OpenFlow Accelerator: a Decomposition-based Hashing Approach for Flow Processing

Hai Sun
Washington State University
Email: hsun1@eecs.wsu.edu

Yan Sun
Huawei US Research Center
Email: yan.yansun@huawei.com

Victor C. Valgenti
Petabi, Inc.
Email: vvalgenti@petabi.com

Min Sik Kim
Petabi, Inc.
Email: msk@petabi.com

***Abstract*—To support scalable, flexible software-defined networking, OpenFlow is designed to provide granular traffic control across multiple vendor's network devices for efficient flow processing. Decision-tree packet classification algorithms do not scale to the number of flow table fields while decomposition algorithms such as RFC fail to provide necessary incremental update and determinism.**

Since searching in a single field is well studied, e.g. Longest Prefix Match (LPM) for prefix fields, we propose a decomposition approach which performs individual search on each flow table field, aggregates these results and conduct a query in a single hash table. Our approach scales well to the number of fields and allows incremental update. Meanwhile deterministic query is enabled for high-speed search. As far as we know our proposal is the first efficient decomposition approach to address multidimensional match in an OpenFlow flow table with an arbitrary number of fields as well as any match type. Theoretical analysis and experiments using synthetic classifiers justify the performance improvement.

I. Introduction

Flow processing requires the creation of sessions for individual flows. A flow is a stream of relevant packets that meet collective matching criteria and share the same characteristics. There is an increasing interest in designing high-performance network algorithms, frameworks or devices to perform flow processing. Applications such as stateful access control, deep inspection and flow-based load balancing require efficient flow processing. Fundamental network operations inherent to such applications include packet classification in flow-level, flow state management for stateful analysis, and per-flow packet order-preserving for traditional switch architecture. Meanwhile maintaining the network state on all flows passing through a system is of great significance for Software-Defined Networking (SDN) [4].

SDN provides an abstraction of network devices and operations [7]. OpenFlow [6] is the first standard interface designed specifically for SDN, providing high-performance, granular traffic control across multiple vendor's network devices [4]. OpenFlow adopts the concept of flows to identify network traffic in terms of pre-defined match rules programmed by the SDN control software. An OpenFlow flow table aggregates a collection of flows and an OpenFlow switch contains one or more flow tables. Each flow table entry (rule) consists of match fields, counters, and a set of instructions (actions) to apply to matching packets [1]. A pipeline in an OpenFlow switch groups a set of flow tables to enable flow processing like packet matching, forwarding, and modification.

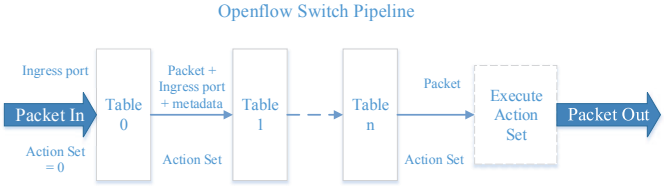

Fig. 1. Flow Table Pipeline

Fig. 1 outlines how flow processing is performed in an OpenFlow switch. Given a packet the beginning flow table must be searched. The subsequent flow tables are searched only when necessary. If a packet is matched by multiple entries, the entry with highest priority is chosen. If a matching entry is found, the actions associated with the specific flow entry are executed [1]. Besides the traditional layer-3 and layer-4 packet header fields, a total of 42 header fields are defined in the OpenFlow specification [1]. In the future more fields are expected to be defined and range representation may be available as range matching is widely used in high-speed networking applications.

TABLE I
A sample flow table

Rule	SA	DA	SP	DP	Prot
R1	161.21.12.0/25	25.1.71.64/26	8080	*	TCP
R2	161.21.12.0/24	25.1.71.64/24	25	3128	TCP
R3	161.21.0.0/16	33.21.79.32/26	*	21	UDP
R4	128.7.112.24/32	17.9.81.0/24	17	gt 1023	UDP

TABLE I shows a sample flow table with four rules and five header fields. SA, DA, SP, DP and $Prot$ indicate source/destination IP address, source/destination port and protocol type respectively. Usually, a rule's priority is implied by its position in the flow table: the earlier it appears, the higher its priority is. Counters and actions are not shown. The OpenFlow specification [1] defines three match types: exact match, prefix match and wildcard match. If range representation, e.g. DP value $gt1023$ of $R4$, is defined in the future OpenFlow specification, there will be four match types. Actually a wildcard is a special range and ranges can be

converted into prefixes. Although current packet classification algorithms can be used to process a flow table whose match fields are exactly the classic quintuple ACL criteria as shown in TABLE I, they suffer from scalability, incremental update and nondeterminism if a flow table contains much more fields.

- Scalability. OpenFlow architectures will face scalability problems in the future, especially in core networks due to the need for extensive number of states related to the active flows [10]. Although some packet classification algorithms scale well to IP length (32-bit IPv4 and 128-bit IPv6) and flow table size, they do not scale to the number of fields. For example in decision-tree algorithms such as HiCuts [6], HyperCuts [13] or Efficuts [17], more fields result in more dimension cutting and longer tree depth. Accordingly more costly memory accesses degrade the query performance. In the meantime a Ternary Content Addressable Memroy (TCAM) device does not scale as well since TCAM entry width is restricted owing to production cost. Only decomposition algorithms such as Recursive Flow Classification (RFC) [5] and Aggregate Bit Vector (ABV) [3] scale to the number of fields.
- Incremental Update. A sophisticated controller dynamically inserts and removes flow table entries on behalf of multiple independent experiments conducted by researchers with different accounts and permission [11]. Primary decomposition algorithms such as RFC are incapable of incremental update.
- Determinism. Highly deterministic performance is particularly desired for any substantial flow processing solution to be implemented and deployed in high-performance, special-purpose networking hardware devices such as Network Processing Unit (NPU). However current decomposition algorithms are heuristic based and nondeterministic. In a NPU device multiple threads are to be coordinated to accelerate processing and the slowest thread determines the overall speed for a stack of classification tasks. Therefore non-deterministic query degrades the system performance sharply.

Given the fact that single-dimensional search is well studied and numerous solutions for exact-match and prefix fields have been proposed, decomposing a multidimensional search into several instances of single-dimensional search is highly desirable due to parallelism supported by NPU. The primary challenge is to efficiently aggregate and combine the intermediate results of single-dimensional search and perform the final search in an efficient data structure. Our approach addresses the challenge by merging the intermediate results to a lookup key used to query a hash table. Our approach scales well to the number of fields and allows incremental update. Meanwhile deterministic query is enabled for high-speed search. As far as we know our proposal is the first efficient decomposition approach to address multidimensional match in an OpenFlow flow table with an arbitrary number of fields as well as any match type.

II. Related Work

Flow processing in an OpenFlow switch is inherently hard from a theoretical standpoint as multidimensional match requires either $O(\log_{d-1} N)$ time and linear space, or $\log N$ time and $O(Nd)$ space, where N is the number of rules and d is the number of fields. Few packet classification algorithms solve multidimensional match with an arbitrary number of fields and various match types.

Fundamental flow processing performance metrics are time, space, update and scalability. Processing time is measured primarily by the number of off-chip memory access. Space usage is evaluated according to the storage size of processing data structures in main memory or fast on-chip caches. Update is crucial as an OpenFlow switch must allow incremental addition or removal of flow table entries. Any substantial solution must scale to flow table size and the number of fields. Other significant metrics are programmability and determinism because of the wide use of Application-Specific Integrated Circuits (ASIC) and NPU in network processing. These special-purpose hardware devices make a good balance between performance and programmability to implement and deploy software algorithms. Particularly NPU open a new venture to explore thread-level parallelism to address the performance bottleneck of flow classification.

Current well-known packet classification algorithms can be implemented on hardware or software. Primary hardware solutions include TCAM, ASIC or NPU devices. Software algorithms can be implemented and deployed in ASIC and NPU devices to accelerate query performance since these devices are optimized for network processing, e.g. with built-in hash units. A TCAM device provides $O(1)$, deterministic query performance but requires too much power and a large board area if a flow table contains a large number of rules. In addition, TCAM chips result in high cost in production and upgrade, and thus have limited capacity and entry width which do not scale to the number of fields.

Software algorithms are usually based on decision tree or decomposition. Decision-tree algorithms, e.g. HiCuts , HyperCuts and Efficuts, partition the multidimensional search space by means of certain heuristics and thus are difficult to implement in hardware. Each query results in a leaf node in a decision tree which stores a small number of rules that can be searched sequentially to find the best match. The long tree depth due to more cutting dimensions gives rise to more costly memory access to find a specific rule. With more flow table fields the situation becomes even worse. Meanwhile determinism cannot be achieved with these solutions.

Decomposition algorithms such as RFC, ABV, or Bloom Based Packet Classification (B2PC) [12] scale well to the number of fields. In general the decomposition approach provides high throughput as a multidimensional search can be decomposed into a set of single-dimensional searches that can be performed in parallel. However crossproducting, the aggregation of intermediate results of single-dimensional searches, usually leads to large memory consumption. If the

problem is not well solved, incremental update may not be supported. RFC and other decomposition algorithms suffer from incremental update because they only work on static rule sets. Anytime a new rule needs to be inserted or an old rule needs to be removed, the data structures of these algorithms have to be rebuilt from scratch. In addition, they are heuristic-based and thus lack necessary determinism.

Our approach makes a good balance among these performance metrics. For high throughput and scalability to the number of fields our approach divides a multidimensional search into multiple of single-dimensional searches. For deterministic query our approach establishes a hash table to contain crossproducting data aggregated from intermediate results and allow $O(1)$ hash access. To achieve incremental update our approach trades memory for flexibility in inserting or removing rules. We will see that even this tradeoff in memory is not a serious drawback under the OpenFlow application context.

III. Multidimensional Flow Processing

3.1 Problem Statement

Definition 1 Flow Table. A flow table FT contains N number of rules. In the remaining paper we suppose that a rule r has d number of fields and each field F_k ($1 \leq k \leq d$) specifies one of three match types: exact match, prefix match or range match. Wildcard match is a special form of range match. F_i ($1 \leq i \leq e$) and F_j ($e+1 \leq j \leq d$ represent an exact-match field and a prefix field respectively where e is the number of exact-match fields.

Definition 2 Exact Match, Prefix Match and Range Match. An exact match specification is a value specified for a rule field F_k. Given a packet with header H, a packet field H_k ($1 \leq k \leq d$) is an exact match for the rule field F_k if and only if $v(H_k) = v(F_k)$ where $v(H_k)$ and $v(F_k)$ are the values in H_k and F_k respectively. A prefix match specification is a prefix specified for a rule field F_k. A packet field H_k is a prefix match for the rule field F_k if and only if the leading length (F_k) bits of $v(H_k)$ are the same as $v(F_k)$ [4]. A range match specification is a range of values $v(F_k) = [l_k, h_k]$ ($0 \leq l_k < h_k \leq 2^W - 1$) where W denotes port width, e.g. 16 bits, and W-bit numbers l_k and h_k are low and high bounds of the range. A packet field H_k is a range match for the rule field F_k if and only if $l_k \leq v(H_k) \leq h_k$.

Definition 3 Multidimensional Flow Processing. Given a packet with header H and a flow table FT with N number of rules, if both have the same d number of header fields, multidimensional flow processing aims to find the highest-priority rule r in FT if and only if each field value $v(H_k)$ matches the corresponding field value $v(F_k)$ in r.

Definition 4 Exact-Match Hash Table. Given a flow table FT, a hash table HT_{F_i} is established for each exact-match field F_i where e is the number of exact-match fields. A HT_{F_i} entry is a key-value pair $< h(v(F_i)), ID_{F_i} : counter >$ where $h()$ is HT_{F_i}'s hash function. $h(v(F_i))$ is the key to the field value $v(F_i)$ and ID_{F_i} is a field ID, an integer value for the field F_i. $counter$ is an integer counter for the field value $v(F_i)$ in the HT_{F_i} entry.

Definition 5 Prefix ID Map. Given a flow table FT, an ID map M_{F_j} is established for a prefix field F_j. A M_{F_j} entry is a key-value pair $< v(F_j), Q(v(F_j)) >$. $v(F_j)$ is the value of prefix field F_j in r. $Q(v(F_j))$ consists of four components: (1) a field ID ID_{F_j}, an integer value for the field F_j; (2) $counter$, an integer counter for the field value $v(F_j)$; (3) a list $PL(ID_{F_j})$ (briefly PL) that contains all the field IDs whose related prefixes belong to rules with higher priority in FT than current rule r and overlap with current prefix value $v(F_j)$; (4) a list $RL(ID_{F_j})$ (briefly RL) that contains all the field IDs whose prefixes belong to rules with lower priority in FT than current rule r and overlap with current prefix $v(F_j)$. For each field F_j a corresponding data structure LPM_{F_j} is constructed by a chosen LPM method in our approach.

Definition 6 Cross Concatenation. Given a rule r with d number of fields including e number of exact-match fields (F_1 to F_e) and $d-e$ number of prefix fields (F_{e+1} to F_d), the field value $v(F_i)$ of each exact-match field F_i is identified by ID_{F_i} and the field value $v(F_j)$ of a prefix field F_j is identified by an ID set IDS_{F_j} that contains $v(F_j)$'s ID ID_{F_j} and all the IDs in $v(F_j)$'s PL. The cross concatenation of r is defined as a procedure of cross concatenating all ID_{F_i} and IDS_{F_j} to produce a set of bit strings CC_r. Suppose each IDS_{F_j} contains n_{F_j} number of prefix IDs. The number of bit strings in CC_r is $\prod_{j=e+1}^{d} n_{F_j}$.

Cross concatenation is performed in three steps. Firstly retrieve each field's ID_{F_i} from exact-match field F_i and IDS_{F_j} from prefix field F_j; next do bit mapping. For a given ID ID_{F_k} we set a l_k-bit string such that $2^{l_k} > m_{F_k}$ where m_{F_k} is the number of unique ID values in field F_k. Bit mapping intends to convert ID_{F_k} to a bit string with l_k bits; eventually we define a L-bit string such that $\sum_{k=1}^{d} l_k = L$. Each CC_r element is a L-bit string generated by cross concatenating all bit-mapped ID_{F_i} and IDS_{F_j}.

Definition 7 Rule Hash Table. Given a flow table FT, a unique hash table RHT in our approach is established. Each entry is a key-value pair $< K, V >$. $h()$ is RHT's hash function. K is generated by applying $h()$ upon any CC_r element for a rule r in FT. V consists of three parts: (1) the highest-priority rule number TP; (2) the rule's associated action; (3) a list LP with a set of rule numbers with lower priority than TP.

3.2 Overview

Preprocessing a given flow table is required in our approach to construct necessary data structures. Given a query request, single-dimensional searching is conducted upon each field in parallel. For an exact-match field its corresponding hash table is queried. For a prefix field its LPM data structure is queried. Next the intermediate query results are aggregated into a lookup key to RHT and eventually the best rule is obtained by querying RHT.

Fig. 2 illustrates the preprocessing procedure. Given a rule r with d number of fields including e number of exact-match

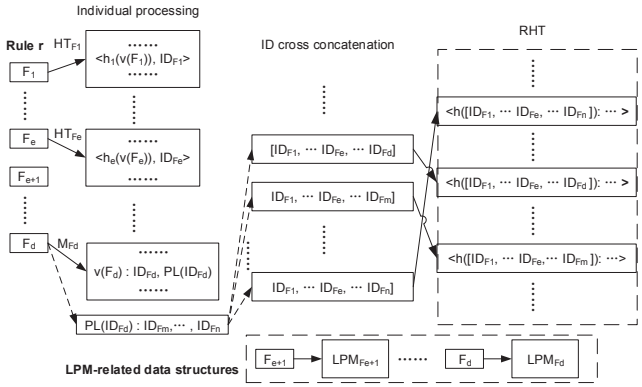

Fig. 2. Preprocess

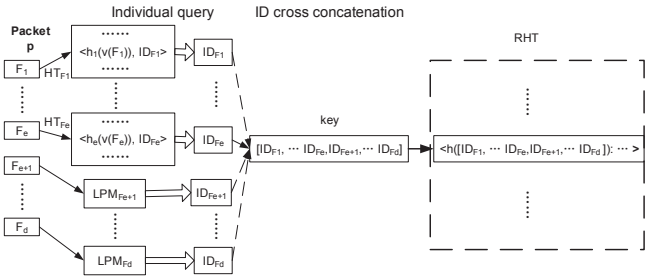

Fig. 3. Query

fields, preprocessing is performed in two steps. Firstly a hash table HT_{F_i} is established for each exact-match field F_i. We do not show $counter$ in F_i entry. Meanwhile an ID map M_{F_j} and LPM_{F_j} are built for each prefix field F_j. For example $v(F_d)$, F_d's field value, is mapped to an M_{F_d} entry with ID ID_{F_d} and PL. We do not show $counter$ and RL. PL may contain multiple IDs from ID_{F_m} to ID_{F_n}. Then the single-dimensional processing is done and we begin to construct RHT by iterating each rule r in FT and performing cross concatenation to produce a pair of key and value using CC_r for each r. Wildcards and range values in some fields of r can be converted to prefixes.

Querying a given packet header H is performed in exact-match hash tables and LPM data structures. Next all the intermediate results are bit mapped and concatenated into a lookup key to RHT. Fig. 3 demonstrates the query procedure. For each exact-match field F_i, lookup in HT_{F_i} using the corresponding field value $v(H_i)$ returns an ID value. We do not show $counter$ in HT_{F_i}. For each prefix field F_j LPM_{F_j} is searched using $v(H_j)$ and the ID of the longest matching prefix is found. Lookup in all the HT_{F_i} LPM_{F_j} can be performed in parallel. Next these IDs are bit mapped and concatenated as a key to RHT.

Any efficient LPM method is desired, e.g. Hierarchical Hashing [14], Binary Search On prefix Length (BSOL) [18] or Tree Bitmap (TBMP) [2]. In our approach simple hash functions such as CRC work well provided that they restrict collision with negligible computation and lookup overhead. To tackle the hash collision we have two options: (1) a chaining mechanism for convenience of implementation; (2) an open-address mechanism such as Peacock Hashing [9] or highly deterministic hashing [15]. Usually only one memory access is expected to query RHT or HT_{F_i} and high determinism can hence be achieved.

We propose two possible techniques to solve range (wildcard) match problem.

- Expand a wildcard or range into exact or prefix values. For a wildcard if the total number s of unique exact values in some field is limited, the rule containing the wildcard can be represented by s number of equivalent rules with each exact value in the field. A range can be similarly represented by an equivalent set of non-overlapping prefixes. The rule expansion is suitable only if s or the size of a prefix set expanded from a range is not large such that the expansion does not generate rule explosion.
- Separate the rules with wildcard or range into another flow table. On this table we may use other flow processing algorithms such as HyperCuts. An OpenFlow switch supports the divide-and-conquer methodology since flow table partitioning is allowed to simply pipeline manipulation.

These two techniques can be used together. For a flow table with multiple fields containing wildcard or range representation, we may expand some rules with few wildcards or range representation and separate other rules. In the evaluation we will see the influence of partitioning technique.

3.3 Preprocessing

HT_{F_i} and M_{F_j} are constructed respectively for each exact-match field F_i and each prefix field F_j. In addition, LPM_{F_j} is established for F_j. M_{F_j} is only used for preprocessing and update by means of PL and RL. PL stores all the field IDs whose prefixes belong to rules with higher priority than current rule and overlap with current prefix. RL contains all the field IDs whose prefixes belong to rules with lower priority than current rule and overlaps with current prefix. PL and RL are used together to update a rule. Algorithm 1 formulates the algorithm to build M_{F_j}.

Algorithm 2 formulates the algorithm to process a rule r into RHT. We set l_k of field value $v(F_k)$ as $\lceil \log_2 m_{v(F_k)} \rceil + 1$ ($m_{v(F_k)}$ is the number of unique values in field F_k. $m_{v(F_k)}$ is configured by the update requirement for a given flow table. Usually we set a larger value to allow sufficient ID values when new rules are added. Once cross concatenation results in CC_r, RHT's hash function is applied upon each CC_r element to generate the corresponding key. If the key does not exist in RHT, a new RHT entry is created and current rule number is set to be the highest-priority value in TP. Otherwise current rule number is inserted into the entry's LP since it has a lower priority than current entry's TP. CrossConcat() is the function to conduct cross concatenation.

Next we use the sample flow table in TABLE I to highlight each preprocessing step. Source/destination port fields are not

Algorithm 1 ID Map Generation Algorithm

FuncProcessPrefix(S) {S is an array of all unique prefixes on field F_j and t is its size}
Initialize an empty ID map M_{F_j};
$idCnt$ = 0; {ID generator for M_{F_j}}
for all p from 1 to t **do**
  if M_{F_j}.containKey($S[p]$) **then**
    increCounter(M_{F_j}.get($S[p]$)); {increment counter}
  else
    $node$ = M_{F_j}.createEntry(); {create a new entry}
    $node$.key = $S[p]$; {set its key}
    $node$.ID = ++$idCnt$; {set its ID}
    $node$.counter = 1;
    for all each prefix q in M_{F_j} **do**
      if isOverLap(q,$S[p]$) == true **then**
        $curNode$ = M_{F_j}.get(q);
        $node$.insert(PL,$curNode$.getID()); {PL: $PL(ID_{F_j})$}
        $curNode$.insert(RL,$idCnt$); {RL: $RL(ID_{F_j})$}
return M_{F_j}

Algorithm 2 Rule Hashing Algorithm

FuncProcessHashing(rNo,RHT,$vExt$,$vPfx$,$rVals$,h)
{rNo: rule No.; $vExt$: a vector of all HT_{F_i}; $vPfx$: a vector of all M_{F_j}; $rVals$: an array of $v(F_k)$; $h()$: RHT's hash function}
Initialize an empty bit string em; {em stores IDs queried from HT_{F_i}}
for all i from 1 to e **do**
  em.append($vExt[i]$.get($rVals[i]$));
Initialize an empty vector array $vCrs$; {$vCrs$ stores overlapping prefixes in M_{F_j}}
for all j from $e+1$ to d **do**
  create a new vector $vTmp$;
  $vTmp$.add($vPfx[rVals[j]]$.ID);
  $vTmp$.addAll($vPfx[rVals[j]].PL$);
  $vCrs$.add($vTmp$);
$bSet$ = CrossConcat(em,$vCrs$); {$bSet$: result vector of cross concatenation}
for all q from 1 to $bSet$.size() **do**
  key = $h(bSet[q])$;
  if RHT.containKey(key) **then**
    RHT.getVal(key).add(rNo,LP);
  else
    $node$ = RHT.createEntry();
    $node$.key = key;
    $node.TP$ = $r.No$;
    $node.LP$ = an empty list;

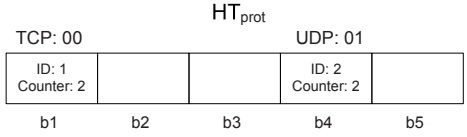

Fig. 4. Processing Protocol Field

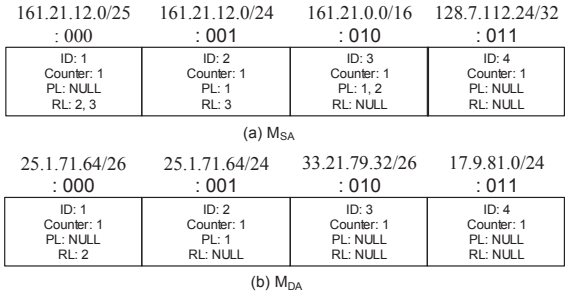

Fig. 5. Processing Prefix Fields

used in the example. Fig. 4 elaborates how to process an exact-match field, protocol type, and establish HT_{prot} with five buckets. Two buckets are assigned to entries with field values, TCP and UDP, respectively. Each entry has a counter value 2. Suppose l_{prot} is 2-bit long and two corresponding bit strings are 00 and 01.

Fig. 5 illustrates how to construct ID maps M_{SA} and M_{DA} for two prefix fields, source/destination IP addresses SA and DA. We do not show the construction procedure of LPM_{SA} and LPM_{DA}. In Fig. 5 (a) for the first prefix 161.21.12.0/25 in $R1$'s SA field, the first entry is created with empty PL and RL, and ID value 1. The subsequent prefix 161.21.12.0/24 overlaps with 161.21.12.0/25 and hence the latter's ID is inserted into the second entry's PL and the second entry's ID is appended into the first entry's RL. For the third prefix 161.21.0.0/16, its PL contains both previous entries' ID values due to overlapping. The last prefix overlaps with no one and hence its PL and RL are both empty. All the entries have the same counter value 1 since there is no duplicate values in SA field. In Fig. 5 (b) M_{DA} is constructed in the similar way.

Fig. 6 shows the procedure to construct RHT using the sample flow table. Suppose RHT has 13 buckets, $b1$ to $b13$. Given $R1$, lookup in M_{SA}, M_{DA} and HT_{prot} results in SA's ID set containing only ID value 1, DA's ID set containing only ID value 1 and protocol ID 1. Perform bit mapping and we get 000, 000 and 00 respectively. Next cross concatenation produces a unique RHT entry in $b1$ with hash key $h(00000000)$, $R1$ as TP and empty LP. Given $R2$, lookup in M_{SA}, M_{DA} and HT_{prot} results in SA's and DA's ID sets containing two ID values respectively and protocol ID 1. After bit mapping, cross concatenation generates 4 RHT entries. One of them falls into the same bucket $b1$ where $R1$ resides and thus $R2$ is added into the entry's LP. Following the same processing steps $R3$ and $R4$ are handled. Except the entry in $b1$ all the other entries have empty LP.

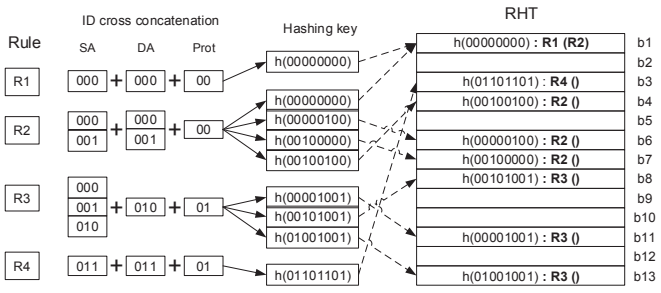

Fig. 6. Rule Hash Table Construction

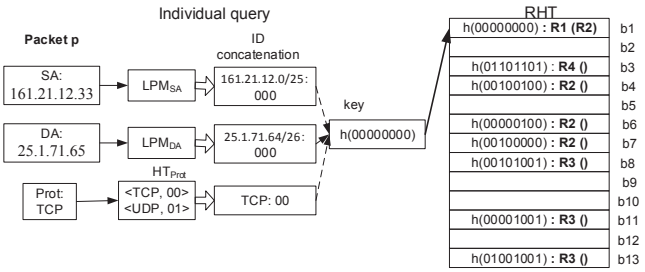

Fig. 7. Query Example

3.4 Query

Algorithm 3 formulates the query algorithm. Given a packet p single-dimensional search is performed in HT_{F_i} and LPM_{F_j} firstly. Next bit mapping and string concatenation are applied upon intermediate result to produce the lookup key to RHT. query() is the function to find LPM.

Algorithm 3 Query Algorithm

FuncQuery($pVals$,$vExt$,$lVec$,RHT,h) {$pVals$: an array of header field values; $vExt$: a vector of all HT_{F_i}; $lVec$: a vector of all LPM_{F_j}; $h()$: RHT's hash function}
Initialize an empty bit string buffer str;
for all i from 1 to e **do**
  str.append($vExt[i]$.get($pVals[i]$));
for all j from $e+1$ to d **do**
  str.append($lVec[j]$.query($pVals[j]$));
key = $h(str)$;
return RHT.get(key);

Fig. 7 demonstrates a query example on basis of data structures constructed after the preprocessing. A given packet has SA 161.21.12.33, DA 25.1.71.65 and protocol type TCP. Lookup in LPM_{SA} and LPM_{DA} returns the longest matched prefixes 161.21.12.0/25 and 25.1.71.64/26 respectively. The matched protocol in HT_{prot} is TCP. Bit mapping and ID concatenation result in a lookup key 00000000. Querying RHT returns the TP value $R1$ in the bucket $b1$.

Like other decomposition solutions our approach scales well to the number of fields since multidimensional search is decomposed into a set of single-dimensional search performed in parallel. Furthermore the light computation overhead in string bit mapping and ID concatenation as well as constant-time access to RHT allow deterministic query, desirable for any substantial implementation and deployment in NPU.

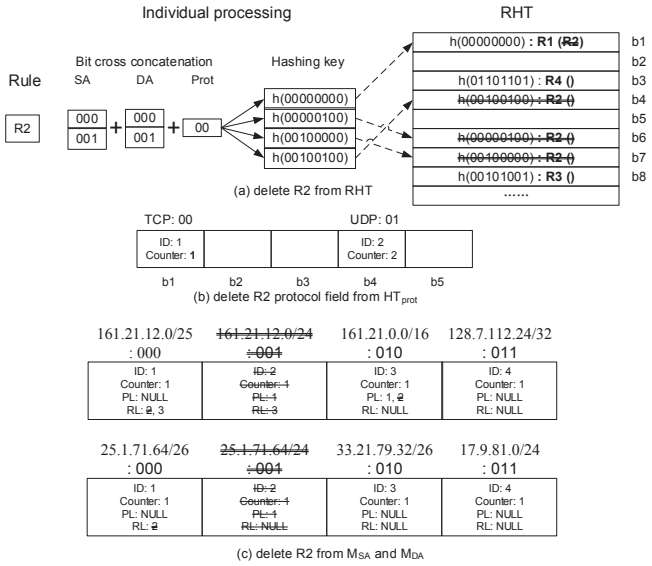

Fig. 8. Delete Example

3.5 Update

Inserting a new rule has been described in the preprocessing procedure. Deleting a rule involves three steps. (1) conduct individual processing on each field by looking up each exact-match hash table and prefix ID map, and then find a set of RHT entries to which the rule is related; (2) delete the rule number from these entries; (3) remove individual field of the rule from exact-match hash tables and prefix ID maps.

Fig. 8 shows the result of deleting $R2$. We do not show irrelevant RHT entries. In Fig. 8 (a) according to SA 161.21.12.0/24, two ID values (1 and 2) are obtained from M_{SA}. As well two DA's ID values are obtained from M_{DA} and one ID value from HT_{prot}. Cross concatenation finds out total 4 RHT entries related to $R2$. Among them LP of the entry in $b1$ is clear to remove $R2$'s rule number. Other entries are directly removed from their corresponding buckets since they only contain $R2$ in TP. In Fig. 8 (b) the counter of TCP entry in HT_{prot} decrements. In Fig. 8 (c) SA 161.21.12.0/24 is the only prefix in M_{SA} and should be hence removed. Before removing it, its PL is checked in order to remove $R2$'s SA ID from RL of the entry with prefix 161.21.12.0/25. As well its RL is checked to remove $R2$'s ID from PL of the entry with prefix 161.21.0.0/16. Eventually the entry is deleted from M_{SA}. So are the same processing steps to delete $R2$ from M_{DA}.

These assistant lists, PL and RL in ID maps and LP in RHT, enable incremental update. As a tradeoff, more memory is used. However it is worth because flow tables must support dynamic incremental update. Therefore our approach overcomes the common drawback of update difficulty in classic decomposition algorithms such as RFC.

IV. EVALUATION

4.1 Theoretical Analysis

TABLE II refers to [16] and compares several performance metrics among HyperCuts, RFC and our approach. Given a flow table, e, d, N and L denote the number of exact-match fields, the number of fields, the number of rules, and the search time using a given LPM method.

We analyze time complexity in two modes. The difference is whether parallel processing in individual field is allowed. Parallel mode is of particular significance to NPU since multiple threading enables single-dimensional processing in parallel. Under sequential mode it takes $O(d)$ time to process all fields for each algorithm. Under parallel mode it takes $O(\log d)$ time in RFC as there are $O(\log d)$ levels of lookup tables and only in each level search can be performed in parallel. For HyperCuts the complexity is still $O(d)$. In our approach each field is separately processed. For each exact-match field F_i the search time is $O(1)$ due to only one hash access to HT_{F_i}. For each prefix field F_j the search time is $O(L)$ to access LPM_{F_j}. Therefore the search time of single-dimensional processing under parallel mode is $O(L)$. In addition, one hash access is needed to query RHT. Consequently the total search time in our approach under parallel mode is $O(L) + O(1)$. From the perspective of memory access which fundamentally determines search time, the number of memory access to HT_{F_i} and RHT is both 1 and hence the majority of memory access occurs in searching LPM_{F_j}. If a TCAM device is used in a given LPM method, $O(1)$ time can be achieved to query LPM_{F_j}. Even some software-based LPM methods are efficient with only a couple of memory accesses. We will adopt three various LPM methods and compare their memory access later.

All the three algorithms consume $O(N^d)$ memory in the worst scenario. In practice our approach consumes more than the others as memory consumption is traded for improvement in other metrics, especially for update purpose. In essence the construction of RHT simply leads to more crossproducting of overlapping prefixes. In RFC partial crossproducting of prefixes is encoded in lookup tables in each phase and hence reduces the overall memory usage. However RFC reduces memory usage with a tradeoff of incremental update. In an OpenFlow switch incremental update is compulsory since remote controller requires dynamic inserting or removing flow table entries. Meanwhile we observe that current flow tables are usually small. The authors of [7] observe that typical implementations of OpenFlow, for example, limit the number of entries in each such table to only 750, while handling about 100,000 concurrent flows. The authors of [8], [19] indicate that the flow table size of most current OpenFlow switches is small, e.g. about 1500 on HP ProCurve 5406z1 and is about 1661 on Pronto 3290. Even using larger flow tables, i.e. synthetic classifiers, it is proved that the actual memory usage in our approach is still moderate. Moreover memory usage can be further alleviated according to the pipeline structure of flow tables which enables flexible partitioning. We observe that only partitioning a small part of table entries gives rise to great memory reduction in our experiments.

Our approach scales to the number of fields well since an arbitrary number of fields with any match type can be processed in parallel. RFC has a moderate scalability as it contains $O(\log d)$ levels of lookup tables to process individual fields. Decision-tree algorithms such as HyperCuts do not scale to the number of fields.

Classbench [16] suggests to process individual field using an LPM method so as to decrease memory access in phase-0 lookup of RFC and allow incremental update. Our approach honestly adopts the suggestion while RFC does not support incremental update.

In RFC search in lookup tables of each phase is not deterministic and phase partition depends on heuristic properties in flow tables. Our approach separates the multidimensional search from final query in RHT. Search in RHT and HT_{F_i} for each exact-match field F_i is deterministic because of straightforward hash access. The overall determinism can thus be improved provided that deterministic query is accomplished by the given LPM method. Special-purpose network processing hardware such as NPU extremely desires deterministic data structure since multiple threading is intensively used to accelerate the query performance.

4.2 Implementation and Measurement

We implement our approach in a commodity PC with g++ 4.6.3 compiler of the Ubuntu/Linaro, 3.0GHz Duo CPU with 32KB L1, 3072KB L2 caches and 4GB DRAM. Total 12 synthetic classifier seeds from Classbench [16] are used to generate flow tables. For each classifier seed we produce five groups of flow tables, containing 1000 (1k), 2500, 5000 (5k), 7500 and 10000 rules respectively. For each classifier we repeat experiments for 10000 times and record the average experimental results. From experimental results we discover that performance evaluation upon searching time, memory usage, update ability and scalability is irrelevant to flow table size and thus we only illustrate the result of 1k and 5k classifiers in the following charts. These seeds cover five firewalls (FW1 to FW5), five Access Control Lists (ACL1 to ACL4) and two IP Chains (ipc1 and ipc2). Each flow table entry (except those in scalability experiments) is a classic quintuple rule (source/destination IP addresses, source/destination port numbers and protocol type), involving all four match types. Our approach is compared with RFC (source code from Classbench [16]) in search time, memory consumption and update efficiency. The range and wildcard field values have been converted into exact or prefix values.

4.2.1 Search Time: Fig. 9 displays the search time of three data structures using 1k flow tables. ***LPM*** represents the longest searching time in LPM_{F_j} for some prefix field F_j, usually source IP address. ***Exact-Match*** indicates the longest searching time in HT_{F_i} for an exact-match field F_i, e.g. protocol type. ***RHT*** indicates the time to query RHT. Querying LPM_{F_j} is much more time-consuming than querying the other hash tables as multiple memory accesses are required.

TABLE II
THEORETICAL COMPARISON

Algorithm	Time (Sequential)	Time (Parallel)	Space	Scaling	Update	Determinism
HyperCuts	$O(d)$	$O(d)$	$O(N^d)$	No	Yes	No
RFC	$O(d)$	$O(\log d)$	$O(N^d)$	Yes	No	No
Ours	$O(d)$	$O(L)+O(1)$	$O(N^d)$	Yes	Yes	Yes

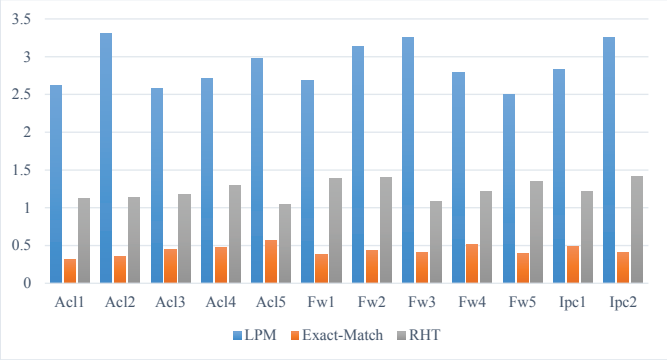

Fig. 9. Search Time (μs) in LPM_{F_j},HT_{F_i} and RHT

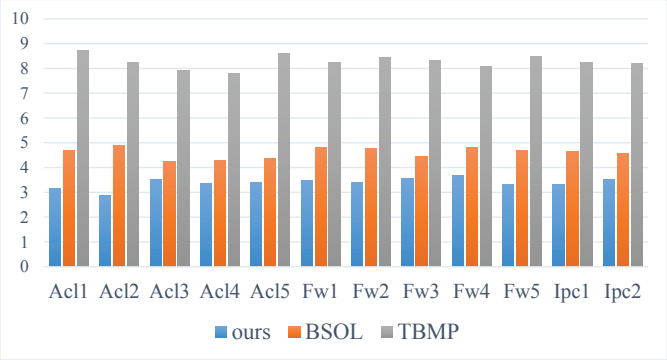

Fig. 10. WAMA comparison

Querying RHT is slower than querying HT_{F_i} because of extra ID concatenation and hash calculation with string-type values. The actual processing time in each data structure is consistent with theoretical analysis.

We use Weighted Average Memory Access (WAMA) to effectively evaluate average memory access. We implement three LPM methods, hierarchical hashing, BSOL and TBMP [2], [14], [18]. Detailed introduction of these LPM methods can be found in [14]. WAMA for hierarchical hashing is defined by Equation 1. $K+1$ is the number of memory access for each hash table $HT_{p,q}$ of the method. $E_{p,q}$ is the number of entries in $HT_{p,q}$. M is the number of prefixes in the field F_j.

$$\frac{\sum(K+1)\times E_{i,j}}{M} \tag{1}$$

WAMA for BSOL is formulated in Equation 2. K is the number of memory access for each hash table in BSOL. E_K is the number of entries in each hash table.

$$\frac{\sum K\times E_K}{M} \tag{2}$$

WAMA for TBMP is formulated in Equation 3. Let hs-bit be the fixed TBMP stride. $2K$ is the number of memory access for each trie node. E_K is the trie node number in K level.

$$\frac{\sum 2K\times E_K}{M} \tag{3}$$

Fig. 10 compares WAMA values of these LPM methods using 1k flow tables. The hierarchical hashing method averagely looks up a prefix using less than 4 memory accesses for the most time-consuming prefix match on source IP address. BSOL and TBMP consume a few more memory accesses per query. Even regarding one additional memory access to RHT, only several memory accesses are needed to perform multidimensional search with an arbitrary number of prefix fields.

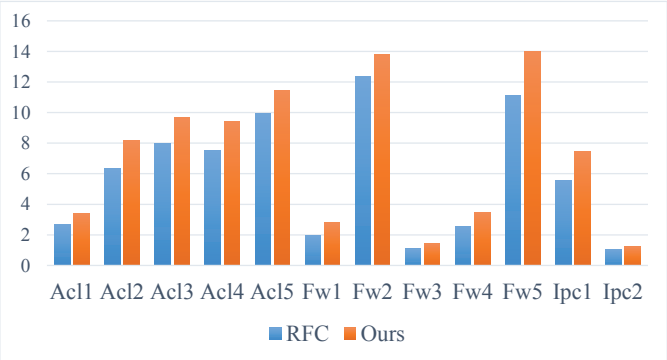

Fig. 11. Memory Usage (Mb) for 1k flow tables

4.2.2 Memory Usage: Fig. 11 illustrates the memory usage of 1k flow tables. For example for FW2 our approach consumes about 18Mb memory while RFC consumes 12Mb. Our approach consumes averagely 40% more memory than RFC. Table partitioning has no influence to 1k flow tables.

Fig. 12 illustrates the memory usage of 5k flow tables with or without table partitioning. Without partitioning our approach consumes more memory than RFC. We observe that usually a small number of contiguous, wide-range entries (prefixes) exist at the end of each flow table, e.g. 128.0.0.0/1 on both source and destination IP addresses. These entries have the lowest priority due to their bottom positions. Although these entries, e.g. the last 43 rules in fw2, occupy a few percentages in a flow table, they overlap with a large number of

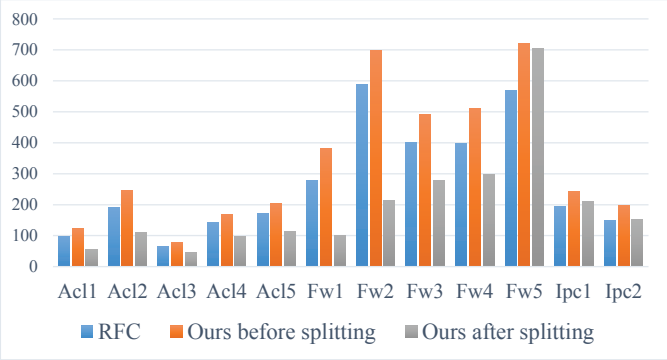

Fig. 12. Memory Usage (Mb) for 5k flow tables

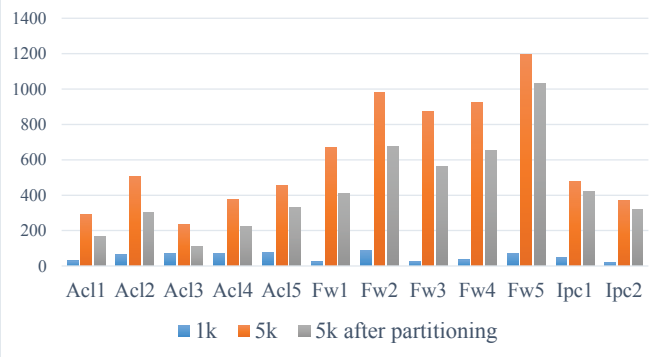

Fig. 13. PL Update frequencies

higher-priority prefixes and hence contribute to a considerably large number of RHT entries due to crossproducting. We apply table partitioning upon each 5k flow table such that these low-priority, wide-range prefixes are separated out to a new small flow table. We evaluate our approach using new flow tables in which these prefixes have been removed and discover that much less memory is consumed in our approach except for classifiers FW5, ipc1 and ipc2. For example for FW2 after table partitioning only one third of original memory consumption is used for new table in which such 43 rules have been removed. In other words these 43 rules contribute to two thirds of original memory consumption. In flow tables FW5, ipc1 and ipc2, table partitioning does not work because those low-priority, wide-range prefixes primarily exist in only one prefix field and thus do not contribute to much RHT entries. The divide-and-conquer methodology actually does not bring more complexity in flow processing because the small number of rules have rare statistical chance to be matched due to their lowest priority.

4.2.3 Update: In our approach updating an exact-match value in a rule needs to access its corresponding HT_{F_i} entry and a number of RHT entries whose number is relevant to the match types of other field values in the rule. The more complicated update occurs in a prefix field F_j while a number of entries in LPM_{F_j}, M_{F_j} and RHT are updated. The complexity of updating LPM_{F_j} is determined by the chosen LPM method. Updating M_{F_j} only involves one row operation

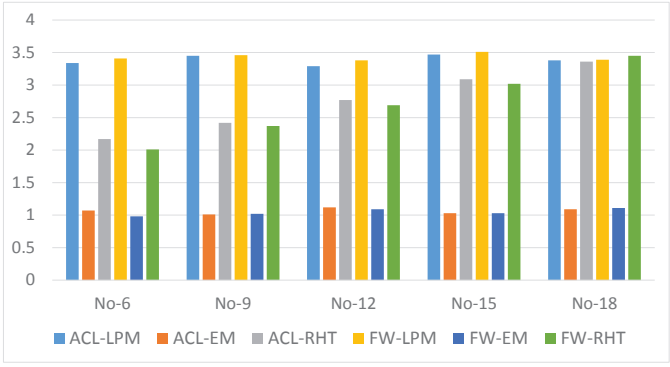

Fig. 14. Search Time (μs) in LPM_{F_j},HT_{F_i} and RHT with more fields

per update request. Owing to crossproducting of prefix values in a rule, updating each RHT entry corresponding to a rule may invite a great number of operations in PL and RL for the entry. These operations in PL and RL constitute the majority of update primitives in our approach. The preprocessing procedure in our approach provides a continuous view of how incremental updates are performed for a given flow table. Fig. 13 exhibits PL update frequencies using 1k and 5k flow tables with or without table partitioning. The PL update frequency is defined as the average number of PL operations (insertions when adding a rule and deletions when removing a rule) within all the RHT entries for each rule during preprocessing. The metric counts the total number of ID insertions in PLs of those RHT entries which are generated by a single rule. In the meantime the same number of ID insertions occur in RLs of corresponding entries. The number of insertions and deletions in PL and RL of RHT entries corresponding to prefix field values for a rule is considerably larger than update operations in HT_{F_i}, LPM_{F_j} and M_{F_j}. Updating a rule in 1k flow tables incurs much less PL operations than in 5k flow tables. In addition, update efficiency actually reflects the overlapping relations of prefixes in RHT entries which primarily determines the memory efficiency. That accounts for the obvious decreasing of PL update frequencies for most flow tables with table partitioning except for three classifiers (FW5, ipc1 and ipc2).

4.2.4 Scalability: Because Classbench seeds only produce two prefix fields (source/destination IP address) and one exact-match field (protocol type), we repeatedly apply the same seed file upon two classifiers (1k versions of ACL2 and FW3) with different parameters for more prefix and exact-match fields and values. These new fields and values are merged to produce new classifiers with more fields. Eventually we obtain 10 new classifiers with 6, 9, 12, 15 and 18 prefix and exact-match fields. The ratio of prefix fields to exact-match field (2:1) is maintained. Range fields are not considered in this experiment. We evaluate the search time in LPM_{F_j}, HT_{F_i} and RHT. Fig. 14 demonstrates the search time using these classifiers. *ACL-LPM* represents the longest searching time in LPM_{F_j} of new ACL2 tables for some prefix field F_j. *ACL-EM* indicates the longest searching time in HT_{F_i} for some exact-match field

F_i and *ACL-RHT* indicates the time to query RHT. *FW-LPM*, *FW-EM* and *FW-RHT* have the same representation for new FW3 tables.

Compared to Fig. 9 we learn that querying LPM_{F_j} or HT_{F_i} makes no difference since the same number of memory accesses are spent regard less of the number of fields. It takes more time querying RHT because more ID concatenation and hash calculation are needed. However the search time in RHT increases moderately when the number of fields increases drastically. For example, for new ACL2 and FW3 tables with 18 fields the search times in RHT are 3.36 and 3.45 μs respectively. By comparison to 1.12 and 1.39 μs for original ACL2 and FW3 tables with 3 prefix or exact-match fields, the search times nearly triple while the number of fields increases to six times. Furthermore the entire query time for new ACL2 table and original ACL2 table are 6.74 μs (3.36 in RHT plus 3.38 in LPM search) and 4.50 μs (1.12 in RHT plus 3.38 in LPM search) respectively. Only 49.78% more time is spent in search even when the number of prefix and exact-match fields increases to six times. Therefore our scheme scales well to the number of fields.

V. Conclusion

To enable highly scalable, flexible software-defined networks, OpenFlow faces a great challenge of classifying a flow table with an arbitrary number of fields as well as any match type. Current well-known packet classification algorithms suffer from scalability to the number of fields, incremental update and deterministic query. We propose an decomposition approach to divide a multidimensional search into a set of single-dimensional searches which can be performed in parallel. A final query is simply conducted in a unique hash table. The scalability to the number of fields, incremental update and deterministic query can be achieved accordingly. This is the first serious proposal to solve multidimensional match in an OpenFlow flow table. Theoretical analysis and experiments using synthetic classifiers justify the performance improvement.

References

[1] OpenFlow Switch specification version 1.4.0 (wire protocol 0x05). In *OpenFlow Specification*, October 2013.

[2] W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap: Hardware/software ip lookups with incremental updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[3] G. Varghese F. Baboescu. Scalable packet classification. *IEEE/ACM Transactions on Networking*, 13(1):2–14, February 2005.

[4] Open Networking Foundation. Software-defined networking: The new norm for networks. In *ONF White Paper*, 2012.

[5] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proc. of IEEE/ACM Sigcomm*, pages 147 – 160, 1999.

[6] P. Gupta and N. Mckeown. Packet classification using hierarchical intelligent cuttings. In *Proc. Hot Interconnects VII*, pages 34–41, 1999.

[7] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *Proc. IEEE Infocom*, pages 545–549, 2013.

[8] X. Kong, Z. Wang, X. Shi, X. Yin, and D. Li. Performance evaluation of software-defined networking with real-life isp traffic. In *Proc. IEEE Symposium on ISCC*, pages 541–547, 2013.

[9] S. Kumar, J. Turner, and P. Crowley. Peacock Hashing: Deterministic and updatable hashing for high performance networking. In *Proc. IEEE Infocom*, pages 556–564, 2008.

[10] M. Martinello, M. R. N. Ribeiro, R. E. Z. de Oliveira, and R. de Angelis Vitoi. Keyflow: a prototype for evolving sdn toward core network fabrics. *IEEE Network*, 28(2):12–19, April 2014.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. In *OpenFlow White Paper*, 2008.

[12] I. Papaefstathiou and V. Papaefstathiou. Memory-efficient 5d packet classification at 40 gbps. In *Proc. IEEE Infocom*, pages 1370–1378, 2007.

[13] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proc. of IEEE/ACM Sigcomm*, pages 213–224, 2003.

[14] H. Sun, Y. Sun, V. C. Valgenti, and M. S. Kim. A hierarchical hashing scheme to accelerate longest prefix matching. In *Proc. IEEE Globecom*, pages 1296–1302, 2014.

[15] H. Sun, Y. Sun, V. C. Valgenti, and M. S. Kim. A highly deterministic hashing scheme using bitmap filter for high speed networking. In *To be published in IEEE ICNC*, 2015.

[16] D. E. Taylor and J. S. Turner. ClassBench: A packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3):499–511, June 2007.

[17] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: optimizing packet classification for memory and throughput. In *Proc. IEEE/ACM Sigcomm*, pages 207–218, 2010.

[18] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *Proc. IEEE/ACM Sigcomm*, pages 25–36, 1997.

[19] P. C. Wang. Scalable packet classification with controlled cross-producting. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 53(6):821–834, April 2009.