ORange: Multi Field OpenFlow based Range Classifier

Liron Schiff*
Tel Aviv University
Tel Aviv, Israel
schiffli@post.tau.ac.il

Yehuda Afek
Tel Aviv University
Tel Aviv, Israel
afek@post.tau.ac.il

Anat Bremler-Barr
The Interdisciplinary Center
Hertzelia, Israel
bremler@idc.ac.il

ABSTRACT

Configuring range based packet classification rules in network switches is crucial to all network core functionalities, such as firewalls and routing. However, OpenFlow, the leading management protocol for SDN switches, lacks the interface to configure range rules directly and only provides mask based rules, named flow entries.

In this work we present, ORange, the first solution to multi dimensional range classification in OpenFlow. Our solution is based on paradigms used in state of the art non-OpenFlow classifiers and is designed in a modular fashion allowing future extensions and improvements. We consider switch space utilization as well as atomic updates functionality, and in the network context we provide flow consistency even if flows change their entrance point to the network during policy updates, a property we name cross-entrance consistency. Our scheme achieves remarkable results and is easy to deploy.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications, Packet-switching networks*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*

; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations* ; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*

Keywords

Software Defines Networks; Packet Classification; Consistency

1. INTRODUCTION

Software Defined Networks (SDNs) are considered the next step of networks management evolution. The SDN paradigm splits the network into a "dumb" data plane whose main purpose is to process and forward packets according to the rules installed by the network operating system, i.e., the "smart" controllers in the control plane. The most dominant SDN control protocol is OpenFlow, which is used by controllers to install rules on the switches and to receive notifications on network events.

The rules used in recent years by different networking devices are becoming more complicated and many times specified by a multi dimensional specification, i.e., specify two or more ranges of possible values. E.g., a range of source IP addresses and a range of destination port numbers. Or, another example, drop packets with source port between 50300 and 50500 and destination port below 1024. However, due to processing speed and resource utilization requirements, OpenFlow doesn't support methods to install range based rules directly. Instead OpenFlow allows to install match-action rules that use ternary strings as match patterns, requiring each range to be encoded by multiple ternary patterns and multi dimensional ranges to be encoded by even larger number of patterns.

The problem of encoding ranges with ternary patterns was studied before OpenFlow has been invented. Regular network switches, as many OpenFlow ones, are based on the same underline packet match-action hardware implementations such as Ternary Content Addressable Memory (TCAM) chips. However, to the best of our knowledge, no encoding has ever been designed for OpenFlow controlled switches.

In this paper we present ORange, a new OpenFlow based multi dimensional classification scheme. Our contributions are in three levels. At the first level we use OpenFlow features in a sophisticated way to implement ORange1, which is an extension and implementation of ideas we presented as brief announcement [4]. ORange1 is a one dimensional range classification scheme which is more space efficient (only 2 entries per range) than any others known classifiers. In the second contribution, using ORange1 as a building block, we present ORange-k, a k dimensional range classifier for SDN switches using techniques for non-OpenFlow classifiers such as [12]. Finally, we show how to update ranges across multiple switches in an atomic manner - allowing to update the set of ranges and their associated actions while packets are classified and the network is changing. Our schemes are suitable for several applications such as load-balancing, QoS, routing, firewalls and ACLs.

The ORange1 scheme utilizes basic OpenFlow features to efficiently implement a complex computation of a non-overlapping range classification scheme which requires only $2n + 2w + 1$ table entries (a significant saving is observed already with 4 ranges - see Figure 5), where n is the number of ranges and w is the size (number of bits) of values (e.g., 32 for IPv4 addresses). The two key OpenFlow features that we use are: (i) A packet can be processed by several forwarding tables using 'goto' action command to decide on the next table. (ii) A packet can be extended with an auxiliary field which may be altered (e.g., xored with a constant) and considered as the packet goes through subsequent flow tables. Moreover, an initial version of our ORange1 scheme (without updates) was presented as a brief announcement at ONS 2014 [4] and HotSDN 2014 [5] conferences. Recently, the ORange1 as presented here has been successfully verified and operated in a Mininet [11] virtual network using RYU [3] based controller [2].

*Supported by the European Research Council (ERC) Starting Grant no. 259085 and by the Israel Science Foundation Grant no. 1386/11.

On top of ORange1 we present the ORange-k classification scheme that supports k dimensional ranges and achieves exponential with k space improvement by using cascaded instances of the ORange1 classifier. Each instance of ORange1 is responsible to map different field in the packet so the final k dimensional classification is simpler and consumes less space.

Three levels of atomicity are considered and supported by our schemes: (i) Per packet consistency - each packet is handled according to a correct configuration either before or after the update. (ii) Per flow Consistency - all packets of a flow are handled according to the same configuration and new flows are handled according to the most recent configuration. (iii) Cross-entrance consistency - keeping Per Flow consistency even when the flow changes the entrance point to the network.

Per Packet Consistency in the general case was dealt by Reitblatt et. al. [14], who suggested to use an extra packet header field to indicate the version of the rules that should be applied to the packet throughout the network. This enables the operator to safely install new rules while packets are marked for old rules and later, by a single (atomic) update, to change the packet marking to the new rules. In our work, we present a range classification scheme that is atomic by itself without consuming packet header space. Therefore, our scheme can be utilized to extend and develop consistency frameworks.

Per Flow consistency, in the load balancing scenario was presented by Wang et. al. [19], considering an update that affects a sub-range of client IPs by changing the associated OpenFlow action from "old" to "new". According to their suggestion the controller should monitor the flows that belong to the updated sub-range and then for each flow that exists before the update a specific rule with "old" action is created and rules with "new" actions are created for new flows. Following Reitblatt et. al. we use DevoFlow [10] to substantially reduce the amount of control traffic generated by our scheme. In DevoFlow a switch configures new rules in an autonomous way. These new rules are produced by a generic DevoFlow rule.

Our ORange1 scheme is based on PIDR, a non-OpenFlow system design by Panigrahy and Sharma [13] which uses special hardware (e.g., ASIC or FPGA) and has no notion of atomicity. PIDR creates two ternary patterns for each range called the ELCP0 and ELCP1 of the range (see definition in Section 3.1) and save them separately in two TCAMs associated with range bounds. The two TCAMs are arranged in a very specific manner to support the scheme. In order to classify a value, it should be queried against the two TCAMs, returning two matches that are associated with two possible ranges. Then the value is checked whether it belongs to one of these two ranges.

Related work is given in Section 2. Section 3 contains the background and details of our ORange1 classification scheme. Section 4 describes our ORange-k scheme which is based on ORange1. In section 5 we extend our schemes to support per-flow and cross-entrance consistency requirements. A correctness proof is given in the Appendix.

2. RELATED WORK

Encoding ranges as ternary strings is a well studied problem. The traditional technique for range representation, originated by Srinivasan et al. [17] and called *prefix expansion*, is to represent a range by a set of prefix patterns, each of which can be stored in a single memory (e.g., TCAM) entry. The worst-case expansion ratio when using prefix expansion for ranges whose endpoints are w-bits numbers is $2w-2$ (entries per range).

State of the art hardware solutions can be categorized to two groups, database dependent or database independent. Database dependent encodings consider the entire set of ranges to be encoded (the database) and seek to optimize the combined output for encoding them all, while in a database independent encodings each range is always encoded the same (regardless of the database). The optimal database-independent encoding [16] uses $2w-4$ patterns per range in the worst case and w entries in the average case [15].

For a multi dimensional (multi field) ranges no better database independent encodings are known other than joining the encoding for each field in a cross product manner, requiring w^k patterns per range on average, where k is the dimensionality of the ranges.

In contrast, database dependent encodings transform field values and reencode the ranges accordingly in a way that optimizes the total space for all rules [8, 9] and/or power consumption per query [6, 12]. Due to the inherent high dependency of one range rule encoding with others, it is hard to add even a single rule without changing huge amount of entries in response, therefore implementations of these encodings usually require to maintain two versions of the configuration [12] and/or performing updates in batches [8]. To the best of our knowledge, no efficient encoding has ever been implemented in an OpenFlow controlled switch.

Note that we consider here general policies where each range can be associated with a unique action, and not two actions policies like ACLs where only ALLOW or DROP actions are possible. In case of a two actions policy better encoding are known [15].

3. OPENFLOW BASED ONE DIMENSIONAL RANGE CLASSIFIER (ORange1)

3.1 Background - PIDR

A set of non-overlapping ranges is a set of the form $\{[a_1,b_1],[a_2,b_2],...\}$, where a_i and b_i are integers, $a_i \leq b_i$ and $b_i < a_{i+1}$, and by range $[a_i,b_i]$ we consider all integers x such $a_i \leq x \leq b_i$. The set supports three operations: the insertion of a new range, deletion of an existing range and a point query (looking up for the range that contains a given point). Only sets of non-overlapping ranges are considered here.

Panigrahy and Sharma presented [13] PIDR (Point Intersection Disjoint Ranges) that can manage a set of disjoint ranges using two TCAM entries per range, and two concurrent TCAM queries per point lookup. Following their work [13] we define:

- *The Longest Common Prefix (LCP) of integers a and b (represented by w bits)* is the longest prefix shared by the binary representations of a and b. For example, LCP($a = 0101$,$b = 0111$) $= 01$.

- *The Extended LCP (ELCP) patterns of integers a and b, named* 0*-ELCP and* 1*-ELCP* are the patterns that extend the LCP of a and b by one more bit (0 and 1) and use '$*$'s' for the remaining bits, for example, let $a = 0101$,and $b = 0111$ then 0-ELCP(a,b)=010$*$, and 1-ELCP$(a,b) = 011*$, ($w = 4$).

The set of ranges is maintained by keeping for each range $[a,b]$ its two ELCP patterns in two separate TCAMs, one for the 0-ELCP's and one for the 1-ELCP's. The ELCP patterns, that are stored in the TCAMs, are associated with range objects that are located in SRAM. It is easy to see that if $x \in [a,b]$ then x matches exactly one of the ELCP patterns of $[a,b]$. But the opposite is not always true, x might match an ELCP of a range that does not contain x. However, while x might match other ELCP's, the a,b ELCP it matches is the longest one, i.e., with the fewest number of $*$'s as proved in [13].

Thus if $x \in I = [a, b]$, then the binary presentation of x matches either 0-ELCP(a, b), or 1-ELCP(a, b). Therefore, if the ELCP's are stored in each TCAM in order of increasing number of "don't care"s ($*$'s) then a TCAM that returns the first entry with a pattern that matches x returns either the 0-ELCP or the 1-ELCP of I, but not both.

The algorithm for range search, as described in Figure 1 (taken from [13]) is simple, we query both TCAMs with the point in question, then we check if one of the matches belongs to a range that contains the point.

However implementing this algorithm requires special pipeline hardware and it does not support updates (changing the set of ranges) concurrently with queries. In this paper we show a redesign and an extension of this algorithm implemented in the OpenFlow framework. Our design supports atomic concurrent updates and conforms to consistency models which makes it applicable to many applications.

3.2 Our Implementation

As a first step in adapting PIDR to OpenFlow, we turn PIDR into a sequential process (see Figure 2). According to the original design, both TCAMs are queried in parallel and the two results are checked (by comparing the queried value to the start or end of the resulted ranges) in parallel. In the new design we query one of the TCAMs, check the first (range) result, if the check succeeds we output the range, otherwise we proceed and query the second TCAM. If both checks fail (or no match found at all) we output "no range found".

In order to implement the sequential PIDR version in OpenFlow we implement (see Figure 3) the two TCAMs (TCAM0 and TCAM1) as (OpenFlow) flow tables and each of the compares is implemented by a special 2w entries flow table, as explained in the next subsection. The values to be compared and the queries' results are transfered from one table to another by rewriting (set-field action) temporary fields (called metadata in the Openflow 1.3 spec [1] supported by most existing switches) of the inspected packet header. Moving and jumping between tables is directed by goto actions defined in the entries of the tables (the goto action is restricted in [1] to jump only forward, a restriction satisfied by our design). Finally, the packet tagged with the matching range (or no-range) is sent through a range id flow table (RIDS) that applies the action that corresponds to the matching range, to the packet.

Note that the original PIDR algorithm maintains longest prefix first order among ELCP rules in the TCAMs which we comply to by utilizing rule priority supported by OpenFlow flow tables (setting priority to the prefix length).

Each entry in 1-ELCPs is associated with an action that sets the id and the upper bound (End) of the relevant range in the packet temp field and a goto action to the first Compare table (see Figure 3). The first Compare table then compares the End in the temp field and the searched field. The entries in the Compare table that are matched when $End \geq searched\_field$ have action goto RIDS table, and others have action goto 0-ELCPs table. 0-ELCPs table is similar to 1-ELCPs table with the exception that it sets the lower bound (Start) of the matched range in the temp field. The second Compare table inspects the Start in the temp field and the searched field and compares them. The entries in the compare table that are matched when $Start \leq searched\_field$ have the action goto RIDS table, and others have a default action corresponding to no matching range is found (probably Drop or Send To Controller).

Note that the RIDs table can be implemented by OpenFlow groups table, since its entries are exact (has no don't cares). Moreover if the entire space is covered by ranges, all queries must match either the result of a 1-ELCPs or the result of a 0-ELCPs table, therefore the second compare is redundant and we can omit it, thereby shortening the worst case pipeline length and saving TCAM space.

3.3 A Comparator Flow Table

A Compare table inspects two fields in the packet's header and compares them (in our use case the first value is the temp header field). The compare table contains $2w + 1$ flow entries, each with 2w bits, as illustrated in Figure 4, where w is the fields' width in bits (e.g., 32 for IPv4 addresses). For $0 \leq i < w$, entries $2i$ and $2i+1$ are used to compare the i-th MSB of the fields, and are of the form $*^i 1 *^{w-i-1} | *^i 0 *^{w-i-1}$ and $*^i 0 *^{w-i-1} | *^i 1 *^{w-i-1}$ respectively, where an entry of the form $X|Y$ means that X is matched against the first value and Y is matched against the second value. Even entries are successfully matched when the first value is larger than the second one and are all associated with the corresponding "larger than" action. Odd entries are successfully matched when the first value is smaller than the second one and are all associated with the corresponding actions. The last entry (row $2w$), contains only don't cares, and is successfully matched when the two values are equal and is associated with the corresponding action.

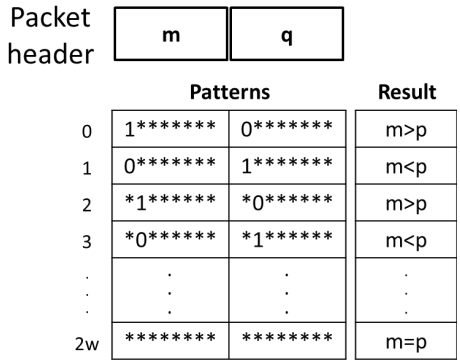

Figure 4: Compare table structure. Packet header contains the compared values m and q where in our scheme the first is a part of a temp field

In addition we point out that using the recently introduced RMT (Reconfigurable Match Table [7]) OpenFlow model it is possible to replace the compare-table with a simple computation stage.

3.4 Space Complexity

We compare our scheme to existing range classification schemes with TCAMs. The best known schemes use in the worst case $2w-4$ TCAM entries per range [16] but close to w per range on average [15]. Our scheme uses a constant 2 entries per range plus a constant $2w$ entries TCAM for the compare operation, regardless of the number of ranges. Thus for $w = 32$ bits (suitable for IPv4 field classification) and for enough ranges our scheme reduces the space usage by $30/32$ factor (more than 90% savings).

3.5 Atomic Updates for ORange1

Since data packets may arrive during range updates, we must ensure that all of them are classified correctly and consistently - all packets should be affected by the configuration before the update or after, and once a packet is classified according to a new version so are all the following packets.

Note that latest OpenFlow specification defines a new feature of transaction which allows to send multiple commands to a switch

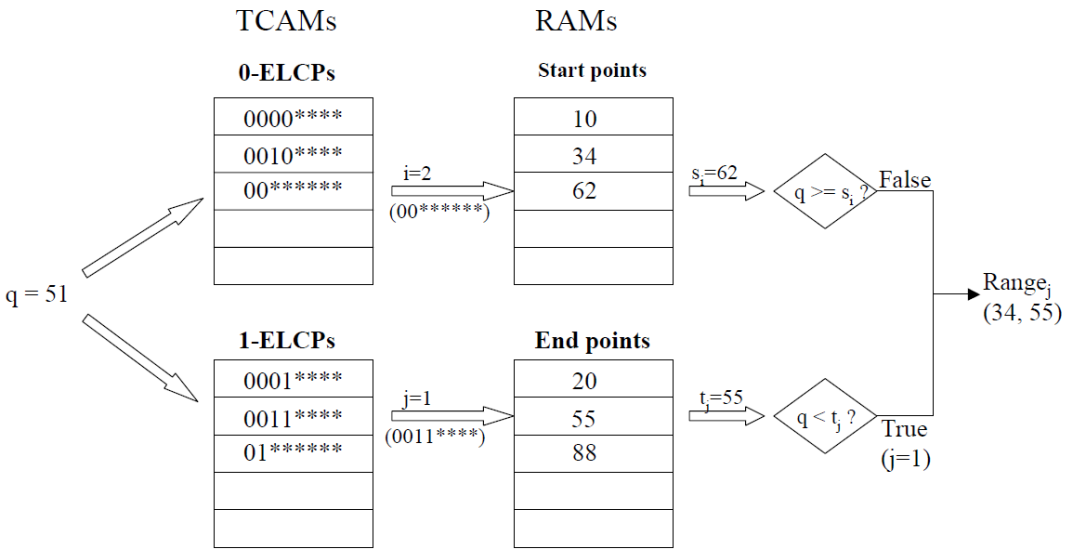

Figure 1: (Figure 3 in [13]) The set contains the ranges $\{[10, 20], [34, 55], [62, 88]\}$ and is searched for the range that contains the value 51.

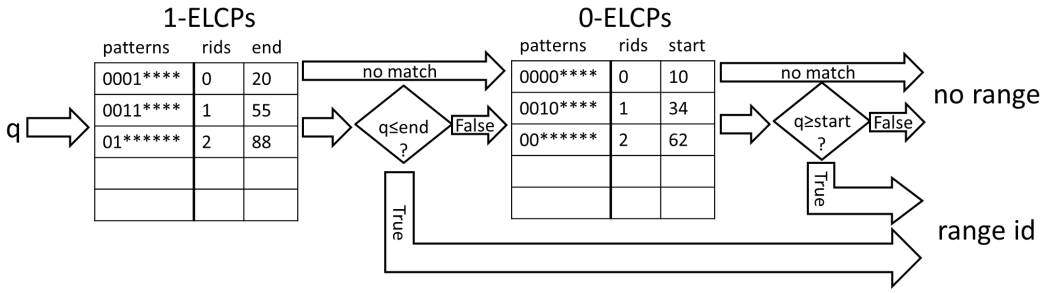

Figure 2: A sequential redesign of Panigrahy and Sharma [13], Figure 1

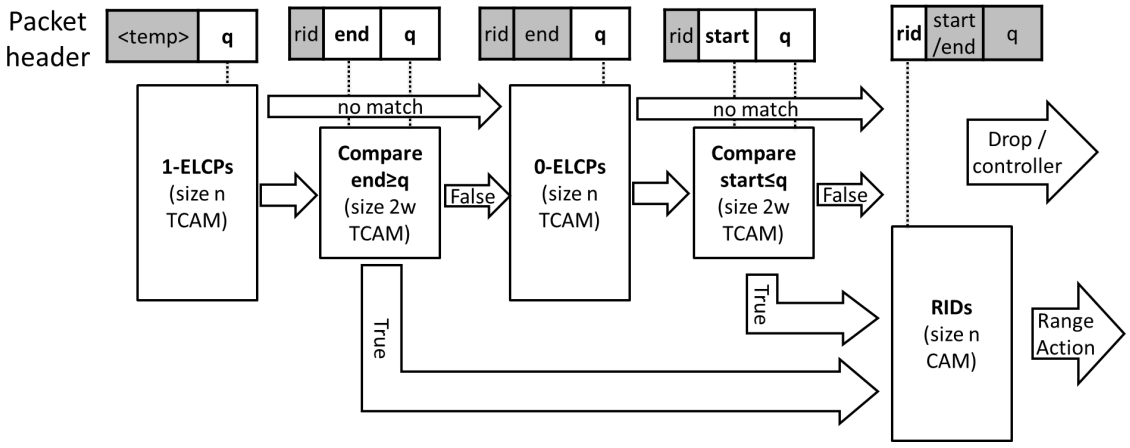

Figure 3: The OpenFlow based design of the range classifier.

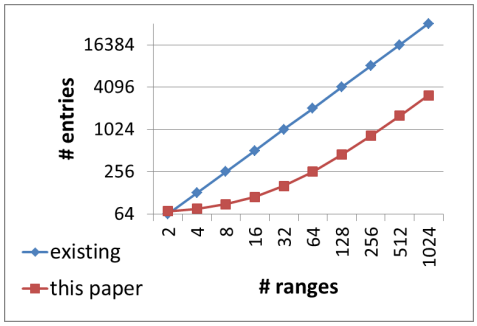

Figure 5: A log-log graph showing the number of flow table entries as a function of the number of ranges.

and to buffer them until a special trigger command instructs to perform them. This enables to make multiple updates to the flow tables atomically while preventing packets from being processed in intermediate states. However this feature, has bad impact on performance (packets may be buffered during transaction) or the cost of the switch (forwarding tables may be duplicated).

In our implementation changing the action associated with a range is done by updating a single entry in the RIDs table and is therefore atomic. We refer to this operation as a range update.

We define two more atomic modifiers to the ranges classifier, a *split* and a *merge*. $Split(R, R_1, R_2)$ takes a range R and splits it to two ranges R_1 and R_2, both with the same action as the original one. $Merge(R_1, R_2)$ takes two adjacent ranges R_1 and R_2 that share the same action and merges them into one range.

We implement the atomic $Split(R, R_1, R_2)$ by considering two types of cases: 1. one of the new subranges shares the same ELCP patterns as R, 2. both subranges have different ELCP patterns . In the first case assuming w.l.o.g., that R_1 shares the same ELCP patterns as R, we add R_2 (ignoring that it overlaps with R), and then we update the upper bound of R to R_1's upper bound (thereby transforming it to R_1). In the second case we add both R_1 and R_2 (ignoring that they overlap with R) and then we completely delete R from all flow tables. As we prove in Appendix A, this order of operations atomically changes the ranges without affecting the packets processed in any step in the way.

Since $Merge$ is the opposite of $Split$, we implement the atomic $Merge(R_1, R_2)$ by following the same steps as $Split$ but in reverse order, thereby ensuring the atomicity proved for $Split$. We provide the pseudo code below and we prove its correctness in Appendix A .

```
1: function SPLIT(R, R1, R2)
2:     if R1 has the same ELCPs as R then
3:         add R2 with new id                          ▷ add to RIDS first
4:         update R's upper bound to R1's upper bound ▷ change action in R's entry in ELSP1s
5:     else
6:         if R2 has the same ELCPs as R then
7:             add R1 with new id                      ▷ add to RIDS first
8:             update R's lower bound to R2's lower bound ▷ change action in R's entry in ELSP0s
9:         else▷ the ELCPs of R1 and R2 are partial of those of R
10:            add R1 with new id                      ▷ add to RIDS first
11:            add R2 with new id                      ▷ add to RIDS first
12:            delete R's entries from all tables
13:        end if
14:    end if
15: end function
16: function MERGE(R1, R2)
17:    set R := R1 ∪ R2
18:    if R1 has the same ELCPs as R then
19:        update R1's upper bound to R2's upper bound ▷ change action in R1's entry in ELSP1s
20:        delete R2's entries from all tables ▷ delete from RIDS last
21:    else
22:        if R2 has the same ELCPs as R then
23:            update R2's lower bound to R1's lower bound ▷ change action in R2's entry in ELSP0s
24:            delete R1's entries from all tables ▷ delete from RIDS last
25:        else▷ the ELCPs of R1 and R2 are partial of those of R
26:            add range R with new id                 ▷ add to RIDS first
27:            delete R1's entries from all tables     ▷ delete from RIDS last
28:            delete R2's entries from all tables     ▷ delete from RIDS last
29:        end if
30:    end if
31: end function
```

Observe that Split and Merge (as defined) doesn't change the way packets are handled. Therefore any combination of multiple $Split$ and $Merge$ operations with one update operation can be regarded as one atomic operation.

Using the last observation we build three additional useful atomic operations;

1. SplitAndCreate(R,R_1,R_2,action2) - splitting R into two adjacent ranges, R_1 and R_2, and then changing R_2's action to action2.
2. MergeDifferent(R_1,R_2) - merging R_2 into R_1 that doesn't share the same actions by first changing R_2's action to R_1's actions and then merging the two ranges.
3. SubRangeTransfer(R_1,R_1-left,R_1-right,R_2) - Moving the border between two adjacent ranges, R_1 and R_2, in a way that shriks R_1 to R_1-left and extends R_2 with R_1-right (R_1-left and R_1-right is a partition of R_1). This is done by first splitting R_1 to R_1-left and R_1-right, then changing the action of R_1-right to be the same as R_2, and merging R_1-right with R_2.

4. THE k FIELDS ORange (ORange-k)

The ORange-k scheme is designed to classify packets according to a set of n multi dimensional range rules, R, that consider k packet fields. Each range $r \in R$ projects a one-dimensional range, $r[i]$, for every field $i \in [k]$ and we denote the lower and upper bounds of $r[i]$ by $r[i].min$ and $r[i].max$ respectively. We denote field size in bits by w which we assume to be much greater than $\log n$, i.e., $n \ll 2^w$.

The scheme operates in two classification phases; In the first phase we reduce the size of each field value to $O(\log n)$ bits, and in the second phase we make a k dimensional classification based on the reduced field values using the prefix expansion encoding (see Figure 6).

The ORange-k scheme drastically improves the space complexity but it increases the pipeline length in k since field reductions are implemented with consecutive instances of our one dimensional classifier ORange1 (see Figure 7). Similar paradigm is used in

some non-OpenFlow designs such as the topological transformation approaches [12] but with prefix-expansion encodings for one dimensional classifiers.

4.1 Per Field Reduction

Here we consider the details of the first phase of the scheme - multiple one dimensional field reductions. For each dimension we independently consider the projection of the ranges on a single axis (dimension). As can be seen in the example in Figure 6.a where 3 two-dimensional ranges are projected on the x and y axises each as two one-dimensional ranges.

For each $i \in [k]$ the one-dimensional projected ranges $\{r[i] : r \in R\}$, are segmented to disjoint subranges, R_i, using the CalcConversionRanges function shown below. The sub-ranges in R_i are matched against the field i of each packet. The index number of the containing sub-range is used as the reduced value for field i. The CalcConversionRanges function also compute the mapping $Map_i[p]$ from original end point value p to the new sub-range index number that contains it, which is used, as explained in the next subsection, to re-encode the original multi dimensional ranges to a reduced form.

Note that for each $i \in [k]$, the CalcConversionRanges function creates (and enumerates) the one dimensional ranges in R_i in consecutive manner and according to an ordered list of all the endpoints of ranges $\{r[i]\}_{r \in R}$. Therefore, every one dimensional range $r[i]$ can be represented as a range over the index numbers of the sub-ranges in R_i. Moreover, since every endpoint of some range $r[i]$ essentially contributes at most one subrange to R_i, the total number of sub-ranges in R_i is bounded by $2n$, and the number of bits used to represent a sub-range index number is $\log_2 n + 1$.

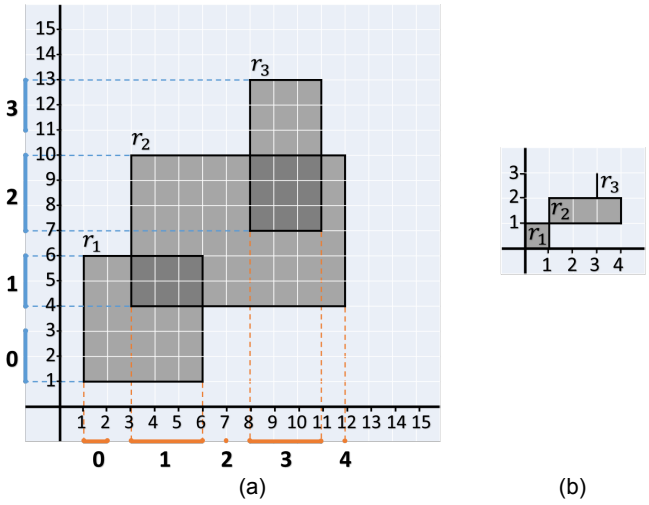

Figure 6: (a) Two dimensional ranges $R = \{[1,6] \times [1,6],\ [3,12] \times [4,10],\ [8,11] \times [7,13]\}$ and the segmentation to subranges in each axis. (b) The ranges after reducing the two dimensions $R' = \{[0,1] \times [0,1],\ [1,4] \times [1,2],\ [3,3] \times [2,3]\}$ using the subranges ids. Note that r_3 is reduced to just a vertical line.

function CALCCONVERSIONRANGES(R)
  for $i \in [k]$ **do**
    $AllPoints \leftarrow \emptyset$
    $MinPoints \leftarrow \emptyset$
    $MaxPoints \leftarrow \emptyset$
    $Map_i \leftarrow new\ integer\ array[]$
    $R_i \leftarrow numbered\ list$
    for $r \in R$ **do**
      $AllPoints.add(r[i].min)$
      $AllPoints.add(r[i].max)$
      $MinPoints.add(r[i].min)$
      $MaxPoints.add(r[i].max)$
    end for
    $low \leftarrow min(AllPoints)$
    for $p \in AllPoints.sorted()$ **do**
      if $p \in MinPoints$ **then**
        if $low < p$ **then**
          $R_i.add([low, p-1])$
        end if
        if $p \in MaxPoints$ **then**
          $R_i.add([p,p])$
          $low \leftarrow p+1$
          $Map_i[p] \leftarrow R_i.size() - 1$
        else:
          $low \leftarrow p$
          $Map_i[p] \leftarrow R_i.size()$
        end if
      else:
        $R_i.add([low, p])$
        $low \leftarrow p+1$
        $Map_i[p] \leftarrow R_i.size() - 1$
      end if
    end for
  end for
  return $\{(R_i, Map_i)\}_{i \in [k]}$
end function

4.2 Constructing a Reduced Policy

Once the reduction of each field is computed, and given the mapping $Map_i[p]$ explained in the previous subsection, we re-encode the original multi dimensional ranges to a reduced form. The re-encoding, done by the CalcReducedRanges function, simply replaces each original range r with the new encodings of all its endpoints(bounds), i.e., $r = r[1] \times r[2] \times \cdots \times r[k]$ is replaced with $r' = r'[1] \times r'[2] \times \cdots \times r'[k]$ where $r'[i] = [Map_i[r[i].min], Map_i[r[i].max]]$ (see example Figure 6).

function CALCREDUCEDRANGES($R, \{(R_i, Map_i)\}_{i \in [k]}$)
  $R' \leftarrow \emptyset$
  for $r \in R$ **do**
    $r' \leftarrow new\ range\ vector[k]$
    for $i \in [k]$ **do**
      $r'[i].min \leftarrow Map_i[r[i].min]$
      $r'[i].max \leftarrow Map_i[r[i].max]$
    end for
    $R'.add(r')$
  end for
  return R'
end function

Once all ranges are at reduced form, any k dimensional classification can be performed. We use the prefix expansion representation since it is easy to encode and to atomically update.

4.3 Classification Pipeline

As illustrated in Figure 7, the classification process of every packet utilizes one dimensional classifier per field and writes the matched sub range index number to a designated offset in the meta-

data field. Once all relevant fields have been converted, the reduced k dimensional classifier is queried with the metadata field to yield the final classification result.

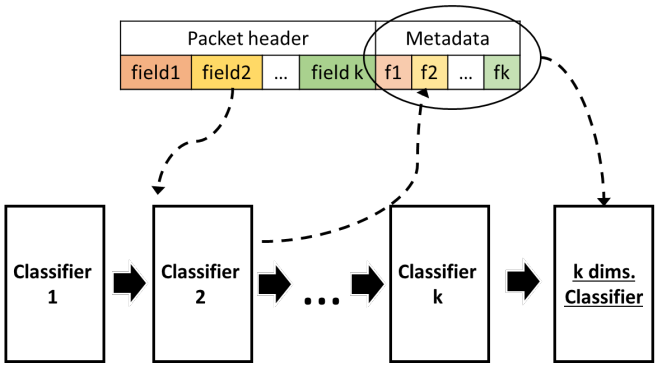

Figure 7: A packet processed by ORange-k. First each field is classified into one dimensional subranges and the subranges ids are written to metadata. Later, a k dimensional classification is performed based on the k subranges ids.

The classification pipeline can be followed by other flow tables implementing additional policies and reusing the metadata for other purposes.

4.4 Evaluation

The space consumed by each one dimensional classifier and k dimensions classifier is $2nw+4w^2$ and $O(n(2\log n)^k)$ respectively, where n is the number of ranges. Therefor the total space consumed by our scheme is $O(2knw+4kw^2+n(2\log n)^k)$ which is asymptoticly lower than the space used by current encoding schemes, $O(wn(2w)^k)$, as long as $n \ll 2^w$.

We extend the evaluation by introducing a simpler variant of our scheme, called Multi Dimensional Expansion (MUD-EXP), which uses simple prefix expansion for every one dimensional classifier. MUD-EXP resembles non-OpenFlow designs that use prefix expansion for one dimensional classifiers [12].

In Figure 8 we consider three datasets, the first is an ACL configuration of Cisco routers in real network which contains 303 two dimensional in-port and out-port rules where about 260 of them consider only one of the ports. The second dataset is a 1000 in-port and out-port rules (two dimensional) generated by classbench [18]. As a third dataset we randomly generated 1000 two dimensional range rules for 16bit numbers (fields).

Comparing the results of the different datasets we observe that cisco dataset requires much less space than the two other benchmark. This can be explained by the fact that the cisco dataset contains mostly one dimensional ranges which contributes few rules.

In Figure 9 we can see the performance of the schemes on random ranges with different dimenssions. As expected, the higher the dimension the bigger the space improvement. Moreover we can clearly see that on one dimensional ranges MUD-EXP doesn't help and even consumes more space than prefix expansion.

As can be seen in Figure 10, the field width has a strong impact on the space. At 32bits the expansion scheme reaches dosens of Mbs per rule which can overflow most switches, while the space consumption of our schemes are more than 2 orders of magnitude lower. Moreover as the width increases farther, the gap between our schemes is becoming more noticeable.

Our scheme trades space with pipeline length. The number flow

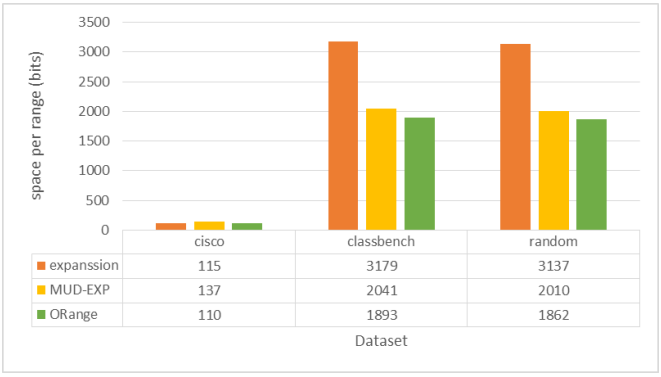

Figure 8: A comparison between the flow table space per range used by the three algorithms for three different datasets

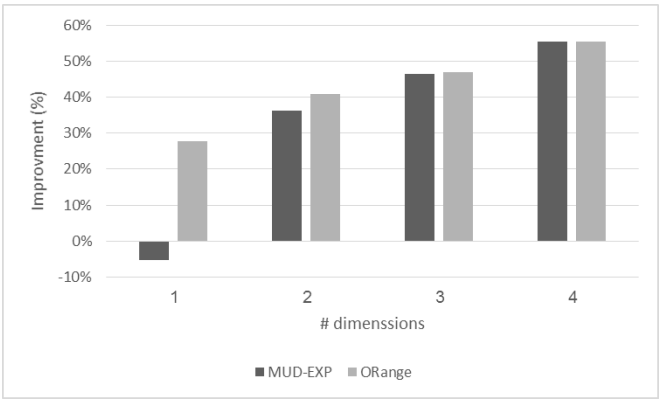

Figure 9: The space improvement of our algorithms vs. the dimension, in relation to the expansion encoding (at 0% as base line) for random datasets with $n = 1000$ and $w = 16$.

tables contributed by the one dimensional classifier and by the k dimensions classifier is 3 and $3k + 1$ respectively. Considering only a 4 microseconds delay per flow table (as measured by our experiments on NoviKit switch) we don't expect a noticeable increase in latency.

4.5 Atomic Updates for ORange-k

Here we present an update scheme that allows to change the multi dimensional classification policy while packets are processed by the switch. We consider the following operations; add rule, delete rule and update rule's actions. We require them to be atomic in the sense that every packet is correctly processed by either the old policy (prior to the operation) or the new policy (no policies mixture or artifacts should take effect) and also packets should be processed in consistent manner, once a packet is processed according to the new policy so should all the following packets.

When deleting a rule from the original policy we only have to delete a rule from the reduced (multi dimensional) policy and we do not have to update the one dimensional classifiers. Therefore deleting a rule from the original policy can be "reduced" to deleting a rule from the reduced policy. In turn, this can be seen as a general problem of any prefix expansion policy update, that requires to atomically delete multiple entries in a flow table. To support such an operation we utilize the OpenFlow ability to reference (modify

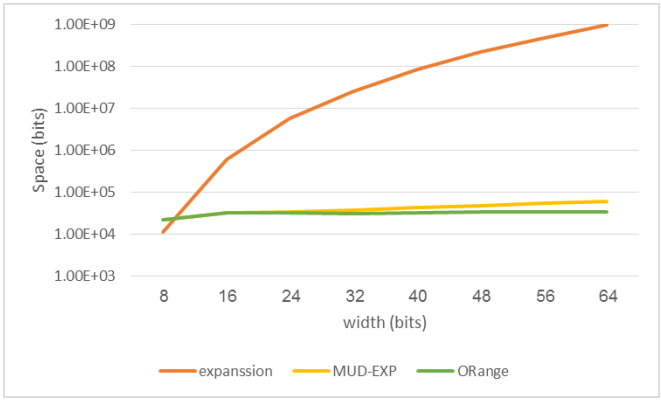

Figure 10: A comparison between the flow table space per range used by the three algorithms for random datasets of 100 4**-dimensional rules with varying word size.**

and delete) multiple flow entries according to an identifier called a **cookie** which is attached to each one of them. When we add entries in the fist time, we use the same unique cookie value to all entries of the same rule so that we delete them all with one (non-strict) delete command (see [1]).

Similarly, in order to update a rule action, we need to atomically change the action for multiple entries associated with that rule. To support such an update we suggest to make all the entries use indirect action (e.g., applying the same indirect group as an action) thereby allowing to change the action in a single OpenFlow modification request.

Adding a rule may require to update the one dimensional classifiers in order to provide new one dimensional sub ranges ids. These one dimensional updates do not have to be performed at the same time while updating the reduced k dimensional policy, and the overall process can be carried in two stages; first updating the one dimensional classifiers and secondly using the new subranges ids to express the reduced version of the added rules. Updating the one dimensional classifiers can safely be performed using the atomic modifiers (mostly $SplitSnadCreate$) described in Subsection 3.5.

Adding a rule also requires to add multiple entries to the table of the reduced policy, and we suggest to do so in the following way. First we add the entries one by one with lowest priority possible (to avoid interference with some other overlapping rule entries), all of them with the same unique cookie value (as in deleting) and with an indirect action that does nothing. Later we update the all entries with one modify command (matching the cookie) to higher priority and with different indirect action as desired.

Note that if all rules have the same priority and we can ignore rules overlaps (since there are no overlapping rules or at least that overlapping rules have the same action) then a much simpler scheme can be used. For adding a rule we can first add the entries one by one using an indirect action (as in deletion) that does nothing and lower priority. Later we update the action to the desired one and we update the priority of each entry (possibly one by one) to the "normal" priority. Similarly, deleting a rule can be done by simply making decreasing the priority of all entries (possibly one by one), then setting the indirect action to do nothing and then to delete each entry one by one.

Moreover, note that allocating new one dimensional subranges ids requires either to renumber existing subranges or to sparsely number the subranges in advance leaving enough space for new ones to come. While the first option can lead to mass updates in the reduced k dimensional classifier (since some reduced ranges will need to be re-encoded), the second leads to use wider ids thereby increasing the overall space complexity. Therefore, there is a tradeof between space complexity and the number of additions supported till a mass update is required, therby making the scheme more optimized for small number of additions or for scenarios where additions and similar deletions distribute evenly in the k dimensional space.

5. CONSISTENCY

Here we discuss consistency requirements that considers the effect of policy (range rule) update on flows of packets and not a single packet as atomicity, where a flow is usually a TCP session but can be any other set of packets defined by header field values. We focus here on the ORange1 classifier but the techniques can be easily adapted to the ORange-k as well.

5.1 Per Flow Consistency

In most scenarios (such as in load balancers), a configuration update shouldn't affect pre-existing flows in the network. Therefore we extend our technique to guarantee flow consistency, which means that all packets of a flow are handled according to the same configuration and new flows are handled according to the newest configuration.

The only base operation that we should extend is range update. Our extended scheme (code of which is given below) follows the ideas presented by Wang et al. [19]; Assume the update, updates the action associated with a sub-range x of range R whose action was "old", changing it to "new". The controller monitors the flows that belong to the updated sub-range by changing the action of this sub-range x to send all packets in x to the controller. Then for each flow that has existed before the update (a non syn packet in the case of TCP) a specific rule that traps this flow is created by the controller and the "old" action is associated with that new rule, and for each new flow a specific rule with the "new" action is created. We do this until all existing flows are reported to the controller (after predefined timeout e.g., TCP timeout) and then we change the action of the sub-range to the new configuration. Finally we delete all the specific rules created for new flows. Thus we are left with specific rules for the pre-existing flows, and a new sub-range x with the "new" action.

function PERFLOW-RANGEUPDATE(R, old-acts, new-acts)
change controller's message handler to PerFlow-MsgHndlr
replace the action in R's entry at RIDs to send to controller
wait for TCP-timeout seconds
▷ to learn all flows in sub-range
replace the action in R's entry at RIDs to new-acts
delete all specific rules created with new-acts
end function
function PERFLOW-MSGHNDLR(message)
if message is SYN **then**
▷ new flow
create specific rule in 1-ELCPs that match message's flow and new-acts
else
▷ old flow
create specific rule in 1-ELCPs that match message's flow and old-acts
end if
end function

5.2 Cross-Entrance Consistency

Per-flow consistent policy update in a network was shown, based on [19] in the previous subsection. However in a network with multiple entry points, changes in the external network (e.g., the Internet) can cause part of the traffic to change route (e.g., route balancing) and enter our network from a different entry point.

Keeping the rules that affect a flow consistent regardless of entrance point through which flows enter is not easy [19], and we name it "Cross Entrance Consistency". To solve it we need to extend the range classifier scheme to keep the different entrances synchronized during a range update. The cross entrance consistent update is similar to the per flow consistent update algorithm, with the main difference being that the controller installs the specific (per flow) rules in all entrance points. The complete pseudo code is given below .

```
function CROSSENT-RANGEUPDATE(R, old-acts, new-acts)
    change controller's message handler to CrossEnt-MsgHndlr
    for router ∈ Entries do
        replace the action in R's entry at RIDs to send to controller
    end for
    wait for TCP-timeout seconds
                                        ▷ to learn all flows in sub-range
    for router ∈ Entries do
        replace the action in R's entry at RIDs to new-acts
        delete all specific rules created with new-acts
    end for
end function

function CROSSENT-MSGHNDLR(message)
    if message is SYN then                                  ▷ new flow
        for router ∈ Entries do
            create specific rule in 1-ELCPs that match message's flow and new-acts
        end for
        forward the message according to new-acts from message.router
    else                                                    ▷ old flow
        for router ∈ Entries do
            create specific rule in 1-ELCPs that match message's flow and old-acts
        end for
        forward the message according to old-acts from message.router
    end if
end function
```

5.3 DevoFlow based Implementation

Here we discuss a more efficient way (in terms of control messages complexity) to implement per-flow and cross-entrance update, using DevoFlow [10]. DevoFlow defines a new type of rules (a Devo Rule), these rules serve as macros where a packet's match triggers the creation of specific rule which resembles the macro in term of associated action, and pattern with the exception that don't care bits in the macro rule are set to the corresponding specific bit values in the packet (and all other packets in the same flow).

The DevoFlow extension allows the controller to easily set a high level policy by creating a DevoRule and also to monitor the specific flows by inspecting the specific rules that have been created by the DevoRule and their statistics. This is done without controller intervention during the creation of each flow which saves time and control traffic. As noted by Reitblatt et. al [14], this extension can be used to achieve per flow consistency, where a policy change is translated to DevoRule change and existing flows remain in their existing policy due to their specific rules. In this way the controller shouldn't be informed and intervene with every flow creation.

A similar technique can be used with our scheme to achieve per flow consistency, when we update the action of a range, we first define its rules (still with the "old" action) in flow Tables 0-ELCPs and 1-ELCPs as DevoRules. As a result specific rules are created for existing flows and after a specified period of time (by which all existing flows have a corresponding specific rule) we change the rules back to normal (non devoflow) but with a new range id which is associated with the new action for the range. Note that entries in the RIDS table can be defined with timeouts so unused ids will be deleted automatically. This updated scheme saves control traffic and avoids the waiting step in our original per flow consistent scheme.

In order to achieve cross entrance consistency we use a DevoRule that in addition to creating a specific rule on a matching packet, it also sends a notification to the controller about the newly created rule. Given this DevoRule, the controller is informed about any flow creation at some network entrance point and it can duplicate the same specific rule at all other entrances. In this way even if the flow changes its entrance point, it is still considered as an old flow and not as a new one.

Our scheme for updating the action of a range x from old to new, while ensuring cross entrance consistency, has four stages; first we install DevoRules for range x associated with the old action in addition the DevoRule sends a copy of each created specific rule to the controller which then duplicates this specific rule in all other entrances. We wait in this stage until all existing flows in x are discovered and synced across all entrances (the entrances are continually being synced with new flows). A reasonable time to wait for flows discovery is of magnitude of TCP-timeout.

Then we begin the second stage where we replace the DevoRules with simple rules to send everything in sub-range x (that hasn't been already trapped by one of the specific rules) to the controller. When new flows are discovered in this stage the controller assigns the "new" policy rules to them directly and on all entrances.

In the third stage, the controller defines simple rules that forward (flows in sub-range x) according to the new policy while old flows are treated by their specific rules that were installed in the first stage, according to the old policy. In the forth and last stage, the controller deletes the specific rules created during the second stage and uses the new policy (these rules are no longer needed as a more general rules forward according to the same policy).

Notice, that sending all packets to the controller is required only during the second stage while defining the "send to controller" rules on all entrances. Therefore the time period in which high volume traffic is expected at the controller is relatively short, and in the case of TCP, only one packet per new flow is sent, since while the SYN packet is delivered and processed by the controller there is no further transmission of data in that flow.

6. CONCLUSIONS

This paper presents an efficient and dynamic range classifier implementation using basic OpenFlow features. The paper also considers multi entrance networks and show how policies (e.g., ranges) can be updated in such networks while keeping per flow consistency.

Acknowledgments. We thank Yotam Harchol for providing realtime NoviKit switch measurements, as well as Hanan R. Haim, Elad Levi, Michal Shagam and Dekel Auster for verifying and utilizing our scheme in the Mininet environment, and the anonymous referees for their comments.

7. REFERENCES

[1] *OpenFlow Switch Specification 1.3.3*.

[2] Orange github repository. `https://github.com/lironsc/ORange`.

[3] Ryu: Component-based software defined networking framework. `http://osrg.github.io/ryu/`.

[4] Y. Afek, A. Bremler-Barr, and L. Schiff. Cross-entrance consistent range classifier with openflow. In *Open Networking Summit 2014 (ONS 2014)*, Santa Clara, CA, 2014. USENIX.

[5] Y. Afek, A. Bremler-Barr, and L. Schiff. Ranges and cross-entrance consistency with openflow. In *Poster session in the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 233–234, New York, NY, USA, 2014. ACM.

[6] T. Banerjee, S. Sahni, and G. Seetharaman. Pc-trio: A power efficient tcam architecture for packet classifiers. *Computers, IEEE Transactions on*, PP(99):1–1, 2014.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, Aug. 2013.

[8] A. Bremler-Barr, D. Hay, and D. Hendler. Layered Interval Codes for TCAM-Based Classification. In *IEEE INFOCOM*, pages 1305–1313, 2009.

[9] H. Che, Z. Wang, K. Zheng, and B. Liu. Dres: Dynamic range encoding scheme for tcam coprocessors. *Computers, IEEE Transactions on*, 57(7):902–915, July 2008.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, Aug. 2011.

[11] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[12] C. R. Meiners, A. X. Liu, and E. Torng. Topological transformation approaches to tcam-based packet classification. *IEEE/ACM Trans. Netw.*, 19(1):237–250, Feb. 2011.

[13] R. Panigrahy and S. Sharma. Sorting and searching using ternary cams. *IEEE Micro*, 23:44–53, January 2003.

[14] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[15] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat. On finding an optimal tcam encoding scheme for packet classification. In *INFOCOM*, pages 2049–2057. IEEE, 2013.

[16] T. Sasao. On the complexity of classification functions. In *ISMVL 2008. 38th International Symposium on Multiple Valued Logic, 2008.*, pages 57–63, May 2008.

[17] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, Oct. 1998.

[18] D. E. Taylor and J. S. Turner. ClassBench: a packet classification benchmark. In *IEEE INFOCOM*, volume 3, pages 2068–2079, 2005.

[19] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

APPENDIX

A. CORRECTNESS OF ORange1 ATOMIC UPDATES

We consider queried values and range boundaries as integers in $\{0, .., 2^w - 1\}$, and patterns as ternary strings of length w. We denote the boundaries of range R by $R.start$ and $R.end$, i.e., $R = \{R.start, ..., R.end\}$. Each range R is associated with an id by $R.id$ and also an action $R.Action$ that should be performed to packets classified to that range.

We denote the 0-ELCP and 1-ELCP patterns of range R (as defined in Section 3.1) by $e_0(R)$ and $e_1(R)$ respectively. Since the ELCP patterns are prefix ones, we can treat them as ranges (e.g., using operators $\in$ and $|\cdot|$). We extend the definition of ELCP patterns to single value ranges: Given $v \in \{0, .., 2^w - 1\}$, the ELCP patterns of v are both v itself.

Given an 0-ELCP (1-ELCP) pattern p in the classifier, we denote the lower (upper) bound that is associated with p in the classifier by $p.start$ ($p.end$) and denote the range id that is associated with p by $p.rid$. Note that it is desired that for any range R in the classifier $R.end = e_1(R).end$, $R.start = e_0(R).start$ and $R.id = e_1(R).id = e_0(R).id$

DEFINITION A.1. *Given a set of non-overlapping ranges S, for $i \in \{0, 1\}$ we define $E_i(S)$ be the set of their i-ELCP patterns, i.e., $E_i(S) := \{e_i(r).r \in S\}$.*

DEFINITION A.2. *We define Longest Prefix Match, LPM, as a function that receives a set of patterns and a query value q, and returns the longest prefix pattern (fewest don't cares) in S that matches q, i.e., $LPM(S, q) := arg\min_{p \in S \wedge q \in p} |p|$.*

Note that LPM's outcome is equal to the result of a TCAM query that stores the patterns of S by the order of their prefix length (longer prefixes are stored before), and also equal to the result of Flow Table query where the patterns of S are stored in flow entries each with priority according to prefix length (longer prefixes have higher priorities).

DEFINITION A.3. *We define the outcome of processing a packet with field value q by our OpenFlow classifier as the action performed by the $RIDS$ table, and denote it by $OFC(S, RS, q)$, where S is the set of ranges loaded in the classifier and RS is the mapping from range id to action ($RIDS$ table). We denote a no action outcome (associated with no range result) by ϵ.*

We denote the action associated with a range id i in the matching entry in the RIDS table by $RIDS[i]$.

To prove atomicity we need to discuss the outcome at intermediate states. We define $ofc(FT_1, FT_0, RS)$ as the outcome of processing a packet with field value q by our OpenFlow classifier where the packet is processed by flow tables FT_1 and FT_0 as the 1-ELCPs and 0-ELCPs respectively, and RS as the RIDS table. Note that $OFC(S, RS, q) = ofc(E_1(S), E_0(S), RS, q)$.

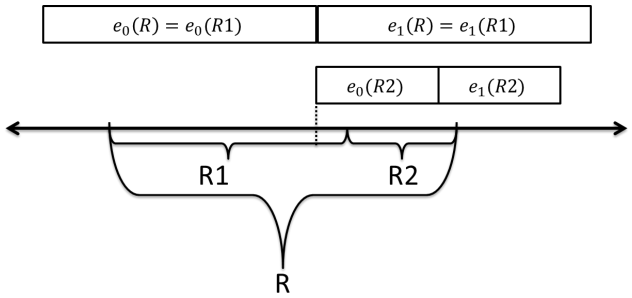

Figure 11: The 1st case of $Split(R, R_1, R_2)$ pseudo code, R_1 has the same ELCPs as R. This is a mirror to the 2nd case.

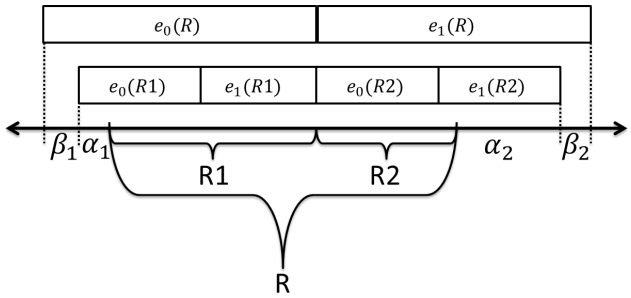

Figure 12: The 3rd case of $Split(R, R_1, R_2)$ pseudo code, the ELCPs of R_1 and R_2 are partial of those of R.

Following the description of the classifier in Section 3.2 and the definitions above, we obtain the expression for $ofc(FT_1, FT_0, RS, q)$.

COROLLARY A.4.

$$ofc(FT_1, FT_0, RS, q) = \begin{cases} RS[LPM(FT_1, q).rid] \\ \quad if\, q \leq LPM(FT_1, q).end \\ RS[LPM(FT_0, q).rid] \\ \quad if\, q > LPM(FT_1, q).end \\ \quad and\, q \geq LPM(FT_0, q).start \\ \epsilon \;\; - \; otherwise \end{cases}$$

LEMMA A.5. *Given a set of non-overlapping ranges, S, for any query q, if q belongs to range $r \in S$ then $OFC(S, RS, q) = RS[r.id]$ otherwise $OFC(S, RS, q) = \epsilon$. This means that our classification scheme is correct for the static case.*

PROOF. The correction of our scheme for the static case is derived from the correction of the original PIDR algorithm described in Section 3.1. □

LEMMA A.6. *Given a set of non-overlapping ranges, S, for any query q, for any step in the execution of a $Split(R, R_1, R_2)$, if q belongs to range $r \in S \cup \{R_1, R_2\}$ then $ofc(FT_1, FT_0, RS', q) = RS[r.id]$, otherwise $ofc(FT_1, FT_0, RS', q) = \epsilon$, where $R \in S$ and $R = R_1 \cup R_2$, and FT_1, FT_0, RS' are the $1 - ELCP, 0 - ELCP, RIDS$ tables during the Split. This means that our classification scheme is correct for the static case.*

PROOF. Note that FT_1, FT_0 start from $E_1(S), E_0(S)$ and end at $E_1(S'), E_0(S')$ respectively, where $S' = (S/\{R\}) \cup \{R_1, R_2\}$. We divide the proof to three cases, according to the pseudo code of Split.

Case 1: If $e_1(R_1) = e_1(R)$ (also means that $e_0(R_1) = e_0(R)$) then we first add R_2 to the RIDS table which has no effect on the outcome. Then we add $e_i(R_2)$ to FT_i. Lets choose to add $e_1(R_2)$ first and consider the state $FT_1^{R_2}$ of table FT_1 after this addition. Note that $e_1(R_2) \subset e_1(R)$. For any q if $q \in e_1(R_2)$ then $q \notin e_0(R)$ and $q \notin e_0(R_2)$ and therefor $LPM(FT_0, q).id \notin \{R.id, R_2.id\}$. Moreover since $e_1(R_2) \subset e_1(R)$ then $LPM(FT_1^{R_2}, q).id \neq R.id$. This means that for $q \in e_1(R_2)$, R is not seen at all and we have the same outcome as if R_2 was fully added and instead of R, R_1 was added (R and R_1 share the same ELCPs), i.e., $ofc(FT_1^{R_2}, FT_0, RS', q) = OFC(S_2, RS', q)$ where $S_2 = (S/\{R\}) \cup \{R_1, R_2\}$ which means we have a correct outcome. If $q \notin e_1(R_2)$ then the outcome remains as if $e_1(R_2)$ wasn't added and the outcome is correct, i.e., $ofc(FT_1^{R_2}, FT_0, RS', q) = ofc(FT_1, FT_0, RS', q) = OFC(S, RS', q)$. Next we add $e_0(R_2)$ to FT_0 resulting with $FT_0^{R_2}$. Now R_2 is fully added, the ELCPs match the state where R_1 is added too but the lower bound of R_1 is as R. This different may affect only items in R_2 or R_1 but since both have the same action the outcome will be correct.

Case 2: If $e_1(R_2) = e_1(R)$, this is a complete mirror case of the former and therefore the scheme is correct in this case as well.

Case 3: If $e_1(R_2) \subset e_1(R)$ and $e_0(R_1) \subset e_0(R)$ (and also $e_0(R_2) \subset e_1(R)$ and $e_1(R_1) \subset e_0(R)$) then we add the ELCPs of R_1 and R_2 and later we delete R. We first consider all the intermediate states during the additions of R_1 and R_2 that includes the addition of new 4 non-overlapping ELCPs and therefor there is no importance to order of their addition. Note that all values inside R are correctly treated since we add contention between the ELCPs of R_1 (R_2) and R for values that are inside R_1 (R_2) so the correct bounds are checked and the same action is returned. Also note that values beyond the ELCPs of R are not affected at all and their outcome is correct as well. We divide the remaining possible values to 4 types: α_1 are values outside R but inside $e_0(R_1)$, i.e., $\alpha_1 = e_0(R_1)/R$.α_2 are values outside R but inside $e_1(R_2)$, i.e., $\alpha_2 = e_1(R_2)/R$. β_1 are values inside $e_0(R)$ but outside the ELCPs of R_1, i.e., $\beta_1 = e_0(R)/(e_0(R_1) \cup e_1(R_1))$.$\beta_2$ are values inside $e_1(R)$ but outside the ELCPs of R_2, i.e., $\beta_2 = e_1(R)/(e_0(R_2) \cup e_1(R_2))$. Due to symmetry it's enough to prove the correctness only for values in α_1 and β_1.

Values in α_1 can't be effected by R and has the same (correct) outcome as if R was already deleted. Values in β_1 are not effected by R_1 and has the same (correct) outcome as if R_1 wasn't added at all.

In conclusion for all values of q the outcome is correct in every step of adding R_1 and R_2. Finally the deletion of R only effects the β_i groups, but this effect only lower the possibility of value outside R to be match by an ELCP of R and therefor for any order of deleting the ELCPs of R the correctness is preserved as well. □

LEMMA A.7. *Given a set of non-overlapping ranges, S, for any query q, for any step in the execution of a $Merge(R_1, R_2)$, if q belongs to range $r \in S \cup \{R_1 \cup R_2\}$ then $ofc(FT_1, FT_0, RS', q) = RS[r.id]$, otherwise $ofc(FT_1, FT_0, RS', q) = \epsilon$, where R_1 and R_2 are adjacent ranges in S and FT_1, FT_0, RS' are the $1-ELCP$, $0 - ELCP$,$RIDS$ tables during the Merge.*

PROOF. The steps made by the Merge procedure are reversed to the steps made by the Split. Therefor all intermediate states of Merge are also intermediate states of Split. Since the Split is correct so is the Merge. □