp. 27:1–27:12.

[19] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 560 – 571, may 2003.

[20] Y.-K. Chang, C.-I. Lee, and C.-C. Su, "Multi-field range encoding for packet classification in tcam," in *Proceedings of IEEE INFOCOM 2011*, April 2011, pp. 196–200.

[21] A. Bremler-Barr, D. Hay, and D. Hendler, "Layered interval codes for tcam-based classification," in *Proceedings of IEEE INFOCOM 2009*, April, pp. 1305–1313.

[22] R. McGeer and P. Yalagandula, "Minimizing rulesets for tcam implementation," in *Proceedings of IEEE INFOCOM 2009*, April 2009, pp. 1314–1322.

[23] R. Wei, Y. Xu, and H. Chao, "Block permutations in boolean space to minimize tcam for packet classification," in *Proceedings of IEEE INFOCOM 2012*, March 2012, pp. 2561–2565.

[24] Y. Ma and S. Banerjee, "A smart pre-classifier to reduce power consumption of tcams for multi-dimensional packet classification," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 335–346, Aug. 2012.

[25] D. Pao, Y. K. Li, and P. Zhou, "An encoding scheme for tcam-based packet classification," in *the 8th International Conference Advanced Communication Technology (ICACT 2006)*, vol. 1, feb. 2006, pp. 6 pp. –475.

[26] I. Papaefstathiou and V. Papaefstathiou, "Memory-efficient 5d packet classification at 40 gbps," in *the 26th IEEE International Conference on Computer Communications (INFOCOM 2007)*, may 2007, pp. 1370 –1378.

[27] ——, "An innovative low-cost classification scheme for combined multi-gigabit ip and ethernet networks," in *IEEE International Conference on Communications (ICC '06)*, vol. 1, june 2006, pp. 211 –216.

[28] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *SIGCOMM Comput. Commun. Rev.*, vol. 27, pp. 3–14, October 1997.

[29] G. Varghese, *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[30] X. Sun, S. K. Sahni, and Y. Q. Zhao, "Packet classification consuming small amount of memory," *IEEE/ACM Trans. Netw.*, vol. 13, pp. 1135–1145, October 2005.

[31] N. Sertac Artan and H. Chao, "Tribica: Trie bitmap content analyzer for high-speed network intrusion detection," in *Proceedings of IEEE INFOCOM 2007*, may 2007, pp. 125 –133.

[32] R. Pagh and F. Rodler, "Cuckoo hashing," in *ESA*, 2001.

[33] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: Deterministic and updatable hashing for high performance networking," in *Proceedings of IEEE INFOCOM 2008*, april 2008, pp. 101 –105.

[34] G. L. Heileman and W. Luo, "How caching affects hashing," in *Proc. the 7th Workshop on Algorithm Engineering and Experiments (ALENEX '05)*, 2005, pp. 141–154.

[35] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, pp. 499–511, June 2007.

[36] H. Song. Multidimensional cuttings (hypercuts). [Online]. Available: http://www.arl.wustl.edu/ hs1/project/hypercuts.htm

[37] K. Zheng, C. Hu, H. Lu, and B. Liu, "A tcam-based distributed parallel ip lookup scheme and performance analysis," *IEEE/ACM Trans. Netw.*, vol. 14, pp. 863–875, August 2006.

Yang Xu (S'05-M'07) is a Research Assistant Professor in the Department of Electrical and Computer Engineering in Polytechnic Institute of New York University, where his research interests include Data Center Network, Network on Chip, and High Speed Network Security. From 2007-2008, he was a Visiting Assistant Professor in NYU-Poly. Prior to that, he completed a Ph.D. in Computer Science and Technology from Tsinghua University, China in 2007. He received the Master of Science degree in Computer Science and Technology from Tsinghua University in 2003 and Bachelor of Engineering degree from Beijing University of Posts and Telecommunications in 2001.

Zhaobo Liu received M.S. degree in Telecommunication Networks from Polytechnic Institute of New York University, Brooklyn in 2009 and M.S., B.S degrees in Electronics Engineering from Beijing Institute of Technology University, Beijing, China, in 2006 and 2004, respectively. He is currently working as Software Engineer in Vmware, Inc.

Zhuoyuan Zhang received B.S. of Electrical Engineering from Polytechnic Institute of New York University in 2010. He's been with Cloud Drive Team in Amazon Inc., and currently with Teza Technologies, NY.

H. Jonathan Chao (M'83-F'01) is Department Head and Professor of Electrical and Computer Engineering at Polytechnic Institute of New York University, Brooklyn, NY, where he joined in January 1992. He has been doing research in the areas of network designs in data centers, terabit switches/routers, network security, network on the chip, and biomedical devices. He holds 45 patents and has published over 200 journal and conference papers. He has also served as a consultant for various companies, such as Huawei, Lucent, NEC, and Telcordia. During 2000-2001, he was Co-Founder and CTO of Coree Networks, NJ, where he led a team to implement a multi-terabit MPLS (Multi-Protocol Label Switching) switch router with carrier-class reliability. From 1985 to 1992, he was a Member of Technical Staff at Telcordia. He received the Telcordia Excellence Award in 1987. He is a Fellow of the IEEE for his contributions to the architecture and application of VLSI circuits in high-speed packet networks. He coauthored three networking books, Broadband Packet Switching Technologies, Quality of Service Control in High-Speed Networks, and High-Performance Switches and Routers, all published by Wiley. He received his PhD in EE from The Ohio State University in 1985.