TupleChain: Fast Flow Table Lookup with Efficient On-line Updates and Scalability

Yanbiao Li*†, Neng Ren*†, Xin Wang‡, Yuxuan Chen*†, Xinyi Zhang*†, Lingbo Guo*†, and Gaogang Xie*† *
Computer Network Information Center, *Chinese Academy of Sciences*, Beijing, China † *School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, Beijing, China ‡ *Department of Electrical and Computer Engineering*, *Stony Brook University*, Stony Brook, NY, USA

***Abstract*—Packet classification is a fundamental operation in modern network systems, playing a central role in traffic management, security enforcement, and policy execution. While traditional algorithms have achieved low lookup latency in static scenarios, emerging applications impose more demanding requirements—not only fast lookup, but also high-frequency online updates and strong scalability. Existing approaches often struggle to handle large rule sets or support rules with an increasing number of matching fields, making them unsuitable for dynamic and large-scale network environments.In this work, we propose TupleChain for fast on-line update table lookup with multifaceted scalability. We group rules based on their masks, each maintained using a hash table, and explore the connections among rule groups to skip unnecessary hash probes for faster searches. We show via theoretical analysis and extensive experiments that the proposed scheme offers competitive computational complexity, strong scalability, and high performance in both search and update operations. TupleChain can process millions of packets per second, while simultaneously handling millions of on-line updates per second at the same time, and its lookup speed remains stable even when processing large flow table with 10 million rules or entries containing up to 100 match fields.**

***Index Terms*—packet classification, lookup, on-line updates, scalability**

I. Introduction

A. Background

AMID the continuously evolving landscape of network policies, modern data planes face dual challenges: high-speed online updates and scalability to large rule sets. In real-world scenarios, the performance demands on packet classification systems are becoming increasingly stringent. For instance, in cloud computing environments, to prevent security policy lapses, cloud providers must deploy millions of security group rules across distributed systems within a single BGP convergence window. According to publicly disclosed AWS practices, this requires a global update capability of over 500,000 rules per second [12]. Meanwhile, in vehicular networks, edge gateways must dynamically adjust V2X message prioritization in response to real-time traffic events, with update frequencies exceeding 1,000 rules per second, in order to meet the requirements of ultra-reliable low-latency communications (URLLC) [4].

To meet these demands, packet classification systems must support both efficient lookup and rapid updates. Their core functionality involves locating the best-matching rule in a rule set based on incoming packet header fields, and efficiently inserting, deleting, or modifying rules in response to policy changes. Each rule typically consists of multiple matching fields (e.g., source/destination IP, port numbers, protocol type) along with a priority value, which determines the effective rule when multiple matches occur. The lookup process must quickly identify the highest-priority match to ensure correct forwarding decisions, while the update process must allow real-time rule modifications without disrupting service, even at scale. Although prior research has significantly reduced lookup latency through various optimizations, update delay and scalability have emerged as the primary bottlenecks in modern classification systems. Consequently, there is an urgent need for a new class of packet classification algorithms that deliver fast lookups, sub-millisecond updates, and scalable rule management, in order to meet the demands of the next-generation programmable, secure, and adaptive network infrastructures.

B. Problem Statement and Scalability Challenges

Given a flow table composed of rules each with d-field, for an incoming packet, flow lookup is performed by matching the d-field key extracted from the packet header (and optionally some metadata) against the flow table to find a rule that matches this packet and has the highest priority. The problem of multi-field rule-matching has been well studied for packet classification. However, in addition to the high speed, flow lookup poses stronger demands on the scalability.

First, a flow lookup scheme must work well in the presence of highly frequent rule updates [14]. Generally, to reduce their influence on the lookup process, rule updates must be performed as quickly as they arrive. However, this is exactly what most packet classification algorithms [6], [7], [17] lack. Recent attentions have been drawn to both fast lookup and fast updates [8], [16], [24], [26].

Second, a flow lookup scheme should work well with large data sets. In a typical data center, an edge switch may have to handle more than 1 million flows [2], while a gateway router at the border of autonomous system may handle about 0.8 million forwarding rules [1]. Intuitively, a future-proof flow lookup scheme should work well when handling millions of rules. However, this scale is 2 orders of magnitude larger than the largest data sets tested with existing solutions [8], [16], [24], [26].

Lastly, a flow lookup scheme should work well with rules having a large number of fields. In modern network envi-

ronments, a single rule often requires joint matching across multiple fields. Beyond the conventional 5-tuple, this may include extended fields such as packet length, tags, application identifiers, and path attributes. The rise of Network Functions Virtualization (NFV), zero-trust architectures, and custom protocol requirements has led to a continuous increase in the number of rule fields.

C. Our Contributions

In this paper, we propose *TupleChain*, a novel flow lookup scheme with both fast lookup and efficient updates, in view of not only computation complexity but also practical performance. Most importantly, its excellent scalability in the aforementioned three aspects is clearly demonstrated. We summarize the main contributions of this paper as follows:

1) We propose the use of a directed acyclic graph to track the connections between rule groups created following the *tuple space search* model [21]. With every rule group referred as a *tuple*, we name this graph a *tuple graph*. On this basis, we introduce two types of lookup guidance, where the tuple connections thus the edges of *tuple graph* are exploited to skip unnecessary searches in flow lookup.
2) We propose the use of tuple chains to trade off between the skipping of search operations and the maintaining of lookup guidance information. We group edges in *tuple graph* into several tuple chains, where every chain is unidirectional and follows a monotonic sequence.
3) We present a series of algorithms based on tuple chains for flow lookup, rule updates and maintenance of guidance information.
4) We analyze the complexity to show that our scheme *TupleChain* supports fast lookup and fast updates with the cost effectively amortized.
5) We extend *TupleChain* to further boost its performance with the optimal construction of tuple chains, the adjustment of inner structure of a tuple chain during tuple insertion, as well as the increase of runtime speed and scalability.
6) We evaluate the performance of basic *TupleChain* and extended scheme. Our proposed *TupleChain* is demonstrated to be able to handle extremely high update frequency (1 million updates per second), super large data sets (10 million rule sets) and a large number of fields (100 fields). When the scale for each becomes large in the experiments, our scheme is the only survivor in all cases, while keeping the system throughput higher than 1 million packets per second all the time.

The rest of this paper is organized as follows. Section II reviews the literature work. Section III presents our motivation, core ideas and the basic scheme of *TupleChain*, which is followed by a comprehensive complexity analysis in Section IV. Section V introduces a series of technics to boost *TupleChain*'s practical performance. Then we evaluate *TupleChain* and some state-of-arts in Section VI Finally, Section VIII concludes the whole paper.

II. Related Work

We begin by reviewing two main categories of related work: 1) conventional packet classification techniques developed for static or infrequently changing rule sets; and 2) recent approaches designed to support dynamic updates and scalable processing in high-throughput data planes.

A. Conventional Scenarios

Hardware-based classifiers are widely adopted in industry. TCAM (Ternary Content Addressable Memory)-based solutions offer very fast speed, but their slow updates [18] restrict their use. With carefully designed structures and pipelines, FPGA (Field-Programmable Gate Array)-based solutions [19], [25] are faster and more flexible than TCAM.

Most algorithmic solutions target to the software scenario. In the early days, in addition to trie-based solutions [20], Cross-Producting [20] and Recursive Flow Classification [10] attracted lots of interestes for their fast speed. However, neither of them works well with large data sets. Current state-of-the-art solutions are based on decision tree [6], [7]. They achieve fast speed with heuristic strategies at the cost of slow updates, and their performance varies a lot across different data sets [11].

B. Dynamic Scenarios

Many classification algorithms only work with static sets of flows, or have expensive incremental update procedures, making them unsuitable for dynamic flow tables due to their slow updates. For a better trade-off between lookup and update, *PartitionSort* [9] divides rules into sortable rulesets, which supports both fast search and fast update by utilizing the binary search tree. On the other hand, the *Bloom Filter Intersection (BFI)* [23] follows the basic mode of *Bit Vector* [15], but represents the results on individual fields as bloom filters. It can achieve fast lookup with efficient updates. However, it can not scale well to the number of rules due to the fixed size of bloom filters.

Tuple space search (TSS) [21] is designed for packet classification but is well suited to flow lookup. Its core idea is to divide a large table into groups, where rules in the same group share the same mask for the fields to match. A flow lookup needs to search all the groups with a hash probe on each, and output the best result. This scheme is proposed with several extensions, among which the *pruned tuple search (PTS)* is the fastest. It processes individual fields and combines the results to pick up the candidate groups to search in the next step. In contrast, *tuple search using a balancing heuristic (TSBH)* focuses on reducing the complexity. By repeatedly selecting the best group to probe with a balancing heuristic, and skipping part of the remaining groups according to the search on the selected one, the number of required probes can be sharply reduced. Its lookup complexity is $\mathcal{O}(m^{\log_3 2})$, where m is the number of groups.

Open vSwitch [5] adopts the basic *TSS* scheme and improves its practical performance with a series of runtime pruning . We refer this scheme as *Priority Sorted Tuple*

Search (PSTS). *TupleMerge (TM)* [8] aims to reduce the number of rule groups at the construction time. Its core idea is to move the rules in some groups to others to restrict the collision rate below a threshold. *MultilayerTuple (MT)* [24] and *TupleTree* [26] inherit this merging approach. Both methods merge all tuples into several "big" tuples, which causes collisions. So they rearrange the rules at collision entry into a substructure, forming a "multilayer" or "tree" architecture. The difference between the two methods is the way of merging. *MT* adopts a static approach while *TupleTree* uses a heuristic one. *CutTSS* [16] exploits the joint use of decision tree and TSS, where it first divides the rule set into several groups and then uses TSS for the groups that contain many overlapping rules.

C. Summary of Prior Arts and Our Solutions

Because of its generality of fields, linear memory cost and constant update complexity, the *TSS* model has been proven to be a good starting point to develop a flow lookup scheme with multifaceted scalability. However, its performance may suffer when the number of groups is large, as it has to probe all groups. *PTS*, *TSPS*, *TM*, *MT*, *TupleTree* all aim at improving the practical performance, but none of them provides the worst-case performance guarantee since they have the same complexity as *TSS*. Although *TSBH* makes a great effort to lower the lookup complexity, its practical performance is not that good and its update is too complicated. *CutTSS* gains performance via cutting but its update and memory cost also deteriorates. In this work, we start from *TSS* as well, but propose a new scheme *TupleChain* to conquer the performance challenge. By exploiting the connections between rule groups and carefully trading off between the lookup speed and update speed, our approach achieves both fast lookup and fast update while guaranteeing the worst-case lookup performance and average update performance.

III. TupleChain: Binary Search on Chained Tuples

In this section, we first introduce the basic model and our motivation in Section III-A, then the concepts of tuple graph and lookup guidance information in Section III-B. We further present the basic scheme of *TupleChain*, as well as its algorithms for packet lookup and rule update in Section III-D.

A. Basic Model and Motivation

We denote a d-field rule r as $(\vec{f}, \vec{m}, pri)$, where the integer pri denotes the rule's priority, the d-dimensional vectors $\vec{f}$ and $\vec{m}$ represent its field to match and mask respectively. Before a flow lookup, the search key $\vec{k}$ with the corresponding d fields is generated based on the incoming packet p and some metadata. A packet p matches a rule r if and only if $\vec{k}$ & $\vec{m} = \vec{f}$. Following the basic tuple space search (*TSS*) model, a flow table is divided into *tuples* (as shown in Fig. 1 and Fig. 2), each is identified by a mask (a d-dimensional vector) and associated with a hash table (keyed by d-dimensional vectors). For simplicity, we refer to an entry of a tuple's hash table as the "tuple's entry".

TABLE I
Symbol Table

Symbol	Meaning
d	Number of matching fields (dimension)
r	A rule consisting of matching conditions
$\vec{f}$	field vector
$\vec{m}$	mask vector
pri	Rule priority, used to resolve multiple matches
$\vec{k}$	Lookup key vector extracted from the packet header
t	A tuple (rule group), defined by a common mask
e	An entry in a tuple's hash table
m	Total number of tuples
l	Number of tuple chains
m'	Number of tuples in the longest chain
n	Total number of rules
n'	Maximum number of rules in a single chain
$F(m,l)$	Number of tuples accessed during lookup

Notation. Unless otherwise specified, all notations used in Sections III–V follow the definitions listed in this table. This table serves as the standard reference for symbols throughout the remainder of this paper.

Rule	SRC/MASK	DST/MASK	PRI	ACT
r_1	0x00 / 0x**80**	0x80 / 0x**C0**	1	FWD 0
r_2	0x00 / 0x**C0**	0xC0 / 0x**F0**	2	FWD 1
r_3	0x80 / 0x**C0**	0xA0 / 0x**FC**	2	DROP
r_4	0x80 / 0x**F0**	0xA8 / 0x**FC**	2	FWD 2
r_5	0x20 / 0x**F0**	0x80 / 0x**80**	2	DROP
r_6	0x20 / 0x**F0**	0xA8 / 0x**FC**	3	FWD 2
r_7	0x80 / 0x**F8**	0xA0 / 0x**F0**	4	FWD 1
r_8	0xA8 / 0x**F8**	0xA0 / 0x**F0**	4	DROP

Fig. 1. simplified 2-field rules

Tuple	Masks	Rule(s)
t_1	(0x**80**, 0x**C0**)	r_1
t_2	(0x**C0**, 0x**F0**)	r_2
t_3	(0x**C0**, 0x**FC**)	r_3
t_4	(0x**F0**, 0x**80**)	r_5
t_5	(0x**F0**, 0x**FC**)	r_4, r_6
t_6	(0x**F8**, 0x**F0**)	r_7, r_8

Fig. 2. constructed tuples

With *TSS*, flow lookup is performed by searching all tuples and returning the one with the highest priority among all matched rules. Fig. 3 shows the *TSS* constructed with the rules shown in Fig. 1, where every tuple is denoted as a labelled cycle. When processing a packet, all 6 tuples are searched.

Our basic idea is to exploit the connections between tuples to avoid unnecessary searches. We propose the use of *TupleChain* to organize tuples into a set of chains, where two consecutive tuples on a chain have a unidirectional connection. Though all chains will be searched, the lookup on each chain can be well-organized to skip unnecessary searches. In the example of Fig. 4, 6 tuples form 2 chains to be searched for the incoming packet. The lookup on the first chain starts from t_4 and ends after searching t_6. When searching along the second chain, the miss of the first probe on t_3 directs the lookup to

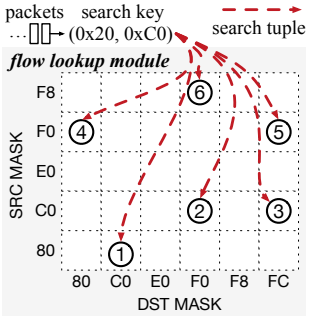

Fig. 3. lookup with *TSS*.

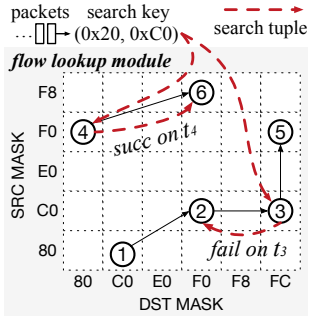

Fig. 4. lookup with *TupleChain*.

t_2. t_5 is skipped, because all its entries have left their *markers* on t_3 and the miss on t_3 indicates that the markers of t_5's entries also cannot be matched.

As the probe on t_2 succeeds, we can skip t_1, because all the rules from t_1 that could be matched by the incoming packet must have been reported as *hints* to t_2. Accordingly, the lookup on this chain is terminated. We will introduce the details of markers and hints in the Section III-B.

For this approach to work, we need to answer a set of questions: 1) How to set up and make use of the connections between tuples? 2) How to construct chains? 3) How to organize the search along each chain and how fast will the search be? How to perform rule updates without impacting the connections between tuples? We will answer these questions in the rest of this section.

B. Tuple Graph and Lookup Guidance

We first introduce the *tuple graph*, a directed acyclic graph that tracks the connections between any pair of tuples, as well as two types of information, *markers* and *hints*, to guide more efficient flow lookups with a tuple graph.

Given two tuples t_x and t_y, if t_x's mask is contained in t_y's on every field, we denote this relation as $t_x < t_y$. In Fig. 2, $t_1 < t_2$ because every field of t_1's mask (0x80, 0xC0), i.e.,(**1**0000000, **11**000000) in binary format, is contained in the corresponding field of t_2's mask (0xC0, 0xF0), i.e., (**11**000000, **1111**0000). The prefix length associated with the mask of t_1 is (1, 2) and the prefix length of t_2 is (2, 4). Obviously, rules in t_2 are more specific and cannot be matched if a search cannot match ones in t_1. Given a set of tuples, we construct a *tuple graph* by making every tuple a vertex, and adding an edge from t_x to t_y if $t_x < t_y$. On the tuple graph, the search of tuples is transformed into the traverse of vertexes.

To reduce the vertex thus tuple to visit in performing the flow lookup with the tuple graph, we leave and apply *markers* and *hints* along its edges backward and forward respectively. Given an edge from t_x to t_y, from any entry e_y of t_y we create an entry e_x and insert it into t_x, whose key is made by masking the key of e_y with the mask of t_x. This makes the key of e_x part that of e_y. We call e_x the *marker* of e_y and e_y the *owner* of e_x. As an entry, a marker can also hold a rule belonging to the tuple t it is inserted into. It can further leave markers in other tuples that have edges to t, one for each tuple. Besides, multiple entries from different tuples can share the same marker in a tuple as well. In the example of Fig. 6, along the edge from t_3 to t_5, an entry e_1 in t_5 can leave a marker in t_3. By masking its key (0x20, 0xA8) with t_3's mask (0xC0, 0xFC), a new key (0x00, 0xA8) is generated to associate with the entry e_2 in t_3. Initially, e_2 does not hold any rule that belongs to t_3 but can further leave its markers to t_2 (e_3) and t_1 (e_4) respectively. The key (0x20, 0xA8) in t_5 is more specific than the key (0x00, 0xA8) in t_3, so we can get an important property for markers: *If a packet succeeds in matching an entry in a tuple, it must be able to match all markers of this entry. On the contrary, if a packet fails to match an entry, it must be unable to match all owners of this entry.*

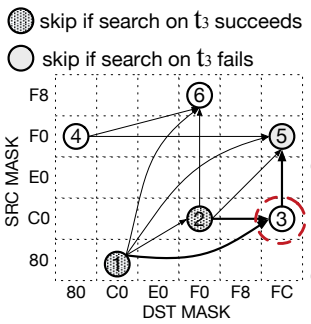

Fig. 5. a tuple graph

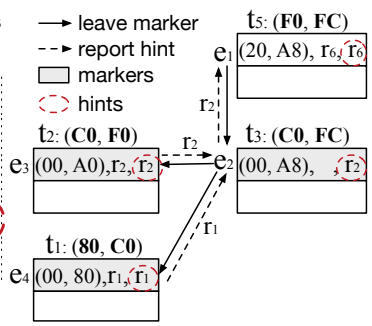

Fig. 6. part of lookup hints for t_3.

On the other hand, along an edge from t_x to t_y, any marker in t_x can report a *hint* to its owner in t_y to help cut short the search path. A hint of a marker is the best rule that can be matched by a packet with this marker and all of its own markers. More specifically, the rule held in an entry (if any) and all hints the entry received from its markers form the candidates to determine the entry's *best rule* for match, from which the one with the highest priority is selected. In Fig. 6, two markers e_3 and e_4 report their hints, r_2 and r_1 respectively, to their owner e_2. Since e_2 does not hold any rule in t_3, it selects r_2 (assuming r_2 has a higher priority than r_1) as its *best rule*, which is further reported to its owner e_1. However, e_1 holds a rule r_6 whose priority is higher than r_2, so e_1's *best rule* would stay as r_6. With the hint, we have another important property: *If a packet matches an entry in a tuple, there is no need to search and match against the tuples hosting its markers, since the best match with the corresponding rule has already been included by the hint from its markers.*

Based on these two types of guiding information, we can reduce unnecessary search. Given an edge from t_x to t_y on a tuple graph, for every entry in t_y, we leave its marker in t_x and update its *best rule*. We can make the following reductions. In the case that t_x is searched first but the probe fails, it's safe to skip t_y because all its entries have left their markers in t_x. A further search is not needed if a search with a coarser prefix fails. On the other hand, when t_y is checked first and it gives a match with the entry e, we can skip t_x as the only entry the packet can match from t_x must be e's marker, which has reported its hint to e.

Fig. 5 shows an example of utilizing the lookup hints in t_3 to avoid unnecessary searches. Once the search on t_3 succeeds, the searches in tuples t_1 and t_2 can be skipped, while a mismatch of t_3 allows us to skip the lookup in t_5.

C. Maintenance of Markers and Hints

To maintain markers and hints, we extend the design of the hash table entries. In our scheme, every entry e of tuple t's hash table is composed of a d-field *key* (as the identity), a *rule* belonging to t, a *hint* and an *owner list*. To save space and for the storage alignment, the *rule* and the *hint* are the pointers that point to corresponding rules, while the *owner list* is the pointer pointing to a linked list that stores the pointers pointing to all entries sharing e as a marker. An empty owner list indicates that the entry is not a marker.

An entry e of a tuple t will be created in two cases: 1) inserting a rule belonging to t or 2) inserting another entry's marker into t. In the first case, e's rule is set as the one being inserted while its hint and owner list are left empty. The owner list will be updated when an entry of another tuple leaves its marker to e of the tuple t at a later time. In the second case, e is created to host the marker of an entry from another tuple, with the entry inserted into its owner list. The rule and the hint of e are left empty initially, and the rule will be updated when a rule is inserted into e of t.

D. Tuple Chains and the Lookup Algorithm

Benefiting from lookup guidance, flow lookup with the tuple graph can be performed more efficiently. However, there are two drawbacks. First, once the search of a tuple is done, current lookup guidance can tell which tuples can be skipped, but not which one is the best to search in the next step. Actually, for different lookup requests there may be different optimal probing paths on the tuple graph. It's hard to pre-compute such optimal paths for future lookups. Besides, maintaining too much lookup guiding information will make rule updates extremely complicated.

To address these issues, we propose the construction of *TupleChain* where we break a tuple graph into disjoint chains that cover all tuples. This scheme brings in three benefits:

1) All tuples in a chain form a monotonic sequence with the "$<$" operator, enabling an efficient binary search.
2) Every rule update involves only a single chain, thus the update can be kept within this chain.
3) A tuple has at most one preceding tuple and one succeeding tuple. Thus, an entry has at most one marker, facilitating the marker maintenance. Although a marker can still be shared by multiple entries where it has to report hints, the overall cost across all tuples on the same chain can be amortized (see the proof in Section IV).

Essentially, all these chains form a disjoint path cover of the tuple graph. We propose an optimal method to form chains in Section V-A. Every chain is maintained as a red-black tree. For each tuple node, we rename its left and right children pointers as "fail" and "succ" respectively, and add two additional pointers "prev" and "next", to point to its preceding and succeeding tuples to facilitate the maintenance of markers and hints.

The flow lookup with *TupleChain* is simple. Every chain is searched by performing a tree traversal, and the output is the best result returned by searches from all chains. As described in ALGORITHM 1, the search on every chain starts from the tree root. In every step, the search is directed to the next node following the "succ" pointer or stops following the "fail" pointer, based on whether the current node yields a match or not.

E. Rule Updates with TupleChain

Here, we discuss how to update a rule with *TupleChain*. We first introduce rule insertion and rule deletion with high level logics, and then dive into details of dealing with markers and hints. We close this subsection with tuple insertion / deletion.

Algorithm 1: flow Lookup with *TupleChain*

Input: $packet$
Output: $bestRule$

```
bestRule = DEFAULT_RULE;
foreach chain do
    tp = chain.root;
    while tp do
        e = tp.table.search (packet.k⃗);
        CHECK_UPDATE_BEST_RULE(bestRule, e);
        tp = e ? tp.succ : tp.fail;
    end
end
```

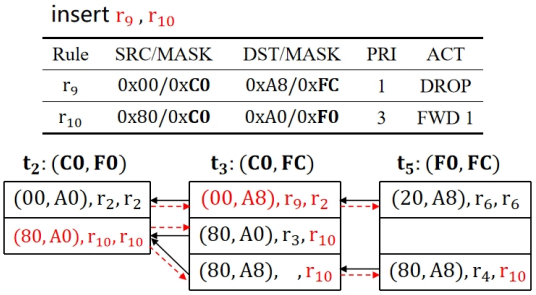

Fig. 7. example of rule insertion

1) Rule insertion: When inserting a rule r, we shall insert it into an entry e of its corresponding tuple t , and then build the marker link and update the hint of e. It starts by finding out tuple t which has the same mask with r. If entry e does not exist, it will be created and leave marker in the preceding tuple of chain. Then e's rule and hint are set to r. The marker of e returns its hint, which can be used to update e's hint. This updated hint will be further reported to e's owner(s). As shown in Figure 7, when inserting new rules r_9 and r_{10}, each rule locates its corresponding tuple and entry. For r_9, the corresponding entry receives the hint from its marker report, compares priorities, and then selects r_2 to continue forwarding. Since r_{10} has no marker, it directly reports its hint and keeps forwarding it.

Algorithm 2: insert a rule with *TupleChain*

Input: $rule$

```
t = find_or_insert (rule.m⃗);
e = t.table.insert (rule.f⃗); e.rule = e.hint = rule;
if k = leave_marker (e, t.prev) then
    e.hint = k.hint.pri > rule.pri ? k.hint : rule
end
report_hint (e);
```

2) Rule deletion: When deleting a rule r, we shall delete it from a corresponding entry e and update the hint of e. It starts by looking for e in a tuple. The deletion process will terminate if no entry e is found. Otherwise, the rule will be cleared from e. If e is not a marker, it will be deleted directly. If e is a marker, it will update its hint and report the change to its owner. If e's marker exists, e's hint will be set as its marker's hint. Otherwise e's hint will be cleared. As shown in Figure 8, when deleting rules r_3 and r_4, each rule locates

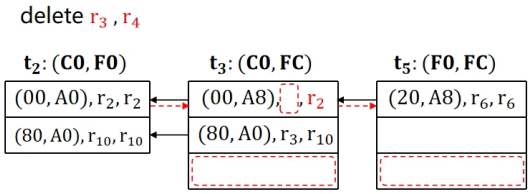

Fig. 8. example of rule deletion

its corresponding entry and is removed. For r_3, since its entry has an owner, its hint is replaced by its marker's hint and then forwarded. For r_4, since its entry has no owner, the entry itself is deleted and its empty marker is further deleted, continuing until reaching the entry containing r_{10}.

Algorithm 3: delete a rule with *TupleChain*

Input: $rule$

if (t = *find* ($rule.\vec{m}$)) ***AND*** ($e = t.table.search\ (rule.\vec{f})$) **then**
  if $e.rule.equals\ (rule)$ **then**
    if $e.owners$ *is empty* **then**
      delete_marker (e, $t.prev$);
      delete_tuple_if_empty(t.erase (e));
    end
    else
      k = obtain_marker (e, $t.prev$);
      $e.hint = k$? $k.hint$: NULL; $e.rule$ = NULL;
      report_hint (e);
    end
  end
end

3) Marker management: Marker is used for maintaining tuple chains. Its management is related to rule insertion and deletion. The process of maker creation is more involved. As described in ALGORITHM 4, the procedure of leaving a marker is recursively performed, until reaching an existing entry, or finding no preceding tuple any more. Finding or deleting a marker is simpler, where just one hash table operation is required.

Algorithm 4: leave the marker for an entry

Input: e /* leave the marker for this entry */
Input: t /* the target tuple of e's marker */
Output: k /* return this as e's marker */

if t ***AND*** ($k = t.table.find\ (e.\vec{f}\ \&\ t.\vec{m})$) *is NULL* **then**
  $k = t.table$.insert ($e.\vec{f}$ & $t.\vec{m}$);
  k' = leave_marker ($k, t.prev$);
  $k.hint = k'$? $k'.hint$: NULL;
end
$k.onwers$.add (e);

4) Hint management: Once a marker's hint is updated, the change must be reported to its owner(s). To avoid search through the whole tuple for the owners of a given marker, as introduced in Section III-B, we store all the owners of a marker in a list to trade the memory for the update speed. Fortunately, this is fairly cost-effective (the detailed analysis is presented in section IV-B).

TABLE II
COMPLEXITY COMPARISON

	lookup	memory	update	
			average	worst
TSPS	$\mathcal{O}(d \times m)$	$\mathcal{O}(n)$	$\mathcal{O}(d)$	
PTS	$\mathcal{O}(d \times m)$	$\mathcal{O}(d \times n)$	$\mathcal{O}(d \times n)$	
TM	$\mathcal{O}(d \times m'')$	$\mathcal{O}(n)$	$\mathcal{O}(d \times m'')$	
TSBH	$\mathcal{O}(d \times m^{\log_3 2})$	$\mathcal{O}(m \times n)$	$\mathcal{O}(m \times n)$	
TC	$\mathcal{O}(d \times l \times \log_2 \frac{m}{l})$	$\mathcal{O}(m \times n')$	$\mathcal{O}(d \times \frac{m}{l})$	$\mathcal{O}(m' \times n')$

[a] $m'' < m$ but an addtiional linear probe is required for searching a tuple.

5) Tuple insertion / deletion: Tuple insertion or deletion will be triggered when its first rule is inserted or all rules have been deleted. With a chain maintained as a red-black tree, inserting or deleting a tuple is fast as no hash computation is required. For tuple deletion, no additional operation is required other than removing the tuple from the chain. However, to insert a newly created tuple, we may have to probe all existing chains to determine which one it should be inserted into (some greedy strategies of selecting chains will be introduced in Section V-A), or create a new chain for it if no one can host it. A newly created tuple is empty at its insertion. After it is inserted into a chain, every entry of its succeeding tuple (if any) will leave their markers in this tuple, which may trigger recursive marker insertion in ALGORITHM 4.

6) Concurrent Updates: Since each update operation exclusively affects an independent chain, multiple updates targeting distinct chains can be executed concurrently. This parallelism can be enabled for system optimization by each chain maintaining its own fine-grained modification lock.

IV. COMPLEXITY ANALYSIS

In this section, we make a comprehensive theoretical analysis to understand how effective *TupleChain* will be and where its bottlenecks are. These analyses will serve as a guideline for us to improve its performance.

Suppose n d-field rules fall into m tuples, and the tuples form a tuple graph, which is then broken into l chains. Among these chains, the "biggest" one (which holds the largest number of rules) contains n' rules, and the "longest" one is made up of m' tuples. We analyze the performance of our *TupleChain* scheme accordingly.

Same as most TSS inspired schemes, the unit operation of flow lookup and rule updates with *TupleChain* is the hash with d-field keys. The cost of this operation linearly scales with the number of fields, so does the cost of storing d-field rules. We ignore the parameter d in the following analyses for simplification. The results of our evaluations are compared with other schemes in Table II.

A. Time Complexity of Flow Lookup

Flow lookup with *TupleChain* needs to search all l chains of m tuples, with a binary search on each chain. We denote

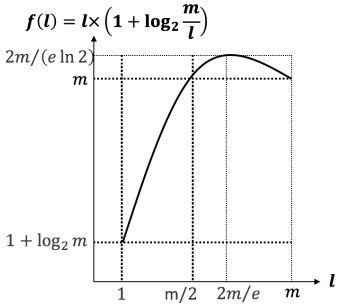

Fig. 9. A sketch of the function $f(l) = l \times (1 + \log_2(\frac{m}{l}))$.

the lookup cost as $F(m, l)$, which is the number of tuples that will be visited with a hash probe.

Theorem 1. *$F(m, l)$ has an upper bound $(l \times (1+\log_2(\frac{m}{l})))$.*

Proof. Suppose the i-th chain has m_i tuples, and its lookup cost is denoted as $F_i(m, l)$. Because of binary search, $F_i(m, l) \leq (1 + \lfloor \log_2 m_i \rfloor)$. For l chains in total, we have

$$\begin{aligned} F(m,l) = \sum_{i=1}^{l} F_i(m,l) &\leq \sum_{i=1}^{l} (1 + \lfloor \log_2 m_i \rfloor) \\ &\leq l + \sum_{i=1}^{l} \log_2 m_i \\ &= l + \log_2 \prod_{i=1}^{l} m_i \end{aligned} \tag{1}$$

According to the arithmetic-geometric average inequality,

$$\prod_{i=1}^{l} m_i \leq (\frac{\sum_{i=1}^{l} m_i}{l})^l = (\frac{m}{l})^l \tag{2}$$

Combining INEQUATION 1 and INEQUATION 2, we get $F(m, l) \leq l \times (1 + \log_2(\frac{m}{l}))$. □

To gain more insight into this upper bound, we treat m as a constant to analyze the function $f(l) = l \times (1 + \log_2(\frac{m}{l}))$, where $l \in [1, m]$. Its first-order and second-order derivatives are

$$\begin{aligned} f'(l) &= -\log_2 l + \log_2(\frac{2m}{e}) \\ f''(l) &= -\log_2 e \times l^{-1} \end{aligned}$$

$f''(l) < 0$ always holds, so $f'(l)$ is monotonically decreasing. Letting $f'(l) = 0$, we get $l = \frac{2m}{e}$. Accordingly, $f'(l) > 0$ holds when l increases from 1 to $\frac{2m}{e}$, so $f(l)$ monotonically increases from $f(1)$ to $f(\frac{2m}{e})$. While l continuously increases to m, $f'(l) < 0$ holds instead, and $f(l)$ monotonically decreases to $f(m)$. A sketch of $f(l)$ is shown in Fig. 9.

Noted from the curve, $f(l) = m$ has two roots across $[1, n]$. It is easy to verify that they are $l = \frac{m}{2}$ and $l = m$ respectively. Since $\frac{m}{2} < \frac{2m}{e}$, $f'(l) > 0$ holds across $l \in [1, \frac{m}{2}]$, namely $f(l)$ monotonically increases. Then, we have:

Corollary 1.1. *When $l < \frac{m}{2}$ holds,* TupleChain*'s lookup complexity is $\mathcal{O}(l \times \log_2 \frac{m}{l})$.*

Proof. $l < \frac{m}{2} \Longrightarrow \log_2 \frac{m}{l} > 1$, then, $F(m, l) < 2l \times \log_2 \frac{m}{l}$, thus $\mathcal{O}(F(m, l)) = \mathcal{O}(l \times \log_2 \frac{m}{l})$. □

Corollary 1.2. *Once the tuple graph can be broken into chains with each having fewer than half the number of tuples (i.e., $l < \frac{m}{2}$),* TupleChain *offers a lower lookup complexity than* TSS *($\mathcal{O}(m)$), and the fewer the number of chains, the lower the lookup complexity.*

B. Total Space Complexity

To evaluate the space complexity of *TupleChain*, we begin with the analysis on a single chain with n_c rules falling into m_c tuples. Its space complexity can be evaluated via the summation of the number of entries inside all tuples on the chain for two reasons. First, the entries of a tuple are stored and managed by a hash table, and the storage for all tuples' hash tables on a chain dominates the space cost. Second, the space consumed by each entry is also related to the number of entries. Every entry of a tuple is designed to have the same length, and is associated with a linked list that stores pointers to its owners. Since one entry owns one marker at most, any entry could be counted as an owner at most once. Accordingly, the total length of all owner lists at most equals to the total number of entries. Therefore, we only need to focus on the number of entries created.

First of all, every rule takes up an entry, and the process of leaving markers will create additional entries. Leaving the marker for a rule is recursively performed tuple by tuple, which may produce at most $m_c - 1$ entries. Therefore, there may be at most $n_c + n_c \times (m_c - 1) = n_c \times m_c$ entries. So the space complexity is $\mathcal{O}(n_c \times m_c)$. In *TupleChain*, every chain is independent, so for all chains in total, we have:

Theorem 2. *TupleChain's space complexity is $\mathcal{O}(n' \times m)$.*

C. Time Complexity of Rule Update

Tuple insertion/deletion are actually rarely triggered[1] and can be performed efficiently. Therefore, we do not count them for complexity analysis. We focus on the complexity of handling markers and hints when inserting / deleting a rule within an existing tuple, as the corresponding operations are the most time-consuming.

Inserting / deleting a rule with a tuple only affects a single chain that hosts this tuple. We denote the number of tuples on this chain and the number of rules belonging to the tuples on the chain as m_c and n_c respectively. Since any entry has one marker at most and leaving the marker for a entry is recursively performed tuple by tuple, at most $m_c - 1$ entries will be accessed or created. As obtaining or deleting a marker only requires one hash operation, the time complexity of marker maintenance turns to be $\mathcal{O}(m_c)$.

Now we evaluate the time complexity for reporting hints. By associating every entry with a separate owner list, we can locate all owners of an entry quickly without any hash operation. In addition, the hint reporting starting from an entry in tuple t_i is also performed tuple by tuple recursively, forming a *reporting tree* with every level of entries residing in a tuple $t_j (j > i)$ excluding the tree root which is in t_i. It is a tree

[1]the rates of tuple insertion/deletion we observed throughout our experiments were as low as 0.1%.

rather than a path because one entry can have multiple owners. For an entry in t_i, its reporting tree excluding this entry can be as large as covering all entries in all tuples $t_j (j > i)$. This determines that the worst case time complexity of hint reporting is $\mathcal{O}(n_c \times m_c)$.

In the average case, however, the cost of reporting hints is perfectly amortized. Suppose the tuple t_i ($i \in [1, m_c]$) has x_i entries, we evaluate the cost of reporting hints for all entries in this tuple. Since one entry has one marker at most, any two reporting trees rooted at two different entries in t_i would never intersect. Accordingly, the union of all reporting trees rooted at t_i will have $\sum_{j=i+1}^{m_c} x_j$ entries at most, which determines the cost of reporting hints for all entries in this tuple. Thus, the total cost across m_c tuples is summed up as:

$$\sum_{i=1}^{m_c-1} \sum_{j=i+1}^{m_c} x_j = \sum_{i=1}^{m_c} (i-1) \times x_i < m_c \times \sum_{i=1}^{m_c} x_i$$

Since this cost can be amortized by all $\sum_{i=1}^{m_c} x_i$ entries on this chain, the average-case time complexity for hint reporting turns to be $\mathcal{O}(m_c)$.

Any update will be performed within one of the chains in *TupleChain*, so we take n' and m' to calculate the overall update complexity.

Theorem 3. *TupleChain's update complexity is $\mathcal{O}(m')$ in the average case, and $\mathcal{O}(n' \times m')$ in the worst case.*

V. Boosting Practical Performance

The comprehensive theoretical analysis in the last section provides us with more insight into *TupleChain*, which enables us to refine the design to boost its practical performance.

A. Optimal Chain Construction

According to Corollary 1.2, to construct a *TupleChain* with the lowest lookup complexity, we should break the tuple graph into a minimal number of chains. This is essentially a classic problem in graph theory, known as the minimum path cover problem, which is NP-hard in general graphs [3]. However, for directed acyclic graphs (DAGs) such as the tuple graph, the problem can be efficiently transformed into a maximum bipartite matching problem. Specifically, each vertex in the original graph is split, and a bipartite graph is constructed in which we search for a maximum matching. According to graph theory, the minimum number of paths required equals the total number of vertices minus the size of the maximum matching. We employ the efficient *Hungarian algorithm* [13] to solve this matching problem and construct the optimal *TupleChain* accordingly. This approach ensures theoretical optimality while maintaining strong computational efficiency.

B. Greedy Strategies for Tuple Insertion

Once a new tuple is created, we first try to insert it into an existing chain whenever feasible to control the number of chains, which is the key factor to restrict the lookup cost (Corollary 1.2). In case that multiple chains can host this tuple, we chose the shortest one to control the length of the longest chain, as it determines the memory cost and update overhead (Theorem 2). Further, if there are multiple chains that can host this tuple, we choose the one with fewer rules to reduce the worst-case update overhead (Theorem 3).

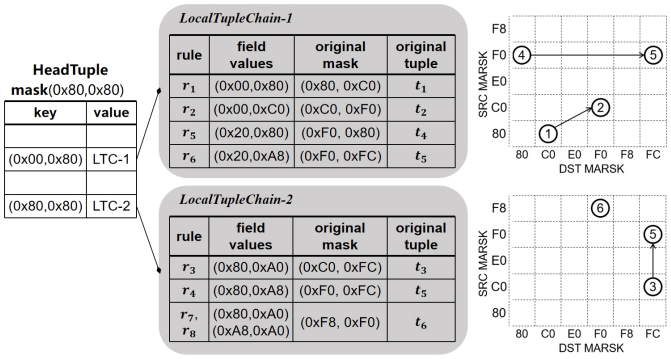

Fig. 10. An Extened Tuple Chain with one head tuple.

C. An Extension by Rule Grouping

Many researchers have observed that the number of tuples grows significantly when the number of rules become larger [8], [24], [26]. This will cause too many and too long chains in our scheme, resulting in the decrease of performance. In view of this, we propose a new data structure, *Extended TupleChain* (ETC in short), to boost the overall performance during practical running.

Our key idea is to reduce the access of tuples by merging chains into groups. Nearby chains in the tuple graph will be merged into one group. For each group, we set up a mask by taking the intersection of all masks from the chains to merge. We create a head tuple with this mask and insert all rules of these chains into this head tuple. With keys formed by applying the mask of the head tuple to the fields of rules, several rules may fall into the same entry. In Fig. 10, masked by $\vec{m}_h$ (0x80,0x80) of head tuple to form the key (0x00, 0x80), r_1 with (0x00, 0x80), r_2 with (0x00, 0xC0), r_5 with (0x20, 0x80) and r_6 with (0x20, 0xA8) are inserted into the same entry. These rules will be further constructed into a local TupleChain, similar to that of the *TupleChain*.

Compared to the original TupleChain, local TupleChains have much shorter lengths. This can boost the performance in all aspects. First of all, any rule update is processed within one "small" instance, so the update overhead can be sharply reduced. Although all head tuples must be probed for a lookup, only one "small" instance managed by every head tuple that yields a match will be checked intensively. Therefore, the lookup cost can also be significantly reduced, especially when there are only a few head tuples (which is the fact in practice according to our evaluations). At last, the total memory footprint of ETC can be reduced as well. The maintenance of additional markers that contains no rule dominates the storage of TupleChain, which can be greatly reduced with shorter chains in "smaller" *TupleChain*.

On top of the basic *TupleChain*(*TC*) design, we further propose the *Extended TupleChain* (*ETC*) structure to improve

practical performance in large-scale rule set scenarios. The core intuition behind *ETC* is that when the number of rules grows rapidly, the number of tuples also expands, leading to either too many chains or overly long single chains, which in turn affects lookup efficiency and update overhead. To address this, *ETC* attempts to aggregate adjacent chains on the tuple graph, merging them into a "group" and introduces a head tuple at the group's entry. All rules from the merged chains are inserted into the corresponding entry after being processed by the head tuple's mask, and then organized into a local TupleChain structure within that entry. In this way, multiple long chains are compressed into a more compact two-level structure: at the global level, only a limited number of head tuples need to be traversed, while at the local level only short local chains are handled. Intuitively, the benefit of *ETC* lies in significantly reducing the number of global chains to probe and simplifying the structure of each chain, thereby accelerating both lookup and update operations.

However, compared to the basic *TC*, *ETC* also introduces additional overhead. First, since the head tuple's mask is the intersection of the masks of multiple chains, rules from different chains may be mapped into the same entry, requiring the construction of a secondary local list or local *TupleChain* inside the entry to resolve conflicts. This increases the storage and maintenance complexity within entries. Second, during updates, inserting a new rule into the head tuple may trigger a reorganization of the local chain within that entry. Compared to the single-chain update in TC, this process is somewhat more complex. In other words, *ETC* trades off reduced global chain length and number for more complex structure maintenance within local entries, thus theoretically increasing both time and space overhead.

From an overall and practical perspective, the benefits of *ETC* far outweigh its overhead. This is because, in real-world rule distributions, the growth of tuple numbers is usually faster than the growth of complexity within a single tuple's entries. When the rule scale reaches millions or tens of millions, the number and length of chains in basic *TC* both increase significantly, leading to degraded lookup and update performance. By contrast, *ETC* compresses the global chain numbers significantly, so that lookups only need to access a small number of head tuples, while updates are mostly confined to small-scale local lists. In real network environments, due to the concentrated and localized nature of rule distributions, *ETC* can achieve substantial improvements in overall system throughput and memory efficiency while keeping local overhead acceptable. Therefore, *ETC* is an effective extension of the basic *TC*, balancing theoretical feasibility and practical efficiency, and demonstrating higher utility in large-scale dynamic flow table scenarios.

VI. Performance Evaluations

To comprehensively evaluate the performance and scalability of *TupleChain* and its extended scheme, *Extended TupleChain*, this chapter designs and conducts a series of experiments. The experiments cover rule sets of different scales, feature distributions, and numbers of fields, with the aim of verifying the proposed algorithms in terms of lookup speed, update efficiency, memory consumption, and scalability, and systematically comparing them with representative existing algorithms.

TABLE III
Number of tuples accessed in average

	1 kilo rules		1 million rules	
	number of tuples	other[a]	number of tuples	other[a]
TSS	80		404	
PTS	1	5.8	1.9	5.3
TSBH	28.2		83.4	
PSTS	55.8		334	
TM	1	1	8	45.6
MT	1	1	5.6	8.3
TupleTree	1	1	5.2	6.5
TC	12.1		23.1	
ETC	1		4.2	

[a]for *PTS*, it's the number of accessed trie nodes;
for *TM*, *MT*, *TupleTree*, it's the number of verifications;

A. Reduction of the Number of Tuples to Search

All *TSS*-based schemes attempt to reduce the number of tuples to search. We compare 9 schemes, *TSS*, *PTS*, *TSBH*, *PSTS*, *TupleMerge (TM)* [8], *MultiLayerTuple (MT)* [24], *TupleTree* [26], *TC*, and *ETC* using two datasets with 1 kilo and 1 million 2-field rules and corresponding traffic traces respectively. The average number of tuples accessed for one lookup is reported in Table III. *TSS* produced 80 and 404 tuples respectively, which are all searched in a lookup. Our results confirm the statement claimed in [21] that *PTS* has a promising practical performance. In this study, it requires fewer than 2 tuple searches on average. However, before tuple search, it needs to process each field with prefix trees, and combine the results via bitmap operations. On the other hand, *TM* reduces the number of tuples of two datasets from 80 to 1 and from 404 to 8 at the cost of additional verifications. *MT* and *TupleTree* also reduce the number of tuples and their additional verifications are smalller than *TM* in 1 million rules case.

Compared with *TSS* on these two datasets, our basic *TupleChain* scheme reduces the number of tuples to search by $\%84.9$ and $\%94.3$ respectively, which can be further improved by its extension *ETC*. *ETC* requires fewer tuple searches than all other approaches except *PTS*. Although *PTS* requires slightly fewer tuple searches than *ETC*, it brings in additional cost on tree traversals. It is clear that *ETC* is a better choice for practical implementation compared with *TC*, but *TC* guarantees the worst-case performance of *ETC*.

B. Evaluation Methodology

1) Algorithms used in Evaluation: We compare the performance of *ETC* with three state-of-the-art schemes for fast packet classification, *MT*, *TupleTree* and *CutTSS*, These methods respectively embody different design philosophies such as prefix-tree pruning, heuristic skipping, rule merging, and decision-tree partitioning, thereby covering the main directions of existing research. All implementations are evaluated on a

unified experimental platform, with efforts made to ensure the openness and fairness of the algorithmic implementations. The codebases are sourced from open repositories.[2]

2) Datasets used in Evaluation: For dataset selection, the experiments adopt two types of sources. First, we use synthetic rule sets and traffic data, including multiple datasets ranging from 1,000 to 10 million rules, with the number of fields varying from 2 to 100, in order to examine the scalability of the algorithms across both size and dimensionality. The rules are generated using a random mask generation method to ensure diversity in match distributions. Second, we employ rule sets and corresponding traffic traces generated by the ClassBench [22] benchmark tool, including access control list (acl) rules, firewall (fw) rules and iptable (ipc) rules, each comprising 1,000 groups of approximately 100,000 rules per group. These rule sets are more representative of real-world applications and thus help to assess the algorithms' applicability in practical scenarios.

3) Evaluation framework: With respect to the experimental environment, the evaluation platform adopts a multithreaded design in which the lookup module and update module run independently, simulating scenarios of concurrent high throughput and high update frequency. A traffic generator continuously injects packets, while an update manager generates and submits rule insertion and deletion requests. Together, they interact with the lookup module to ensure stable experimental execution under controlled conditions.

The experiments are organized into two levels according to their objectives: *performance evaluation* and *system evaluation*.

Performance evaluation focuses on quantitative assessment of single or combined metrics in a controlled environment, including:

- **Lookup performance**: measured in million packets per second (MPPS) to evaluate throughput under different numbers of rules and fields;
- **Update performance**: measured in million updates per second (MUPS) to assess the efficiency of insertion and deletion operations;
- **Memory overhead**: measuring storage requirements of different algorithms with large-scale rule sets;

Typical experiments in this category include lookup complexity tests, ClassBench benchmark rule set evaluations, and scalability comparisons in terms of rule scale, field scale, and update rate.

System evaluation examines the runtime behavior of the algorithms under conditions close to practical deployment. We set up a multithreaded environment consisting of a lookup module, update module, traffic generator, and update manager. The traffic generator continuously injects test packets, while the update manager issues rule insertion and deletion requests in parallel, simulating the dynamics of datacenter or edge networks. System evaluation focuses on throughput stability under concurrent lookup and update, real-time responsiveness under high-intensity traffic, and overall system capacity at different traffic rates. This category corresponds to the link-rate experiments and high-frequency update experiments in the original design, and reflects the advantages and bottlenecks of *ETC* compared with other approaches in realistic network scenarios.

Through this layered organization, *performance evaluation* reveals the theoretical potential of *TC*/*ETC* in individual performance metrics, while *system evaluation* verifies their practicality in integrated runtime environments. Combined, the two sets of results provide a complete chain of evidence for assessing the effectiveness and deployability of the proposed methods.

C. Performance Evaluation with multi-faced Scalability

1) Performance Scalability with Table Size: We evaluate each of the 4 schemes with 6 groups of 2-field rule sets of different sizes to measure its lookup speed, update speed and memory cost. *ETC* achieves the highest performance in all cases (as shown in Fig. 11 and Fig. 12), while *MT* has the lowest memory cost (Fig. 13). Overall, in comparison to *MT*, *ETC* achieves a speedup of $1.25 \sim 3.7$ at the cost of only $\%10 \sim \%30$ additional memory consumption, while maintaining competitive update performance. . Additionally, *ETC*'s throughput decreases the slowest as the scale increases. Only *ETC* can offer a throughput higher than 1 MPPS to process a data set with 10 million rules. We can see that the performance decreases and the memory cost increases sharply for *CutTSS*, which is due to the copy of rules in the decision tree algorithm.

2) Performance Scalability with Rule Characteristic: We evaluate 4 schemes with datasets sized around $100\ K$, where the number of fields ranges from 2 to 100. With the tester flushing packets at 10 MPPS, we measure the system throughput and memory cost for each scheme. As shown in Fig. 17 and Fig. 19, only *ETC* works in all cases. The others experience a sharp decline in throughput. The throughput of each drops below 0.01 MPPS once the number of fields exceeds 20. In contrast, *ETC* shows excellent scalability, with its throughput only decreasing from 6 MPPS to 1.1 MPPS. For the memory cost, others require more than $1\ GB$ of memory, and can not be constructed when there are more than 50 fields (the system runs out of memory). In contrast, *ETC* requires less than $70\ MB$ of memory to handle $100\ K$ 100-field rules. Among the four evaluated schemes sized around $100\ K$. *ETC* demonstrates dominant algorithmic advantages by achieving an optimal balance between high performance and low resource consumption. As shown in Fig. 14, Fig. 15 and Fig. 16, *ETC* consistently demonstrates overall optimal performance across all evaluation metrics. This exceptional scalability stems from its unique chain design, which enables efficient handling of complex rule sets while maintaining operational efficiency. The *MT* algorithm exhibits significant performance variations across rule sets with different characteristics, while the *TupleTree* algorithm demonstrates stable but mediocre performance. In contrast, the *CutTSS* algorithm consistently underperforms in both update speed and storage utilization.

[2]https://github.com/drjdaly/tuplemerge; https://gitee.com/dave_ta/TupleTree; https://github.com/zcy-ict/MultilayerTuple;http://www.wenjunli.com/CutTSS;

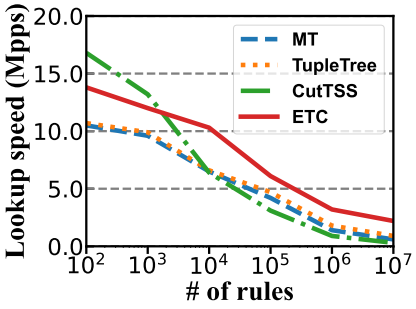

Fig. 11. lookup speed cost versus table size

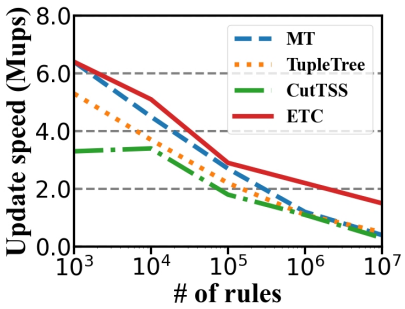

Fig. 12. update speed versus table size

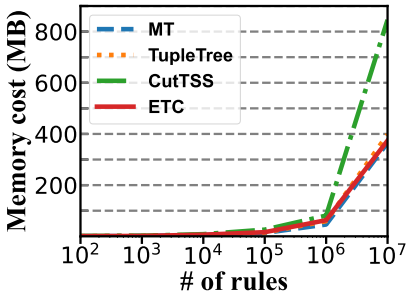

Fig. 13. memory cost versus table size

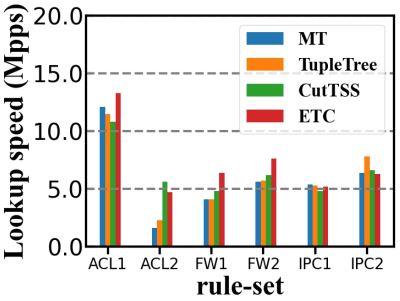

Fig. 14. lookup speed versus rule characteristic

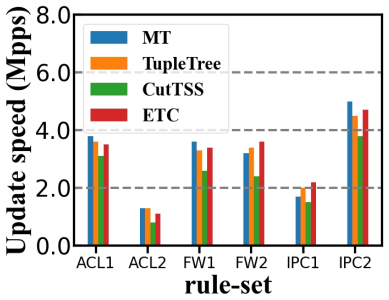

Fig. 15. update speed versus rule characteristic

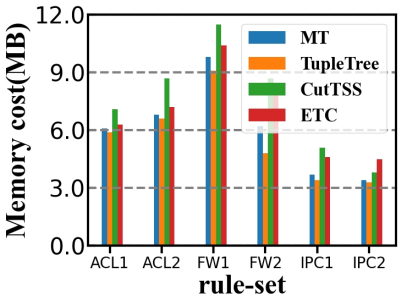

Fig. 16. memory cost versus rule characteristic

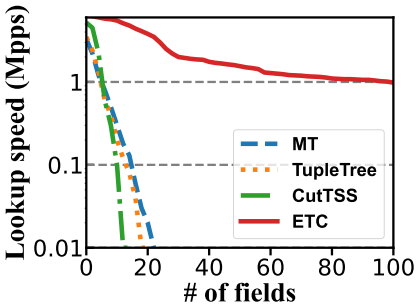

Fig. 17. lookup speed versus rule complexity

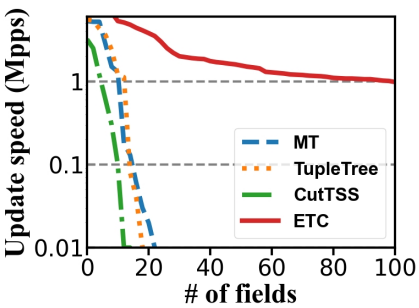

Fig. 18. update speed versus rule complexity

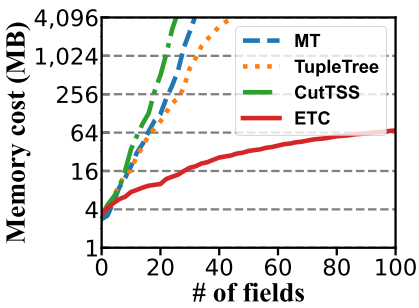

Fig. 19. memory cost versus rule complexity

3) Performance Scalability with Rule Complexity: We evaluate 4 schemes with datasets sized around 100 K, where the number of fields ranges from 2 to 100. With the tester flushing packets at 10 MPPS, we measure the lookup speed, update speed and memory cost for each scheme. As shown in Fig. 17, Fig. 18 and Fig. 19, only *ETC* works in all cases. The others experience a sharp decline in lookup speed, update speed. The speed drops below 0.01 MPPS once the number of fields exceeds 20. In contrast, *ETC* shows excellent scalability, with its throughput only decreasing from 6 MPPS to 1.1 MPPS. For the memory cost, others require more than 1 GB of memory, and can not be constructed when there are more than 50 fields (the system runs out of memory). In contrast, *ETC* requires less than 70 MB of memory to handle 100 K 100-field rules.

D. System Evaluation

1) Lookup Throughput: We compare the system throughput of 4 schemes with a dataset of 10 million 2-filed rules and with the tester flushing packets at increasing rates. Figure 20 shows the similar trend for each of them. As the transmission rate of the tester increases, its receiving rate increases linearly at the beginning and then reaches the peak around a particular rate. *ETC* can accommodate a maximum throughput of around 2 MPPS, with a speed up of $1.2 \sim 2$ compared to *MT*, *TupleTree* and *CutTSS*.

2) Update Rate: We evaluate the 4 schemes with a dataset of 10 million 2-field rules as well as 1 million insertion/deletion requests. The tester flushes packets at a fixed rate of 2 MPPS. As the update rates increases from 100 to 10 million per second, we measure the receiving rate at the tester. In Fig. 21, all schemes experience a decrease around 20% in system throughput, but *ETC* remains the fastest all the time, staying as fast as 1.4 MPPS.

VII. Discussion

The main innovation of *TupleChain* lies in its paradigm shift from tuple merging to relationship-aware chaining. By preserving the partial order among tuples and exploiting it through markers and hints, *TupleChain* transforms heuristic

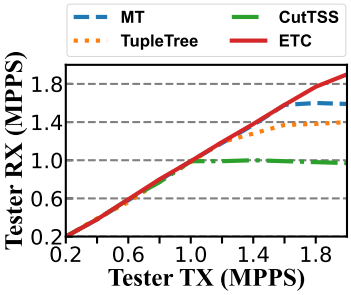

Fig. 20. lookup performance with increasing link rates.

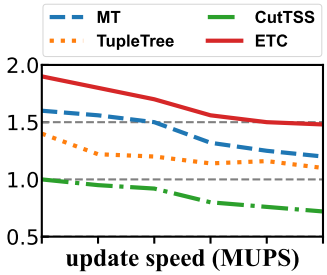

Fig. 21. lookup performance with frequent updates.

skip operations into provably correct guidance. This structure ensures that lookups can be organized as binary traversals along chains, while updates remain confined to a single chain. The dual mechanism of markers and hints further distinguishes *TupleChain*: markers enable systematic pruning when coarse tuples miss, while hints allow early stopping when finer tuples succeed. Together, they provide a rigorous basis for both search reduction and update localization, establishing theoretical guarantees that were absent in earlier schemes.

These advantages, however, come at the cost of additional overhead. The use of markers requires creating extra entries in coarser tuples, potentially expanding the number of stored items, while the maintenance of owner lists for hints adds to the memory footprint. During updates, markers may be recursively inserted into preceding tuples, and hints must be propagated through owner relationships to maintain correctness. Although these mechanisms introduce time and space overhead, they are effectively amortized within individual chains. In practice, the extra cost is outweighed by the benefit of sharply reduced lookup operations and strictly localized updates, particularly when the number of chains is significantly smaller than the total number of tuples.

From a hardware implementation perspective, *TupleChain* exhibits both opportunities and challenges. On FPGA platforms, its reliance on hash operations, pointer-like successor/failure transitions, and recursive but localized marker/hint maintenance aligns well with the strengths of parallel memory access and pipeline organization. The predictable structure of tuple chains makes it feasible to design lookup pipelines with controlled latency, while updates can be handled through staged or batched processing. The main strength of FPGA implementation is the potential for high throughput and fine-grained parallelism, but it requires careful engineering to manage marker proliferation and memory consumption. On the other hand, mapping *TupleChain* to P4-based programmable switches is conceptually possible by representing tuples as separate tables and expressing successor/failure branches through conditional control flow. Hints can be stored in metadata to enable early termination, while markers may be simulated through pre-inserted auxiliary entries. The advantage of a P4 realization is its integration into existing switch hardware and compatibility with standardized programming models. However, it is constrained by table size limits, action complexity, and the reliance on control plane involvement for recursive updates, which may increase update latency.

In summary, *TupleChain* introduces a fundamentally new mechanism for scalable flow table lookup, coupling structural innovation with provable complexity guarantees. Its additional overhead in space and time is a deliberate trade-off that secures efficient and predictable performance. Furthermore, preliminary analysis suggests that both FPGA and P4 offer feasible paths for hardware realization, with FPGAs providing stronger support for parallel, high-speed operation, and P4 offering portability and integration at the expense of update efficiency. *TupleChain* thus not only advances the algorithmic frontier but also demonstrates promising prospects for practical deployment in next-generation programmable network systems.

VIII. Conclusion

In this paper, *TupleChain* is proposed as a scalable flow lookup scheme that builds upon *TSS* algorithm but departs fundamentally from prior approaches that rely on heuristic merging of rule groups. Instead of coalescing multiple tuples into coarse structures, *TupleChain* explicitly constructs a directed acyclic graph (tuple graph) based on the containment relations of rule masks, and subsequently decomposes the graph into disjoint tuple chains. Within these chains, *TupleChain* introduces markers and hints as guiding information: markers propagate from more specific to coarser tuples to indicate miss-based skipping of successors, while hints propagate in the opposite direction to embed early-termination information into finer tuples. In this way, flow lookups become near-logarithmic in complexity, and rule updates are localized to the affected chain, thus avoiding global restructuring. Compared with traditional merging-based algorithms, *TupleChain* provides explicit performance guarantees on both lookup and updates, and achieves stability when scaling to millions of rules and hundreds of match fields.

Acknowledgment

We sincerely thank the anonymous reviewers for their insightful and constructive suggestions. This work was supported by the National Key R&D Program of China under Grant No.2022YFB3104800 and the National Natural Science Foundation of China under Grant No. 62072430. The corresponding authors of this paper are Yanbiao Li and Gaogang Xie.

REFERENCES

[1] Bgp routing table analysis reports. http://bgp.potaroo.net/bgprpts/bgp-active.png. Accessed: 2018-07-21.

[2] Break scalability barriers in openflow sdn. https://www.infoworld.com/article/3061152/networking/break-scalability-barriers-in-openflow-sdn.html. Accessed: 2018-07-21.

[3] Path cover. https://en.wikipedia.org/wiki/Path_cover.

[4] 5GAA Automotive Association. Technical requirements for edge-enabled v2x networks. White Paper v2.1 5GAA-WP-2024-021, 5G Automotive Association, 2024. p. 18.

[5] J. Pettit B. Pfaff, E. Jackson T. Koponen, J. Rajahalme A. Zhou, A. Wang J. Gross, P. Shelar J. Stringer, et al. The design and implementation of open vswitch. In *NSDI*, pages 117–130, 2015.

[6] G. Voskuilen B. Vamanan and T.N. Vijaykumar. Efficuts: Optimizing packet classification for memory and throughput. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 207–218. ACM, 2010.

[7] J. Daly and E. Torng. Bytecuts: Fast packet classification by interior bit extraction. In *INFOCOM, 2018 Proceedings IEEE*. IEEE.

[8] James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourtchenko. Tuplemerge: Fast software packet processing for online packet classification. *IEEE/ACM transactions on networking*, 27(4):1417–1431, 2019.

[9] S. Yingchareonthawornchai et.al. A sorted partitioning approach to high-speed and fast-update openflow classification. In *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[10] P. Gupta and N. McKeown. Packet classification on multiple fields. In *ACM SIGCOMM' 99*, pages 147–160. ACM, 1999.

[11] Peng He, Gaogang Xie, Kavé Salamatian, and Laurent Mathy. Meta-algorithms for software-based packet classification. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 308–319. IEEE, 2014.

[12] Chen Jin, Zhenyu Zhang, Xiaoxi Xiang, Shengqi Zou, Guoqiang Huang, Xiaowei Liu, and Xin Jin. Ditto: Efficient serverless analytics with elastic parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 406–419. ACM, September 2023.

[13] Harold W Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 2(1-2):83–97, 1955.

[14] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about sdn flow tables. In *Passive and Active Measurement*, pages 347–359. Springer, 2015.

[15] TV Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 203–214. ACM, 1998.

[16] Wenjun Li, Tong Yang, Ori Rottenstreich, Xianfeng Li, Gaogang Xie, Hui Li, Balajee Vamanan, Dagang Li, and Huiping Lin. Tuple space assisted packet classification with high performance on both search and update. *IEEE Journal on Selected Areas in Communications*, 38(7):1555–1569, 2020.

[17] F. Baboescu S. Singh and J. Wang G. Varghese. Packet classification using multidimensional cutting. In *ACM SIGCOMM '03*, pages 213–224. ACM, 2003.

[18] Devavrat Shah and Pankaj Gupta. Fast updating algorithms for tcam. *IEEE Micro*, 21(1):36–47, 2001.

[19] Haoyu Song and John W Lockwood. Efficient packet classification for network intrusion detection using fpga. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 238–245. ACM, 2005.

[20] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *ACM SIGCOMM '98*, pages 191–202, New York, NY, USA, 1998. ACM.

[21] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 135–146. ACM, 1999.

[22] David E Taylor and Jonathan S Turner. Classbench: A packet classification benchmark. *IEEE/ACM Transactions on Networking (TON)*, 15(3):499–511, 2007.

[23] Tong Yang, Alex X Liu, Yulong Shen, Qiaobin Fu, Dagang Li, and Xiaoming Li. Fast openflow table lookup with fast update. In *INFOCOM, 2018 Proceedings IEEE*. IEEE.

[24] Chunyang Zhang and Gaogang Xie. Multilayertuple: A general, scalable and high-performance packet classification algorithm for software defined network system. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2021.

[25] Jincheng Zhong and Shuhui Chen. Efficient multi-category packet classification using tcam. *Computer Communications*, 169:1–10, 2021.

[26] Jincheng Zhong, Ziling Wei, Shuang Zhao, and Shuhui Chen. Tupletree: A high-performance packet classification algorithm supporting fast rule-set updates. *IEEE/ACM Transactions on Networking*, 2022.