# An Ultra High Throughput and Memory Efficient Pipeline Architecture for Multi-Match Packet Classification without TCAMs

Yang Xu, Zhaobo Liu, Zhuoyuan Zhang, H. Jonathan Chao
Polytechnic Institute of New York University
yangxu@poly.edu, {zliu01, zzhang04@students.poly.edu}, chao@poly.edu

## ABSTRACT

The emergence of new network applications like network intrusion detection system, packet-level accounting, and load-balancing 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 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 in multiple hash tables at different stages of the pipeline, 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 with minimum overhead. Extensive simulation using realistic classifiers and traffic traces shows that the proposed pipeline architecture outperforms HyperCut and B2PC schemes in classification speed by at least one order of magnitude, while with a similar storage requirement. Particularly, with different types of classifiers of 4K rules, the proposed pipeline architecture is able to achieve a throughput between 19.5 Gbps and 91 Gbps.

## Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General Security and protection (e.g., firewalls); C.2.6 [**Internetworking**]: Routers

## General Terms

Algorithms, Performance, Design, Experimentation, Security

## Keywords

Packet Classification, Signature Tree, TCAM, Hash Table


*ANCS'09*, October 19-20, 2009, Princeton, New Jersey, USA.


## 1. 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 [2][3][4][5], to differ from the conventional best-match packet classification.

Typical NIDS systems, like SNORT [6], use multi-match packet classification as a pre-processing module to filter out benign network traffic and so that 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 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 huge memory requirement or steep performance degradation under certain types of classifiers [13][18]. Ternary Content Addressable Memory (TCAM) is well-known for its parallel search capability and constant processing speed, and is widely used in IP routing 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 work 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 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 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 on 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. 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 could 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 could be exploited. By taking advantage of the properties of the signature tree, the proposed edge-grouping algorithms well solve the location problem, overhead minimization problem, and balancing problem involved in the process of hash table partition.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 formally defines the multi-match packet classification problem, and presents terms to be used in this paper. Section 4 introduces the concept of signature tree, based on which Section 5 proposes an asynchronous pipeline architecture. Section 6 presents two edge-grouping algorithms which are used to exploit intra-stage parallel query. In Section 7, we discuss the implementation issues and present the experimental results. Finally Section 8 concludes the paper.

## 2. RELATED WORK

Many schemes have been proposed in literature to address the best-match packet classification problem, such as trie based schemes [9][12], decision tree based schemes[11][13], TCAM based schemes [14][15], and two-stage schemes[12][18][19][20]. However, most of them can not be used directly in multi-match packet classification.

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 two-stage schemes is usually encoded as either one range ID or several segment IDs. Consider the classifier shown in Figure 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 rule 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 segment encoding method, each rule is represented by multiple segment ID combinations, which may cause serious storage explosion problem [12][18]. Several schemes [19][20] have been proposed to address the storage explosion problem by using TCAM and specially designed encoding scheme. However, the use of TCAM increases the power consumption and implementation cost, and more importantly, it limits the use of the schemes only in best-match packet classification.

With 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 [16], which uses multiple Bloom Filters to fasten 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 returns only the first matched range ID combination, which may not always correspond to the highest-priority matched rule due to the inherent limitation of B2PC scheme. Furthermore, B2PC can not support multi-match packet classification.

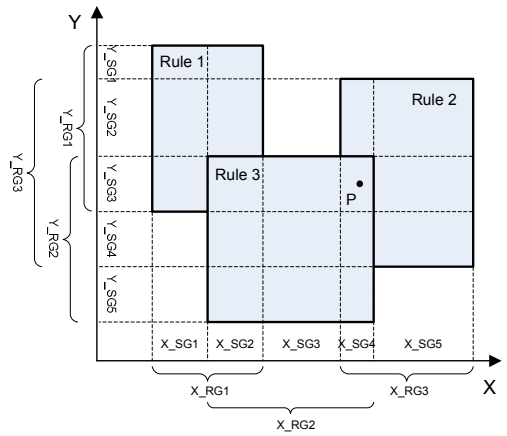



**Figure 1. Segment encoding vs. range encoding.**

## 3. 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 on 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 covert it to an encoded counterpart by assigning each unique range a unique ID on each dimension. Given the classifier in Table 1, its encoded counterpart is shown in Table 2, in which $f_{ij}$ is the ID of the $j^{th}$ unique range appeared on the $i^{th}$ dimension of the classifier.

**Table 1. A classifier with seven rules**

| Rule | Src IP | Dest IP | Src Port | Dest Port | Protocol |
|---|---|---|---|---|---|
| $r_1$ | 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 2. 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 3. 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 |

**Table 4. Range IDs returned by single-dimensional searches**

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

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 3, the range IDs returned from five single-dimensional search engines are shown in Table 4, and can form 1x2x3x3x2=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 have been well addressed in literature [17], in this paper we focus on only the second step. In the left part of the paper, packet classification will specifically refer to this second step unless special notation is given.

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 *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.

# 4. 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. Figure 2 shows a signature tree corresponding to the encoded classifier in Table 2. Each edge in 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.

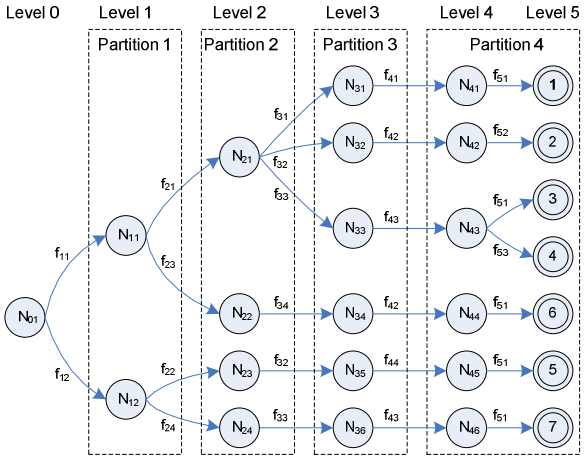



**Figure 2. An example of signature tree.**

The concatenated multi-string matching could 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 would 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 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. 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 pointer compressing scheme [21], 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.

Apparently, the processing speed of the signature tree based packet classification is determined by the number of hash table accesses required for classifying each packet. In the following sections, we divide the hash table into multiple partitions to exploit parallel hash table access to 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 could be encoded as any bit strings as long as there are no two characters being given the same encoding.*

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

# 5. 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 Figure 2, which has four partitions. The corresponding hash tables of these four partitions are shown in Figure 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 as the ID of the corresponding destined level-1 node. For instance, in Figure 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 level-1 active nodes.

For a $d$-dimensional packet classification application, we propose an asynchronous pipeline architecture with $d$-1 stages. Figure 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 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 at a certain partition of the signature tree.

Since each packet may have multiple matching characters/active nodes at each 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 Figure 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.

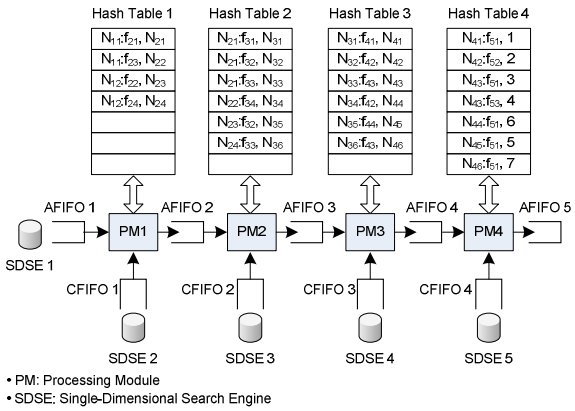



**Figure 3. Pipelining architecture for packet classification.**

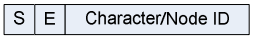



**Figure 4. The format of entries in CFIFO/AFIFO.**

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 $P$ have $x$ active node IDs in AFIFO $i$, which are denoted by $n_1$, $n_2$, …, $n_x$, and $y$ matching characters in CFIFO $i$, which are denoted by $c_1$, $c_2$, …, $c_y$. The processing of packet $P$ at PM $i$ could be decomposed to the processing of $x$ active node IDs. In the processing of each active node ID, say $n_j$, PM $i$ takes out matching character $c_1$, $c_2$, …, $c_y$ from the attached CFIFO, and concatenate each of them (if the character is not NULL) to $n_j$ to form $y$ hash keys to access the attached hash table. Results from the hash table indicate the IDs of $n_j$'s child nodes, and will be pushed into the output AFIFO when the output AFIFO 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, it will push a "NULL" node ID encoded as all "0" into the output AFIFO to indicate the downstream PMs that the packet won't match any rule.

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

# 6. 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 hash table accesses. 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 could be accessed in parallel. To keep the pipeline easy to control and avoid 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 next active node ID could not 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 but not exactly the same as the concept of independent rule set proposed by Sun et al. in [23].

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 to $f_2$ 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 $f_2$. $c_1$ is called independent to $c_2$ if $f_1$ is independent to $f_2$.

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 to $e_2$ if $s_1$=$s_2$ and $c_1$ is dependent to $c_2$; otherwise, $e_1$ is called independent to $e_2$.

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 table 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.

[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.

## 6.1 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 could be maximized and 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 to 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 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 to 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 would 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 not fitting into the first $M$ sets. The above action is called *edge-grouping*. We call edges in the independent edge sets regular edges, and call edges in the residual edge set residual edges.

Given the $M$+1 edge sets after the edge-grouping, we could store edges of each independent edge set into 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 is 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 aspects.

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

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

## 6.2 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 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 to be 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 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 solved.

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

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 to $c_2$, $L(c_1, c_2)$:=1; otherwise, $L(c_1, c_2)$:=0.

The WCG problem is formally described in Table 5, which is to find a valid configuration of $U_k$ ($k$=1,…,$M$+1) achieving the given objective. We have proved that WCG problem is an NP-hard problem (due to space limitations, the proof is not given in the paper). Thus, we use the greedy algorithm in Table 6 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 could get $M$ partitioned hash tables by allocating edges of $G_k$ ($k$=1,…,$M$) to hash table $k$, and duplicating edges of $G_{M+1}$ to every hash table.

Let's consider an example, which partitions the last hash table of Figure 3 into two work-conserving hash tables with 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 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 to both partitioned hash tables. The final partition result is shown in Figure 5(a).

We use $PE$ to denote the partition efficiency of a hash table partition, and define it in (1).

$$PE = \frac{\#\text{ of edges in the original hash table}}{M \times \#\text{ of edges in the largest partitioned hash table}} \quad (1)$$

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

As a matter of fact, in real classifiers, the second reason degrades the partition efficiency more severely than the first reason. This is because many fields of real classifiers have very unbalanced distributions on 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, and etc. Most of entries, say 80%, of real classifiers are associated with TCP protocol. With CB_EG scheme, edges labeled with TCP have to be allocated to the same hash table, which may cause extreme unbalance of hash tables, and thus result in a low partition efficiency.

**Table 5. The weighted character grouping problem**

**Subject to**.

$$U_k \subseteq U \quad (k = 1,...,M+1)\ ; \quad (2)$$

$$\bigcup_k U_k = U\ ; \quad (3)$$

$$U_{k1} \bigcap U_{k2} = \phi \quad (k1, k2 = 1,...,M+1 \,\&\, k1 \neq k2) \quad (4)$$

$$L(c_1,c_2) := \begin{cases} 1 & c_1 \textit{ is depedent to } c_2\ (c_1,c_2 \in U) \\ 0 & c_1 \textit{ is independent to } c_2\ (c_1,c_2 \in U) \end{cases} \quad (5)$$

$$L(c_1,c_2) = 0 \quad \forall c_1,c_2 \in U_k (k = 1,...,M) \quad (6)$$

$$W(U_k) := \sum_{c \in U_k} W(c) \quad (7)$$

**Objective**.

$$\textbf{Minimize}: \ \underset{k=1,...,M}{Max} (W(U_k) + W(U_{M+1})) \quad (8)$$

**Table 6. Greedy algorithm for the WCG problem**

**Input**: $U$: the universal character set; $M$: the number of independent character sets; $W(c)$: the weight of character $c$.

**Output**: Independent character sets $U_1$,… ,$U_M$ ; and residual character set $U_{M+1}$.

$U_k := \phi\ (k = 1,...,M+1)$ ;

$W(U_k) := 0\ (k = 1,...,M)$ ; //the weight of set $U_k$

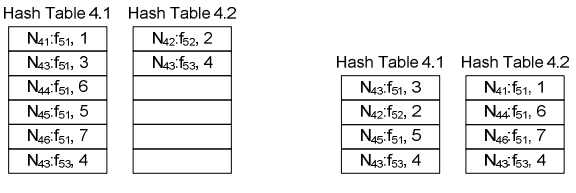
Sort $U$ in decreasing order of the character weight.

(1) if $U$ is empty, return ( $U_k$($k$=1,…,$M$+1) );

(2) From $U$ select the character $c$ with the largest weight;

(3) Select the set $U'$ with the smallest weight among sets $U_1$,… ,$U_M$ whose characters are all independent to $c$. 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 (1);

(4) Put $c$ into set $U'$ ; remove $c$ from set $U$; $W(U')+=W(c)$ ; Go to step (1).




(a) CB_EG scheme (b) NCB_EG scheme

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

## 6.3 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 could be assigned to any values as long as there are no two nodes at the same level assigned the same ID. With this property, 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, NCB_EG scheme reserves the first $\lceil \log_2(M+1) \rceil$ bit of each character to be the *locating prefix*, and the first $\lceil \log_2 M \rceil$ bits of each node ID to be 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 could be found in any partitioned hash tables. Otherwise, the edge is a regular edge, and could 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, NCB_EG scheme solves the overhead minimization problem and the balance problem in two different steps, which are named Locating Prefix Assignment (LPA), and Shift Prefix Assignment (SPA).

The overhead minimization problem is solved in the step of LPA, 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 could also be described by the WCG problem given in Table 5 with only the objective changed from (8) 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 Table 6 to solve it.

The purpose of the SPA step is to balance the sizes of independent edge sets. This is achieved by assigning shift prefix to each node to adjust the edge sets in which the outgoing edges of the node are allocated. A heuristic algorithm for the shift prefix assignment is given in Table 7.

Consider using NCB_EG scheme to partition the last hash table of Figure 3 into two work-conserving hash tables. The LPA step of NCB_EG is same to the 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 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 be first 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. After the edge-grouping, the final partitioned hash tables are shown in Figure 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 CB_EG scheme. The main reason for this improved partition efficiency is that NCB_EG scheme is capable of spreading edges labeled with character $f_{51}$ into different hash tables, so that a better balance between partitioned hash tables is achieved.

**Table 7. Shift Prefix Assignment Algorithm**

**Input**:
$M$: the number of independent character sets; Independent character set $U_1,\ldots,U_M$, and residual character set $U_{M+1}$;
$S$: the set of nodes at level $i$ of the signature tree;
$E$: the set of outgoing edges of nodes in $S$;

**Output**:
Shift prefixes of nodes in $S$;
Independent edge sets $G_1,\ldots,G_M$, and residual edge set $G_{M+1}$;

$G_k := \phi\ (k = 1,...,M+1)$ ;
Sort nodes of $S$ in decreasing order of the number of outgoing edges;
(1) **for** each node $n$ in $S$ **do**
(2) Divide the outgoing edges of $n$ into $M$+1 sets. The principle is that for characters in $U_k$ ($k$=1,…,$M$+1), put their associated outgoing edges to $Z_k$;
(3) Select the largest edge set $Z_t$ among $Z_k$ ($k$=1,…,$M$); if there are multiple largest edge set, select the first one;
(4) Select the smallest edge set $G_v$ among $G_k$ ($k$=1,…,$M$); if there are multiple smallest edge set, select the first one;
(5) Let $p$:= ($v$-$t$) mod $M$, and set the shift prefix of $n$ as $p$;
//align $Z_t$ to $G_v$ to achieve balance among $G_k$ ($k$=1,…,$M$);
(6) **for** each set $Z_k$ ($k$=1,…,$M$) **do**
(7) Move edges from set $Z_k$ to set $G_{(k+p-1) \bmod M+1}$;
(8) **rof**;
(9) Move edges from set $Z_{M+1}$ to set $G_{M+1}$;
(10) **rof**;

# 7. IMPLEMENTATION ISSUES AND PERFORMANCE EVALUATION

## 7.1 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 hash tables. Since the complexity of insertion/remove operation in hash table is O(1), the pipeline architecture has a very low complexity for incremental update.

## 7.2 Hash Tables and 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 to 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 is required to be the single-dimensional search engine, which might require 70~100 Kbytes of memory [17].

For hash tables in the pipeline, we use a low load factor ($LF$) of 0.5 to lower the chance of hash collisions, and use the simple linear probing scheme to resolve the hash collision [22]. The storage requirement for hash tables at each stage ($H$) is determined by the number of edges associated to 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. $H$ could be represented by (10).

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

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 $\lceil \log_2 M \rceil$ bit are the shift prefix, and the last $\lceil \log_2 N \rceil$ bit are used to uniquely identify the node at each level of the signature tree. The number of characters in the universal character set on 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. Sum up, the number of bits used for representing each edge could be obtained in (11).

$$B \le 3\lceil \log_2 N \rceil + 2\lceil \log_2 M \rceil + \lceil \log_2 (M+1) \rceil \quad (11)$$

The number of edges to be stored at each stage of the pipeline is bounded by $N$, therefore $T \le N$ . 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 $LF$, $T$ and (11) into (10), we can get the total storage requirement of the hash tables at each stage as in (12).

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

Since there are $d$-1 stages in the pipeline, the total memory requirement is about $N(d\text{-}1)\log_2 N$ bytes.

Regarding AFIFOs and CFIFOs, later we will show that each AFIFO/CFIFO only needs 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 could be ignored 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 about $N(d\text{-}1)\log_2 N$ bytes. If we substitute $N$=4K, $d$=5 in it, the storage requirement is about 192 Kbytes, which is among the compact packet classification schemes proposed in literature even if we count in the memory required by the single dimensional search engines.

## 7.3 Performance Evaluation

To evaluate the performance of the proposed pipeline architecture, we use ClassBench tool suites developed by Taylor to generate classifiers and traffic traces [24]. Three types of classifiers are used in the evaluation, which are 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 as 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 8 shows the partition efficiencies of CB_EG and NCB_EG under classifier ACL1, FW1, and IPC1 with the number of partitioned hash tables ($M$) at each stage changed from 2 to 4. 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 partition efficiency is because Destination IP field corresponds to level 1 of the signature tree, which has fewer nodes than other levels, although each node at level 1 has a large fan-out. The small number of large fan-out nodes lowers the efficiency of SPA algorithm, and thus increase the unbalance between partitioned hash tables. Fortunately, the number of edges associated to the first stage of the pipeline is far less than that associated to other stages. Therefore the relatively low partition efficiency would not increase too much of the storage requirement. In the left part of this section, all simulations are conducted with 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 AFIFO might affect the throughput of the pipeline to a certain extent. Figure 6 shows the relationship between AFIFO size and the average time slots for exporting one classification result, where one time slot is defined as the time for one memory access. Curves in the figure show that the throughput of the pipeline is not sensitive to the AFIFO size. When AFIFO size is large than 8, the pipeline can achieve stable throughputs regardless of the classifier types and the value of $M$. Further increasing the AFIFO size can not lead to significant throughput improvement. Therefore, in the left part of simulations, the sizes of AFIFOs are all set to 8 entries.

[2] The generated rules are slightly less than 4K because of the existence of redundant rules.[24]

**Table 8. The partition efficiency of CB_EG and NCB_EG at different stages of the pipeline with different classifiers**

| Classifer | # of Hash Tables (M) | Dest IP | | | Src Port | | | Dest Port | | | Protocol | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | # of edges | CB_EG | NCB_EG | # of edges | CB_EG | NCB_EG | # of edges | CB_EG | NCB_EG | # of 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% |

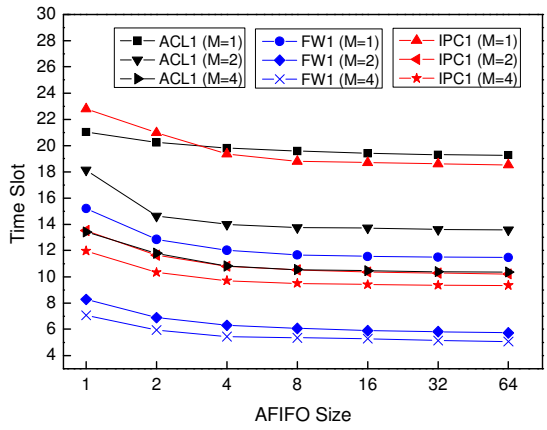



**Figure 6. Time slots for generating one result vs. AFIFO Size**

In Table 9, we compare the proposed pipeline architecture with HyperCut [13] and B2PC scheme [16] in terms of the average time slots for each classification operation. Since the original B2PC scheme was designed to return only the most specific rule, we made changes on it to return all matched rules. The bucket size of HyperCut is set to 16, and its space factor is set to 4 (optimized for speed). We suppose that each memory access of HyperCut could read 64 bits. When measuring the time slots of processing each packet for the proposed pipeline, we don't count in the time spent for single-dimensional searches. This is because single-dimensional search engines [17] are able to return a search result in every 2.2 memory accesses (time slots), which is smaller than the time spent by the pipeline. When single-dimensional search engines operate in parallel with the pipeline architecture, they won't affect the pipeline's throughput.

Table 9 shows that for IPC2 the proposed pipeline can complete one classification operation in every 3.79 time slots even there is only one hash table at each stage. The performance improvement is not obvious when *M* increases from 2 to 4, because the packet classification speed has already reached the speed limitation of single-dimensional search engines. In contrast, for ACL1 and ACL2, the proposed pipeline architecture needs more than 20 time slots to finish a packet classification when *M*=1. The performance gets significantly improved when *M* increases to 4. When we put 4 hash tables at each stage, the proposed pipeline can export one classification result in every 10.52 time slots even with the worst classifier. The proposed pipeline architecture has very strong robustness. It significantly outperforms HyperCut and B2PC schemes for all tested classifiers. Although part of the performance improvement is gained from the parallelism of the pipeline (in fact, 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 NCB_EG scheme.

For ACL2, FW1, and IPC1, HyperCut scheme requires more than 200 time slots on average to perform each packet classification. The reason for this slow processing speed is because that these classifiers have lots of overlapping ranges/fields 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 [13], which leads to a steep increase in the number of memory accesses. Although authors in [13] claimed that the performance of HyperCut could be improved by using pipeline, it is unclear yet what performance the pipelined-version HyperCut would achieve, since the last stage of the pipelined-version HyperCut still need to search a large number of replicated rules in leaf nodes.

**Table 9. Average time slots required for classifying a packet**

| Classifier | Proposed Pipeline | | | | HyperCuts | B2PC |
|---|---|---|---|---|---|---|
| | M=1 | M=2 | M=3 | M=4 | | |
| ACL1 | 27.14 | 13.73 | 11.05 | 10.52 | 52.02 | 105.78 |
| ACL2 | 24.90 | 15.68 | 9.70 | 9.22 | 523.34 | 91.34 |
| FW1 | 11.67 | 7.50 | 7.03 | 5.30 | 215.17 | 24.31 |
| FW2 | 6.66 | 4.61 | 4.35 | 4.26 | 22.12 | 33.45 |
| IPC1 | 18.81 | 10.46 | 9.48 | 9.46 | 803.64 | 95.19 |
| IPC2 | 3.79 | 2.30 | 2.26 | 2.25 | 33.28 | 77.65 |

**Table 10.Storage requirement required by different schemes**

| Classifier | Proposed Pipeline | HyperCuts | B2PC |
|---|---|---|---|
| ACL1 | 504K | 611K | 540K |
| ACL2 | 504K | 214K | 540K |
| FW1 | 504K | 3536K | 540K |
| FW2 | 504K | 2766K | 540K |
| IPC1 | 504K | 445K | 540K |
| IPC2 | 504K | 1482K | 540K |

In Table 10, we compare the storage costs of three algorithms. According to the analysis in section 7.2, we know that the worst-case storage requirement of the proposed pipeline is about 192 Kbytes. According to the single-dimensional search proposed in [17], which requires 78Kbytes of memory, the total storage requirement of the pipeline including four single dimensional

search engines is about 504 Kbytes (the storage requirement of the single-dimensional search engine for transport-layer protocol field is omitted here). The small storage requirement makes the proposed pipeline be able to fit into a commodity FPGA, on which the hash tables could be implemented by on-chip SRAM. Suppose the on-chip SRAM access frequency is 400 MHz, the smallest size of IP packet is 64 bytes. The proposed pipeline can achieve a throughput between 19.5Gbps and 91Gbps with different types of classifiers.

Table 11 shows the hash table collision rate under different classifiers and settings of *M*. In fact, if we further reduce the load factor of hash tables, a lower hash collision rate could be achieved, which may bring an even higher pipeline throughput.

**Table 11. Hash table collision rate**

| 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 |

# 8. CONCLUSION

In this paper, we model the multi-match packet classification as a concatenated multi-string matching problem, which could 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 work in parallel when packet classification operations are performed. 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 HyperCut and B2PC schemes in classification speed by at least one order of magnitude with a similar storage requirement of HyperCut and B2PC schemes.

# 9. REFERENCES


[1] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for Advanced Packet Classification with Ternary CAMs. In Proc. ACM SIGCOMM 2005.

[2] M. Faezipour and M. Nourani. Wire-Speed TCAM-Based Architectures for Multimatch Packet Classification. IEEE Transactions on Computers, Volume, 58, Issue, 1, Jan. 2009.

[3] M. Faezipour and M. Nourani. A Customized TCAM Architecture for Multi-Match Packet Classification. In Proc. IEEE GLOBECOM 2006.

[4] F. Yu, R.H. Katz, and T.V. Lakshman. Efficient Multimatch Packet Classification and Lookup with TCAM. In Proc. IEEE HOTI 2004, pp. 28-34.

[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 Proc. ACM/IEEE ANCS '05.

[6] SNORT Network Intrusion Detection System, www.snort.org.

[7] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In Proc. ACM/IEEE ANCS'06, pp. 81-92, 2005.

[8] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In Proc. ACM SIGCOMM, Sept 2006.

[9] P. Gupta and N. McKeown. Algorithms for Packet Classification. IEEE Network. March/April 2001, v15, n2, pp 24-32.

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

[11] P. Gupta and N. McKeown. Packet Classification using Hierarchical Intelligent Cuttings. In IEEE Micro, vol. 20:1, Jan/Feb 2000, pp 34-41.

[12] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvagel. Fast and scalable layer four switching. In ACM SIGCOMM, Vancouver, Canada, Aug. 1998, pp. 191-202.

[13] S. Singh, F. Baboescu, G.Varghese, and J.Wang. Packet classification using multidimensional cutting. In ACM SIGCOMM, Karlsruhe, Germany, Aug. 2003, pp. 213-224.

[14] H. Liu. Efficient Mapping of range classifier into Ternary-CAM. In Proc. of 10th Hot Interconnects, Stanford, CA, Aug. 2002, pp. 95-100.

[15] A. X. Liu, C. R. Meiners, and Y. Zhou. All-Match Based Complete Redundancy Removal for Packet Classifiers in TCAMs. In Proc. IEEE INFOCOM'08, March 2008.

[16] I. Papaefstathiou and V. Papaefstathiou. Memory-Efficient 5D Packet Classification At 40 Gbps. In Proc. IEEE INFOCOM'07, May 2007.

[17] I. Papaefstathiou and V. Papaefstathiou. An innovative low-cost Classification Scheme for combined multi-Gigabit IP and Ethernet Networks. In Proc. IEEE ICC'06, June 2006.

[18] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In ACM SIGCOMM'98, pp. 203-214.

[19] J. van Lunteren and T. Engbersen. Fast and scalable packet classification. IEEE Journal on Selected Areas in Communications, vol. 21, May 2003, pp. 560-571.

[20] D. Pao, Y. K. Li, and P. Zhou. An encoding scheme for TCAM-based packet classification. The 8th International Conference on Advanced Communication Technology, Volume 1, 20-22 Feb. 2006.

[21] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In Proc. ACM SIGCOMM 1997, pp. 3-14.

[22] G. L. Heileman and W. Luo. How caching affects hashing. In Proc. the 7th Workshop on Algorithm Engineering and Experiments (ALENEX '05), pp. 141-154.

[23] X. Sun, S.K. Sahni, and Y.Q. Zhao. Packet classification consuming small amount of memory. In IEEE/ACM Transactions on Networking, Volume:13, Issue: 5, Oct. 2005.

[24] D. Taylor and J. Turner. ClassBench: A Packet Classification Benchmark. In Proc. IEEE INFOCOM'05, March 2005.