Abstract

One of the most critical resource management issues in the use of ternary content addressable memory (TCAM) for packet classification/filtering is how to effectively support filtering rules with ranges, known as range matching. In this paper, a Dynamic Range Encoding Scheme (DRES) is proposed to significantly improve TCAM storage efficiency for range matching. Unlike the existing range encoding schemes requiring additional hardware support, DRES uses the TCAM coprocessor itself to assist range encoding. Hence, DRES can be readily programmed in a network processor using a TCAM coprocessor for packet classification. A salient feature of DRES is its ability to allow a subset of ranges to be encoded and hence to have full control over the range code size. This feature allows DRES to exploit the TCAM structure to maximize TCAM storage efficiency. DRES is a comprehensive solution, including a dynamic range selection algorithm, a search key encoding scheme, a range encoding scheme, and a dynamic encoded range update algorithm. While the dynamic range selection algorithm running in software allows optimal selection of ranges to be encoded to maximize the TCAM storage efficiency, the dynamic encoded range update algorithm allows the TCAM database to be updated lock-free without interrupting the TCAM database lookup process. DRES is evaluated based on real-world databases and the results show that DRES can reduce the TCAM storage expansion ratio from 6.20 to 1.23. The performance analysis of DRES based on a probabilistic model demonstrates that DRES significantly improves TCAM storage efficiency for a wide spectrum of range distributions.

Key Words: Packet Classification, Range Matching, Ternary CAM, Network Processor
I. Introduction

Packet classification has long been recognized as a critical data path function for high speed packet forwarding in a router. To keep up with multi-gigabit line rates, a high performance router needs to be able to classify a packet in a few tens of nanoseconds (\(\text{ns}\)). In the past few years, significant research efforts have been made to design fast packet classification algorithms for both Longest Prefix Matching (LPM) and general policy/firewall filtering (PF) \[2\] \[5\] \[8\] \[9\] \[18\] \[19\] \[20\] \[22\]. However, most of these algorithmic approaches cannot provide deterministic lookup performance that matches multi-gigabit line rates.

An alternative approach, which has been gaining popularity, is the use of a ternary content addressable memory (TCAM) coprocessor for fast packet classification. In general, a TCAM coprocessor works as a look-aside processor for packet classification on behalf of a network processing unit (NPU) or network processor. When a packet is to be classified, an NPU generates a search key based on the information extracted from the packet header and passes it to the TCAM coprocessor for classification. A TCAM coprocessor finds a matched rule in \(O(1)\) clock cycle and therefore offers the highest possible lookup/matching performance \[7\]. Indeed, packet processing at OC-192 line rate using an integrated NPU and TCAM coprocessor solution has been reported \[1\].

However, despite its fast lookup performance, the TCAM-based solution has its own shortcomings, including high power consumption, large footprint, and high cost. These shortcomings directly contribute to a critical issue for packet classification using TCAM, namely, supporting rules with ranges, or range matching. The difficulty lies in the fact that multiple TCAM entries have to be allocated to represent a rule with ranges. Today’s real-world PF tables were reported \[2\] \[8\] \[12\] \[17\] to involve significant amounts of rules with ranges. In particular, our statistical analysis of real-word rule databases shows that TCAM storage efficiency can be as low as 16% due to the existence of a significant number of rules with port ranges. Apparently, the reduced TCAM memory efficiency due to range matching makes TCAM power consumption, footprint, and cost even more serious concerns.

A widely adopted solution to deal with range matching is to do a range preprocessing/encoding by mapping ranges to a short sequence of encoded bits, known as bit-mapping. The idea is to view a \(d\)-bit rule as a region in a \(d\)-dimensional rule space and encode any distinct overlapped regions among all the rules so that each rule can be translated into a sequence of encoded bits, known as rule encoding. Accordingly, a search key that is based on the information extracted from the packet header is preprocessed to generate
an encoded search key, called `search key encoding`. Then the encoded search key is matched against all the encoded rules to find the best matched rule. Unlike rule encoding which can be pre-processed in software, search key encoding is performed on a per packet basis and must be done in hardware at wire-speed. For this reason, a key differentiator of range-encoding schemes is the approach taken to do the search key encoding.

Bit-map based range encoding for packet classification was originally proposed by Lakshman and Stiliadis [13]. The application of the bit-map-based range encoding for packet classification using a TCAM has also been reported [14] [15] [16]. In particular, the rule encoding schemes proposed by Lunteran and Engbersen [15] [16] are the most effective schemes, as they can encode $N$ non-overlapping ranges using only $\log_2(N + 1)$ bits.

All the existing range encoding approaches, including the aforementioned ones, are top-down approaches. Namely, they strive to design the most efficient range encoding schemes and make assumptions on the existence of needed hardware to support the search key encoding at high speed. For example, the search key encoding approaches proposed in [13] [15] [16] assume the existence of multiple processors and/or multiple memories for parallel search key encoding. Another example of the top-down approach is the work by Spitznagel, Taylor, and Turner [17] that addresses TCAM power and range matching issues by designing a new TCAM architecture. What is missing is an approach from the bottom up, i.e., designing range-encoding schemes subject to the hardware availability of the existing TCAM-based packet classification solutions, e.g., commercial TCAM coprocessors. A bottom-up approach may not provide the most efficient solutions for range encoding but it ensures that when applied to an existing TCAM-based packet classifier, it will work with only software upgrades.

The research based on a top-down approach is of great importance because it provides insights, directions, or even solutions on how the next-generation TCAM-based packet classifiers should be designed. However, it is equally important to address the range matching issue from the bottom up. A bottom-up approach is of paramount importance for any system vendors who use a third-party TCAM coprocessor in their system design. For these TCAM users, a range-encoding scheme should not only significantly improve TCAM storage efficiency but also leave the existing hardware unchanged. As NPUs and TCAM coprocessors are fully integrated for major NPU vendor solutions and widely used by system vendors, including Intel IXP2xxx series [10] and AMCC nP7xxx series [6] [11], there is an increasingly pressing need to develop
efficient bottom-up range encoding schemes that can be immediately programmed in any NPUs using a TCAM coprocessor for packet classification. Unfortunately, to the best of our knowledge, no such a solution exists today. To ensure wide applicability, a bottom-up approach must satisfy the following requirements:

1) It generally applies to any existing integrated NPU and TCAM coprocessor solutions without additional hardware support;
2) The algorithm design must take the TCAM coprocessor memory structure into account;
3) The search key encoding must be fast enough to keep up with the wire-speed and the TCAM storage efficiency must be significantly improved;
4) The encoded range update process must not cause significant performance degradation to the PF table lookup process.

In this paper, we propose a Dynamic Range Encoding Scheme (DRES). DRES is a bottom-up approach and meets all the above requirements, and therefore it can be readily programmed in any NPU using a TCAM coprocessor for packet classification. One salient feature of DRES is its ability to allow a subset of ranges from any field to be encoded. This allows DRES to have full control over the encoded rule size and exploit the TCAM structure to maximize the encoding gain. The idea of encoding only a subset of ranges is in line with the observation that in today’s real-world PF databases, a small number of ranges are found to frequently appear in a large number of rules, which leads to the low TCAM storage efficiency. For example, in the four real-world databases we studied (See Section V), the range \( \{ > 1023 \} \) that occupies six TCAM entries is the most popular range and it constitutes about 88% of all ranges. Hence, encoding this range alone can significantly improve the TCAM storage efficiency.

Similar to the existing top-down approaches, DRES uses the bit-map technique for range encoding. Therefore, the existing bit-map schemes are fully leveraged and incorporated in the design of the proposed range encoding scheme. To ensure the applicability of DRES for any existing NPUs and TCAM coprocessors, DRES uses the TCAM coprocessor itself for search key encoding. DRES provides a range selection algorithm to allow dynamic selection of ranges for encoding so that the maximum encoding gain is maintained whenever the rules are updated. Moreover, we present the design of a lock-free encoded range update algorithm that allows uninterrupted search key encoding and rule matching during the update process. Statistical analyses on both real-world PF samples and a probabilistic model demonstrate that DRES can significantly improve TCAM storage efficiency in support of range matching.
Finally, we note that, although primarily designed to be a bottom-up approach, most of the ideas developed in DRES can be leveraged in the design of top-down approaches as well. For example, a variation of DRES with parallel search key encoding is successfully adopted in the design of a distributed TCAM-based packet classifier [23] that matches OC-768 line rate.

The rest of the paper is organized as follows. Section II describes how the rules are expressed in a TCAM. We give a detailed description of DRES in Section III and Section IV. Section III describes the dynamic range selection algorithm and the range encoding scheme. Section IV details the encoded range update procedure. In Section V, we evaluate the performance of DRES based on real-world databases. Section VI analyzes the performance of the DRES based on a probabilistic model. Finally, Section VII concludes the paper.

II. RULE IMPLEMENTATION IN TCAM

Figure 1 gives a logic diagram showing how an NPU works with its TCAM coprocessor. A TCAM coprocessor includes two pieces of memory in general, i.e., a TCAM and an associated memory (e.g., a SRAM). The TCAM is organized in slots. The slot size is fixed and may take different values for different vendor solutions, e.g., 64, 72, or 128 bits. A bit in each slot can take one of the three values: 0, 1 or ‘don’t care’, denoted as ‘*’. The rules in a PF table are placed in the TCAM and each rule entry (as will be defined shortly) maps to a memory address in the associated memory in which the corresponding action code is kept. A rule takes one or multiple slots. When a packet comes, the NPU generates a search key based on the packet header information and passes it to the TCAM coprocessor to be classified, via a NPU-TCAM coprocessor interface. A local CPU is in charge of rule table update through a separate CPU-TCAM coprocessor interface.

We use an example to guide the discussion throughout the rest of the paper. Consider a typical 5-tuple, 104-bit PF rule composed of the following 5 fields: \{source IP address, destination IP address, source port, destination port, protocol number\}. Each rule is associated with an action. Figure 2 gives 8 such rules $L_1, ..., L_8$ and their corresponding actions. A port number can be an exact number or a range and each distinct port number for a given port field is called a unique port for that port field. For example, \{> 1023\}, \{2047\}, \{256-512\}, and \{<1024\} are four unique source ports.

Figure 3 depicts what it looks like when rules $L_1$ and $L_2$ are placed in a TCAM with 64-bit slot size. $L_1$
Fig. 1. A network processor and its TCAM coprocessor.

Fig. 2. An example of 5-tuple rules. 'x' represents a wildcarded byte.

does not have a range in any of its fields and hence it takes 2 slots with 24 free bits left in the second slot. Each of such rules in the TCAM takes the minimum number of slots which is defined as a rule entry. $L_2$ has a range \{256-512\} in its destination port field. This range cannot be directly expressed in a TCAM and must be partitioned into two sub-ranges: \{256 - 511\} and \{512\} as shown in Figure 3. Hence, $L_2$ takes 4 slots (slots 3, 4, 5 and 6), or 2 rule entries in the TCAM.

To facilitate further discussion, a range is said to be exactly implemented in a TCAM if it is expressed in a TCAM without being encoded. A rule with an exactly implemented range therefore may take multiple TCAM rule entries, as is the case for $L_2$. As another example, six rule entries are needed to express range \{>1023\} in a TCAM, if the range is exactly implemented, as shown in Figure 4. Hence, $L_5$ takes 6 x 6
Fig. 3. Rules in a TCAM. The range \{256-512\} is split into 2 sub-ranges \{256-511\} and \{512\}, and implemented as sub-range 1 and sub-range 2. ‘*’ represents a ‘don’t care’ bit, and ‘x’=‘******’, a wildcard byte. The other numbers represent the actual byte values.

36 rule entries if it is exactly implemented. These ranges which can not be exactly implemented using one rule entry in a TCAM are called non-compact ranges. Other ranges that can be exactly implemented in one rule entry in a TCAM are called compact ranges. For example, range \{<1024\} in \(L_8\) is a compact range and it can be exactly implemented as 000000** ****** in a TCAM. To be exactly implemented, a non-compact range must be decomposed into a set of compact sub-ranges as is the case for \(L_2\). In what follows, we simply use range to represent non-compact range if it is not specified, but sub-range to stand for both compact and non-compact sub-ranges.

\[
\begin{align*}
0000 & 01** ***** ***** \\
0001 & **** ***** ***** \\
01** ***** ***** & 1*** ***** ****
\end{align*}
\]

Fig. 4. Range \{>1023\} is expressed in terms of 6 sub-ranges in a TCAM.

### III. Dynamic Range Encoding Scheme

A Dynamic Range Encoding Scheme (DRES) is proposed in this section. DRES is a comprehensive solution for range matching including a dynamic range selection algorithm, a range encoding algorithm and an encoded range update algorithm.
A. Dynamic Range Selection Algorithm

As we shall explain in more detail in Section III-C, for simplicity, we use the bit-map scheme in [13] [14] to encode ranges, i.e., each unique range is mapped to a unique bit. This scheme is referred to as the bit-map intersection scheme [16]. This scheme allows us to design an optimal dynamic range selection algorithm. The algorithm is run in software in the control plane.

Figure 5 gives a general range selection procedure for selecting \( m \) ranges for encoding out of \( n \) ranges. \( S_i \) is the number of sub-ranges needed to exactly implement range \( R_i \) \((i = 1, 2, ..., n)\) in a TCAM. \( E_i \) is the number of rule entries to implement all the rules which contain range \( R_i \). \( G_i \) is the encoding gain for \( R_i \), defined as the number of rule entries which can be eliminated if \( R_i \) is encoded. To select \( m \) ranges to be encoded, \( m \) steps are required, with each selecting one range. In the first step, the values of \( E \) and \( G \) for all the ranges are calculated assuming no range is selected for encoding. Then, the range with the maximum \( G \) is selected as the first range for encoding. Suppose \( R_1 \) is selected. In the second step, \( E \) and \( G \) for all the ranges except \( R_1 \) are updated, assuming that all the rules containing \( R_1 \) have \( R_1 \) encoded. Then the range with the maximum \( G \) is chosen to be the second encoded range. This procedure continues until \( m \) ranges are selected. The computational complexity for this algorithm is \( O(nm) \).

![Fig. 5. Dynamic range selection procedure. The tables are range encoding gain tables in each step.](image)

For example, in the PF table as shown in Figure 2, there are a total number of \( n = 7 \) ranges in both source and destination port fields. Ranges \( R_1 - R_5 \) come from the destination port field. They are: \( R_1 = \{256 - 512\}; R_2 = \{768 - 2047\}; R_3 = \{6000 - 6064\}; R_4 = \{>1023\} \) and \( R_5 = \{512 - 1536\} \). The rest of the two ranges come from the source port field, i.e., \( R_6 = \{>1023\} \) and \( R_7 = \{256 - 512\} \). Note that although \( R_1 \) and \( R_7 \) have the same range value, they are counted as two different ranges because they come from different fields. The same is true for \( R_4 \) and \( R_6 \). \( \{<1024\} \) in \( L_8 \) is a compact range and hence not counted.
as a range here. Suppose $m = 3$. Then the range encoding gain tables are given in Figure 6.

<table>
<thead>
<tr>
<th>Range</th>
<th>S</th>
<th>E</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>R₂</td>
<td>2</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>R₃</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>R₄</td>
<td>6</td>
<td>36</td>
<td>30</td>
</tr>
<tr>
<td>R₅</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>R₆</td>
<td>6</td>
<td>48</td>
<td>40</td>
</tr>
<tr>
<td>R₇</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Range</th>
<th>S</th>
<th>E</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>R₂</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R₃</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>R₄</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>R₅</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>R₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R₇</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Range</th>
<th>S</th>
<th>E</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>R₂</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R₃</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>R₄</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>R₅</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>R₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R₇</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Range</th>
<th>S</th>
<th>E</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R₂</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R₃</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R₄</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R₅</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R₆</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R₇</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 6. An example of range encoding gain tables. (a) Initial gain table; (b) After $R₆$ is chosen; (c) After $R₆$ and $R₄$ are chosen. (d) After $R₆$, $R₄$ and $R₁$ are chosen.

Initially, $E$ and $G$ as shown in Figure 6 (a) are calculated without any ranges being encoded. For instance, the encoding gain with respect to $R₆$ is calculated as follows. As stated in the previous section, six sub-ranges are needed to express $R₆$ (also $R₄$) in a TCAM. Since $L₅$ has $R₆$ in its source port and $R₄$ in its destination port, 36 rule entries are needed to exactly implement $L₅$. Similarly, 12 rule entries are needed to exactly implement $L₃$ because 2 sub-ranges are needed to express $R₂$. Thus a total of 48 entries are needed to exactly implement both $L₃$ and $L₅$, both having $R₆$ in their source port field. If $R₆$ is encoded, the six sub-ranges that represent $R₆$ in the TCAM are reduced to one encoded bit in an encoded rule. Then the numbers of rule entries that are required to express $L₅$ and $L₃$ are reduced to six and two, respectively. Hence, the gain for encoding $R₆$ is 40, the largest in column $G$ in Figure 6 (a). According to the range selection procedure, $R₆$ is selected to be encoded at this point. Then, the values of $E$ and $G$ for all the ranges except $R₆$ are updated, assuming $R₆$ is encoded. The results are shown in Figure 6 (b). Now $R₄$ is selected to be encoded since it has the largest $G$ value. Consequently, the values for $E$ and $G$ are updated again, assuming that $R₆$ and $R₄$ are encoded. The results are shown in Figure 6 (c). Since both $R₁$ and $R₃$ have the largest $G$ value of three, one of them is randomly selected (here $R₁$ is selected), then the encoding gain table is shown in (d).

When the rules with ranges are added or deleted, the encoding gain for some or all the ranges may be changed. Hence the above range selection procedure needs to be executed in software upon each rule update involving range changes. However, since a single rule update involving range changes cannot drastically
change the ranks of the existing ranges in terms of encoding gain, most of the executions of the selection procedure are not expected to lead to the encoded range updating. When encoded ranges need to be updated, both the encoded rule table and the range tables (as shall be explained shortly) must be updated.

B. Structures of Encoded Search key and Encoded Rule

A rule takes one or multiple slots in a TCAM depending on the rule size. Usually there are some free bits left for each rule entry. Instead of encoding all (e.g., [16]) or certain rule fields (e.g., [14]) (collectively called complete encoding approach in this paper), we have designed a hybrid encoding approach in DRES. This approach retains all the rule fields and makes use of those free bits to encode a subset of ranges to reduce the number of slots needed to express a rule with certain ranges. However, if there is no free bit or the number of free bits is too small, DRES allows extra slots to be allocated for each rule entry as long as the overall TCAM storage efficiency is improved.

On the surface, it appears to be advantageous to adopt a complete encoding approach rather than a hybrid approach, because the encoding gain can be improved not only by eliminating the ranges but also the rule fields to minimize the encoded rule size. However, in practice, this may not be true for a bottom-up design for three reasons. First, since we do not assume the availability of parallel search key encoding and we use the TCAM coprocessor to sequentially encode search key fields, it is impractical to encode all the rule fields due to the potential throughput penalty. Second, encoding the search key itself consumes TCAM resources due to the addition of a range table for each field to be encoded (see the next section for details), whose size is proportional to the number of ranges to be encoded. Hence, it is important to have full control over the range table size, i.e., what and how many ranges should be encoded. Third, the length of an encoded rule changes over time if some or all the fields are encoded and the actual length of an encoded rule is dependent on the rule database structure. In this case, negative encoding gain may occur, as we shall see in Section V. In our scheme, the ability to decide what ranges should be encoded allows for the full control of the encoded rule size, thus exploiting the available free bits for range encoding and ensuring positive encoding gain. Moreover, when a TCAM slot size is given, blindly striving to minimize the encoded rule size may not be worth the effort. For example, unless the encoded rule is guaranteed to fit into one slot, e.g., a 64-bit slot, it is wasted effort to encode a five-tuple rule database in an attempt to minimize the encoded rule size to improve TCAM storage efficiency.
A 5-tuple rule entry in a TCAM with 64-bit slot size takes two slots as shown in Figure 3. There are 24 free bits in the second slot. In DRES, the free bits are taken by a code vector. If a rule has no encoded range in any of its fields, the code vector is wildcarded and the rule itself remains unchanged. If there is an encoded range in any field in the rule, that field is wildcarded, and the corresponding code vector is encoded based on the encoding rules described in Section III-D. Hence, after a range is encoded, the rule with respect to that range field can be implemented with one rule entry, improving the TCAM storage efficiency.

The encoded search key has the same structure and length as the encoded rule. In DRES, an index vector is associated with each search key and takes over the free bits. Each bit in the index vector corresponds to an encoded range. If a search key has no field value which falls into any encoded range, the index vector in the search key is set to NULL, or has all 0 bits. If one or more fields of a search key fall into some encoded ranges, a non-zero index vector is associated with the search key. The index vector encoding scheme is given in Section III-D.

C. Search Key Encoding Process

Assume that \( m \) ranges from \( K \) fields in a rule have been selected for encoding. Then \( K \) separate range tables are needed for the search key encoding. The search key encoding involves \( K \) search key fields matching against the corresponding \( K \) range tables to get an index vector or an encoded search key. We refer to this process as the search key encoding process.

The search key encoding must be performed at wire-speed by the NPU for every packet to be classified. A match of a particular range in the \( k \)th range table (i.e., the value of the \( k \)th field in the search key falls into a range in the \( k \)th range table) causes an intermediate index vector to be returned. Upon receiving a returned intermediate index vector, the NPU updates an index vector (initially set to NULL) by performing an OR operation between the index vector and the returned intermediate index vector. This process continues until all \( K \) range table matches are performed. Next, the encoded search key is formed by appending the final index vector to the original search key. Finally, this encoded search key is used to match against the encoded rule table in TCAM. In summary, a PF table lookup with range encoding requires \( K \) range table lookups for search key encoding plus one encoded PF table lookup.

To achieve deterministic, one-memory-access performance per field encoding, a possible solution for the search key encoding is to perform direct range table indexing using an off-chip memory, as suggested in
[14]. Namely, allow the $k$th field’s size worth of memory entries to be allocated for the $k$th range table. Each entry contains an intermediate index vector corresponding to the range the memory address of the entry falls into. This, however, works only for fields with small sizes, e.g., no more than 20 bits. For example, to support a 32-bit field range table with an index vector size equal to 64 bits, $2^{32} \times 64$ bits of memory need to be allocated, which is prohibitively large. Moreover, a range update in a range table requires all the memory entries to be updated in the worst-case, causing significant interruption to the search key encoding process. To ensure the applicability of DRES in general, we propose to use the TCAM coprocessor itself for the search key encoding. This involves adding $K$ range tables and the associated intermediate index vectors to the TCAM and the associated memory, respectively.

Fig. 7 describes how the TCAM and the associated memory in a TCAM coprocessor are partitioned to enable search key encoding. In the $k$th range table, $m_k$ ranges are listed. Each range in the range table maps to an intermediate index vector in the associated memory. Note that since each range to be encoded is composed of multiple sub-ranges, and hence, each range must be represented by multiple TCAM entries (not shown in Fig. 7), the corresponding intermediate index vector must be duplicated for every entry belonging to the same range. In the encoded PF table, a code vector (cv) is appended to each rule to form an encoded rule. Each encoded rule is mapped to an action in the associated memory.

Using TCAM for search key encoding provides an immediate solution applicable to any NPU using a TCAM coprocessor for packet classification. However, heavy use of the TCAM coprocessor for the search key encoding may result in reduced packet classification performance due to TCAM access contention. Here we resolve this concern by giving an estimation of the TCAM throughput performance for DRES. Assume that the TCAM runs at 100 $MHz^1$, i.e., 100 million lookups per second. For wire-speed forwarding at OC-192 line rate, up to 25 million packets need to be classified in one second in the worst-case. Thus each packet is allowed to have $100/25=4$ TCAM lookups. If a rule entry takes $s$ slots, a search key matching against a PF table requires $s$ lookups. For a 5-tuple rule, two lookups are needed, assuming that the slot size is 64 bits. Further, assume the size of any field with encoded range is no larger than the size of a slot, i.e., 64 bits. Then each range table matching for search key encoding requires one TCAM lookup. Now, if both the source and destination port fields have ranges to be encoded, i.e., $K = 2$, each PF table matching

---

$^1$Today the TCAMs provided by Ayama and IDT have the fastest speeds of up to 133 $MHz$. We use 100 $MHz$ TCAMs as examples in this paper because they are widely used in industry.
Step K+1: 
Encoded Search Key

Step 1: 
Encoded Sub-field 1

Step K: 
Encoded Sub-field K

Fig. 7. Encoding process: \( K + 1 \) TCAM searches for PF table matching in a TCAM coprocessor. 'cv' stands for code vector.

requires four TCAM lookups. In this case, the wire-speed forwarding at OC-192 line rate can be achieved. For an NPU supporting OC-48 line rate, each packet is allowed to perform 16 lookups. In this case, all five fields can be encoded if necessary. Hence, when using the scheme proposed in this paper, care must be taken as to how many rule fields can be selected for range encoding. Encoding more fields leads to heavier TCAM access contention, although better TCAM storage efficiency. Fortunately, for the typical 5-tuple PF, ranges only appear in the source and destination port fields, resulting in no more than 4 TCAM lookups per PF table matching, making DRES a viable solution matching multi-gigabit line rates.

D. Code Vector and Index Vector Encoding Scheme

In this paper, the bit-map schemes developed in [13] [14] [15] [16] are fully leveraged for code vector and index vector encoding. The most efficient bit-map scheme is the \( \mathbb{P}2\mathbb{C} \) scheme proposed in [16], which allows \( N \) ranges to be encoded using only \( \log_2(N + 1) \) bits in the best-case, when the ranges do not overlap with one another. The scheme requires \( N \) bits to encode \( N \) ranges in the worst-case, when any range may overlap with any other range. It can be shown that the dynamic range selection problem for \( \mathbb{P}2\mathbb{C} \) scheme can be formulated as a weighted knapsack problem, which is NP-complete, and hence, a suboptimal heuristic
algorithm must be sought for dynamic range selection. Moreover, it is much easier to design a lock-free encoded range update algorithm for the bit-map intersection scheme than P\textsuperscript{2}C. Since the number of bits needed to encode a given number of ranges for P\textsuperscript{2}C scheme in the worst case is the same as that in the bit-map intersection scheme proposed in [13], we simply choose bit-map intersection scheme to test the worst case performance bound for DRES. The performance using P\textsuperscript{2}C scheme is analytically evaluated in Section VI. Note that when incorporating either of these encoding schemes in DRES, they have to be used in the context of the hybrid encoding approach, i.e., encoding a subset of ranges. Hence, in what follows, we simply refer to them when used in DRES as the Hybrid Encoding approach with Bit-map Intersection (HE-BI) and the Hybrid Encoding approach with P\textsuperscript{2}C (HE-PC), respectively.

In HE-BI, each bit in a code vector is assigned to a specific encoded range which can come from any field in a rule. The code vector for a particular range \(R_i\) has value 1 in its assigned \(b_i\)-th bit, and ‘don’t care’ ‘∗’ in all other bits. For example, let us encode 7 ranges \(R_i\) coming from either the destination or source port field of the 8 rules \(L_i\), where \(i = 1, 2, ..., 8\). Suppose the code vector has 8 bits, and the \(i\)th bit (assuming \(b_i = i\)) is assigned to \(R_i\). Then the code vector for \(R_i\) has 1 in the \(i\)th bit and ‘∗’ in all other bits. For example, the code vector for \(R_1\) is 1********.

If a rule has more than one field with encoded ranges, then the code vector for the rule has 1s in the corresponding bits which are assigned to these encoded ranges and has ‘∗’ for all the other bits. For example, \(L_3\) has \(R_6\) in its source port and \(R_2\) in its destination port. Hence the code vector for \(L_3\) is *1***1**. If a rule has no encoded ranges, the corresponding code vector is simply wildcarded.

Now let us discuss how to encode the index vector. As stated in the previous subsection, the ranges from different fields must be encoded using different range tables. The index vector encoding method is the same for each range table. In what follows, we first show, by example, how the index vector is encoded, given that a set of ranges from a single rule field is to be encoded. Let us encode 5 ranges \(R_i\) \((i = 1, 2, ..., 5)\) which are taken from the destination field. Among these ranges, \(R_1\) overlaps with \(R_5\); \(R_2\) overlaps with \(R_4\) and \(R_5\); \(R_3\) is a sub-range of \(R_4\); \(R_4\) overlaps with \(R_5\). The encoded index vectors are given in Table I. Note that \(R_{r_1,r_2,...,r_n}\) represents the common sub-range among \(R_{r_1}, R_{r_2}, ..., R_{r_n}\).

Similar to the code vector, the \(b_i\)-th bit in the index vector is assigned to range \(R_i\). The encoding rules used to generate the index vectors can then be stated as follows:

1) For \(R_i\), the \(b_i\)-th bit in the index vector must be set to 1.
2) If $R_i$ is a sub-range of $R_j$, its index vector must have its $b_j$-th bit set to 1.

3) $R_{r_1,r_2,...,r_n}$ for $n$ overlapping ranges $R_{r_1}$, $R_{r_2}$, ..., $R_{r_n}$ needs to be expressed as a separate range if it is a new range other than any existing encoded ranges. The corresponding index vector must have its $b_{r_1}$-th, $b_{r_2}$-th, ..., $b_{r_n}$-th bits set to 1.

4) All other bits in the index vector must be set to 0s.

5) The weight or match priority for a range is equal to the number of 1s in the corresponding index vector.

When the field value in a search key matches a common sub-range of $n$ encoded ranges, it means that this field belongs to all $n$ ranges. The greater the number of 1s is in the index vector, the more ranges are matched and the higher the match priority this sub-range must have. Hence, the weight or match priority for a (sub-)range can be simply set as the number of 1s in the corresponding index vector. Note that there are two types of TCAM coprocessors in the market today. One is known as Order based TCAM (OTCAM) and the other WEIght based TCAM (WEITCAM) [4]. For an OTCAM, rules/ranges are arranged in an ordered
list. In the case when multiple rules/ranges are matched, the one in the lowest memory location is selected. Hence, in an OTCAM, the range with a larger weight value must be placed in the higher match priority (lower memory) location than a range with a smaller weight value whereas ranges with the same weight can be arbitrarily interleaved. On the other hand, for a WEITCAM, there is a weight field associated with each rule/range entry. When multiple matches occur, the rule/range with the highest weight value is selected, independent of the relative locations of the matched rules/ranges. Hence, in a WEITCAM, with weights assigned as in Table I, the ranges can be placed at any available memory location in the block allocated for the range table.

The index vectors in the range table I are encoded as follows. The first step is to assign a bit \( b_i \) (here \( b_i = i \)) to each range \( R_i \) \( (i = 1, 2, ..., 5) \), and set the corresponding bit to 1. The second step is to set the bit in the index vector for each range at the bit location corresponding to a super-range of that range if any. For example, since \( R_3 \) has \( R_4 \) as its super-range, the 4th bit in the index vector for \( R_3 \) is set to 1. Then all the common sub-ranges are added to the range table and the corresponding index vectors are set, following rule (3). For example, \( R_{15} \) is a new sub-range of \( R_1 \) and \( R_5 \) and hence it is added to the range table and the 1st and 5th bits in the corresponding index vector are set. On the other hand, the common range of \( R_3 \) and \( R_4 \) is the same as \( R_4 \), and hence it does not need to be encoded again.

Non-compact range \( R_0 \) with all wildcarded bits is added to every range table and it has the lowest match priority. The index vector for this range is NULL. If the field of a search key does not match any range in a range table, \( R_0 \) will be matched. All 0s in the index vector means no encoded range is matched for this field value. After all the 5 ranges in the destination port field are encoded, the last three bits are not assigned to any ranges yet. These bits can be used to encode ranges from other fields. Let us encode \( R_6 \) and \( R_7 \) from the source port field. The resulting range table is shown in Table II.

<table>
<thead>
<tr>
<th>Range</th>
<th>Weight</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_0 )</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>( R_6 )</td>
<td>1</td>
<td>00000100</td>
</tr>
<tr>
<td>( R_7 )</td>
<td>1</td>
<td>00000010</td>
</tr>
</tbody>
</table>
Based on the above range tables, we are now in a position to give a concrete example for the search key encoding process. Assume that the NPU generates a 5-tuple search key $sk = \{1.2.3.4, 5.6.7.8, 1025, 1028, 17\}$ from the IP header of a received packet. Initially, the index vector $iv$ is set to NULL. First, the NPU passes the source port number 1025 to the TCAM coprocessor to match against Table II. Since the port number 1025 falls into $R_6$, an intermediate index vector $iv_1 = \{00000100\}$ is returned. Second, the NPU updates the index vector by performing an OR operation, i.e., $iv = iv \mid iv_1 = \{00000100\}$. Third, the NPU passes the destination port number 1028 to the TCAM coprocessor to match against Table I. The port value 1028 matches $R_2, R_4, R_5, R_{24}, R_{25}, R_{45}$ and $R_{245}$. As a result, the index vector $iv_2 = \{01011000\}$ for $R_{245}$ is returned because it has the highest weight value. Fourth, the NPU updates the index vector again, i.e., $iv = iv \mid iv_2 = \{01011100\}$. The bit value 1 at the 2nd, 4th, 5th and 6th bits in $iv$ indicates that the destination port number of the search key falls in the ranges $R_2, R_4$ and $R_5$ and the source port number falls into $R_6$. Finally, an encoded search key is formed by simply appending $iv$ to $sk$, i.e., encoded search key = $\{sk, iv\}$, which is used to match against the encoded rule table in the TCAM.

E. Proof of Correctness of DRES Range Encoding

In DRES, some ranges may be encoded, while others not. In what follows, we prove the correctness of DRES range encoding.

*Theorem 1*: DRES range encoding ensures correct rule matching.

*Proof*: There are two cases for a rule entry: (1) it has no encoded range; (2) it has one or more encoded ranges. Note that $sk$, the search key portion in the encoded search key $\{sk, iv\}$, is the original search key. In case (1), the encoded rule entry is the same as the original one, i.e., the field bits are not wildcarded, and the code vector is wildcarded. Hence, no matter what the index vector is, the rule matching is correct. In case (2), the field(s) of the rule with encoded range(s) is (are) wildcarded. This implies that no matter what values the corresponding fields in a search key take, this (these) encoded range field(s) in the encoded rule will be matched. In this case, whether the range(s) is (are) matched is completely determined by whether the index vector matches the code vector or not. Note that the code vector has 1(s) in the assigned bit(s) corresponding to the encoded range(s) in the rule and has *s for all the other bits. Also note that the index vector has 1s in all the positions corresponding to any encoded ranges the fields in the search key fall into.
and has 0s in all the other positions. Then it is not difficult to realize that a match between the index vector and the code vector occurs if and only if the encoded range field(s) in a rule matches (match) the corresponding fields in the search key.

IV. ENCODED RANGE UPDATE PROCESS

In a PF table, new rules may be added and old rules may be deleted from time to time. Hence, some popular ranges may eventually become unpopular, and some unpopular ranges may become popular. The dynamic range selection algorithm selects ranges with the largest encoding gains for encoding. If any of the newly selected ranges are different from the existing encoded ranges, the encoded ranges which have not been selected must be un-encoded to release the assigned bits to encode the newly selected ranges. The process for updating encoded ranges and the corresponding rules is called an encoded range update process.

In this section, we address this issue by proposing a lock-free encoded range update algorithm, which allows the encoded range update and the search key encoding/PF table lookup processes to occur simultaneously, without impacting lookup performance. While the encoded range update process is carried out through a CPU-TCAM coprocessor interface, the search key encoding/PF table lookup processes are performed through an NPU-TCAM coprocessor interface as shown in Fig.1. The basic idea is to maintain consistent and error-free rule and range tables throughout the update process, thus eliminating the need for locking the tables. The lock-free idea is first proposed in [21] to allow lock-free rule update for PF in a TCAM. We extended the idea in [21] to allow encoded range update, which involves both range table update and PF table update. The algorithm that applies to an OTCAM is described in this paper. The one that applies to a WEITCAM is straightforward and hence it is not discussed here.

Generally speaking, updating a TCAM database without TCAM locking may generate two possible types of incorrect TCAM lookups, i.e., erroneous and inconsistent lookups. An erroneous lookup may occur if a TCAM rule gets a match while the rule or its corresponding action is partially updated. Note here that rules and actions are used in generic sense, and may represent ranges and index vectors as well. Inconsistent lookup means that the rule that the search key matches is not the best rule match for that search key. An inconsistent lookup may occur when a match takes place in the middle of a database update process and there is no guarantee of table consistency until the process finishes.

In general, each TCAM slot has a valid bit field associated with it, which allows a rule entry to be activated
or deactivated/deleted by simply setting or resetting the valid bit for that rule entry. As explained in [21], the key to avoid erroneous lookups is to avoid directly overwriting rule fields and/or the corresponding action when that rule entry is active, i.e., it may be matched. Instead, any write operations for a rule/action over an existing rule/action must be decomposed into a write process including three operations: (1) inactivate the rule; (2) write the rule/action; (3) activate the rule again. Writing a rule/action into an empty rule entry only requires the last two operations. Likewise any operations to move a rule-action pair to a new TCAM-associated memory location must be decomposed into a move process including: (1) using a write process to write the pair to the new location; (2) inactivating the rule at the old location. For DRES, it is assumed that the write and move processes are used and hence no erroneous lookups may occur. In what follows, we simply use write and move to stand for a write process and a move process, respectively.

Inconsistent lookups will not occur if, for each rule move, a search key matching results in a matched rule that is the same as the one that would be matched before the rule move; for each rule addition or deletion, a match always results in a matched rule that is the same as that would be matched either right before or after the rule addition or deletion [21]. Any TCAM database update algorithm that ensures consistent and error-free rule matching does not require TCAM locking for database updating. In what follows, a lock-free update algorithm is described in detail. We first introduce an important theorem.

**Theorem 2:** If a range is exactly implemented in the rule table, no inconsistency associated with this range may occur whether the range is encoded in its range table or not.

**Proof:** If the range is not encoded in its range table, the Theorem is obviously correct. If the range is encoded in its range table, a bit in the index vector is assigned to encode this range. If the field of a search key falls into this range, the corresponding bit in the index vector is set. However, in the rule table, the corresponding bit in the code vector for this range is wildcarded since the rule having this range is exactly implemented. Hence, setting the bit for this range in the index vector of the search key has no effect on the rule matching. □

In DRES, all the empty range/rule entries in a range/rule table are kept either at the top or at the bottom of the table. At least one empty range/rule entry is assumed to be available to serve the purpose for consistent encoded range update. For each table in an OTCAM, we assume the entries at the top in a table have higher
match priorities than the ones at the bottom. This means that the action for the matched range/rule at the top end is returned if multiple ranges/rules in a table are simultaneously matched by a field/search key.

In DRES, a free bit in the index/code vector is reserved and used for encoded range update. The encoded range update process can be decomposed into two phases: (1) use the free bit to encode a newly selected range; (2) un-encode an encoded range to release the free bit. These two phases are explained separately in the following two subsections.

A. Encoding a Newly Selected Range

For a newly selected range to be encoded, that range appeared in any rule in the TCAM is initially exactly implemented. A consistent rule table is maintained if the range table update is ahead of the rule table update, according to Theorem 2. Hence, in our algorithm, the range table is updated first, followed by the rule table update.

Note that only the range table associated with the field that the newly selected range belongs to needs to be updated. The free bit in the index vector is assigned to encode this range. Then the newly selected range and all possible common sub-ranges generated by this and the existing encoded ranges are added to the table. If this newly selected range is a super-range of an existing encoded range, the corresponding bit in the index vector for that existing range must be set to 1.

Similar to PF table update [21], there are two steps for the range table update. The first step is to consistently move the ranges and their index vectors from the top (bottom) to the bottom (top), while leaving the entries for the newly selected range and corresponding sub-ranges empty. To ensure table consistency upon each move, the ranges are moved according to their original order. This order of moves maintains the original priority relationships and thus the table consistency. If a range is a sub-range of the newly selected range, the corresponding bit in the index vector is changed to 1 after it is moved to a new location. This change has no effect on the search key matching since the corresponding bit in the code vector in an encoded rule is wildcarded.

The second step is to write the newly selected range and the associated sub-ranges and their index vectors to the pre-allocated locations in decreasing priority order (i.e., the ranges with higher match priorities are added before the ranges with lower priorities are added).

After finishing the range table update, the following simpler process is used to update the rule table. The
rules are moved from the top (bottom) to the bottom (top) with their relative orders unchanged. If a rule involves the newly encoded range, then at the new location the rule is moved to, the range is encoded by adding the corresponding bit in its code vector and wildcarding the field of the rule in which the range appears. Hence, multiple exactly implemented rule entries belonging to the same rule are reduced to one rule entry after the rule is moved to the new location.

![Diagram](attachment:image.png)

**Fig. 8.** Updating process in a rule table. (a) $L_1$ is at slots 1 and 2, $L_2$ is exactly implemented by two rule entries, one at slots 3 and 4, the other at slots 5 and 6; (b) write $L_2$ to a new location with encoded destination field; (c) delete the two exactly implemented rule entries at slots 3 - 6; (d) move $L_1$.

Figure 8 gives a simple example demonstrating how to update a rule with a newly encoded range. Assume $L_1$ and $L_2$ are implemented in a TCAM. $L_1$ has higher match priority than $L_2$. The update process must keep the relative match priority for $L_1$ and $L_2$ unchanged at all times. $L_2$ has $R_1$ in its destination port field. The exact implementation of $L_2$ with respect to $R_1$ requires two rule entries, corresponding to two sub-ranges $\{256-511\}$ and $\{512\}$, represented by two-byte values 1x and 20, respectively, as shown in Figure 8 (a). Assume the first bit of the code vector is assigned to encode $R_1$ and the range table is already updated. Then the rule update for $L_2$ involves two steps: (1) write $L_2$ to a new location with the newly selected range...
encoded; (2) delete the rule entries at its old locations. Figure 8 (b) gives the configuration after step (1) completes, i.e., \( L_2 \) is written and activated at slots 7 and 8. Finally, \( L_1 \) is moved to slots 5 and 6. During the update process, \( L_1 \) always has a valid copy above \( L_2 \) that maintains the original priority relationship and provides a consistent table.

### B. Releasing an Encoded Range

When the newly selected range is encoded, some rule entries are released. If no free bit is left in the index and code vector, the encoded range with the least encoding gain is un-encoded to release a free bit. To un-encode a range, the corresponding field in a rule with this encoded range needs to be exactly implemented, which increases the number of rule entries in the table. However, the increased number of rule entries must be less than the reduced number of rule entries by encoding a newly selected range. Otherwise, the encoded range update will not happen in the first place. Hence, only one empty rule entry in the rule table is required to do an encoded range update.

To release an encoded range, the rule table is updated first, followed by the range table update. For the rule table update, the update process changes the encoded range into an exactly implemented range in all the rule entries having this encoded range. It does this through consistent rule moves and by changing the corresponding bit in the code vector to *s. For example, to release the encoded range \( R_1 \), assume \( L_2 \)'s encoded range is initially given in Figure 8 (d). \( L_1 \) is first moved to slots 1 and 2 as shown in 8 (c). Then two exactly implemented rule entries of \( L_2 \) are written and activated in slots 3 - 6 as presented in Figure 8 (b). Finally the encoded \( L_2 \) at slots 7 and 8 is deleted or inactivated, as shown in Figure 8 (a). For the range table update, both the encoded range and the derived sub-ranges need to be deleted. To ensure range table consistency, these ranges must be deleted in increasing match priority order. Also if another encoded range is the sub-range of the range to be un-encoded, the corresponding bit in the index vector must be reset. The consistent move process for the range table update is similar to that for adding an encoded range.

### C. Encoded Range Update Delay

The proposed encoded range update algorithm does not require locking TCAM tables during the update process. However, to ensure consistent and error-free lookups, the algorithm requires a relatively large number of write and delete operations, resulting in a longer update delay. This raises the concern as to
whether the update delay would be too large such that the update process cannot keep up with the update requests. We resolve this issue by giving a rough estimation.

We only consider the rule table update delay for doing the encoded range update. The update delay of the range table is neglected because the size of a range table is, in general, much smaller than the size of a rule table. Assume there are $N_{er}$ rule entries in the rule table. All the rule entries in the table are moved once for adding a newly encoded range, and once for releasing an encoded range. Hence, the number of rule entry writes and deletes is $2N_{er}$ for each encoded range update. Assume one rule entry write and delete cost 100 $ns$. Then for a rule table with 100,000 rule entries, the encoded range update delay is 0.02 seconds, and for a table with one million rule entries, the update delay is 0.2 seconds. This update delay is negligible, given that the range popularity distribution changes much slower than the rule update rate, which occurs on the order of once every few seconds to once every few days.

V. PERFORMANCE EVALUATION BASED ON REAL-WORLD DATABASES

Note that DRES cannot be compared with the existing range encoding schemes, because DRES is a bottom-up approach whereas the existing approaches are top-down. Hence, in this section, we focus on testing the performance of DRES using HE-BI in terms of the overall TCAM storage efficiency. However, to demonstrate why the hybrid encoding approach has to be used in DRES, we also test the performance of the complete encoding approach with Bit-map Intersection (CE-BI) in terms of TCAM storage efficiency for the PF table alone. Note that since the range tables for a complete encoding approach can be too large to be placed in a TCAM, a complete encoding approach, in general, cannot be used for the design of a bottom-up solution. The performance data for CE-BI is used only as a reference to quantify the performance difference between HE-BI used in DRES and CE-BI in terms of the TCAM storage efficiency for the PF table alone, not the overall TCAM storage efficiency, which is not comparable.

Four real-world 5-tuple PF databases (see Acknowledgement for the source of these databases) are used for testing. The range statistics of the four real-world databases are given in Table III. The number of rules varies from 183 to 1550. The storage expansion ratio, defined as the number of rule entries divided by the number of rules in a TCAM, ranges from 1.41 to 6.20. The percentage of rules with ranges varies greatly, from 7.7% to 54.9%. The number of unique ports, including both exact port numbers and port ranges, is from 9 to 34 for the source port field and from 40 to 54 for the destination port field. The number of unique
ranges found in any of these rule tables is small. For example, for both databases 3 and 4, the maximum number of unique ranges is 4 for the destination port and the maximum number of ranges in both port fields is 7. The total number of unique ranges found in all 4 databases is 10.

TABLE III

<table>
<thead>
<tr>
<th>Database</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules</td>
<td>279</td>
<td>183</td>
<td>264</td>
<td>1550</td>
</tr>
<tr>
<td>TCAM entries</td>
<td>949</td>
<td>553</td>
<td>1638</td>
<td>2180</td>
</tr>
<tr>
<td>Storage Expansion Ratio</td>
<td>3.40</td>
<td>3.02</td>
<td>6.20</td>
<td>1.41</td>
</tr>
<tr>
<td>Rule with range (%)</td>
<td>9.3</td>
<td>7.7</td>
<td>54.9</td>
<td>9.8</td>
</tr>
<tr>
<td>Rule with range in source port (%)</td>
<td>7.8</td>
<td>6.6</td>
<td>43.2</td>
<td>1.4</td>
</tr>
<tr>
<td>Rule with range in destination port (%)</td>
<td>7.8</td>
<td>6.6</td>
<td>20.1</td>
<td>8.5</td>
</tr>
<tr>
<td>Unique source ports</td>
<td>13</td>
<td>9</td>
<td>28</td>
<td>34</td>
</tr>
<tr>
<td>Unique destination ports</td>
<td>43</td>
<td>40</td>
<td>49</td>
<td>54</td>
</tr>
<tr>
<td>Unique ranges in source port</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Unique ranges in destination port</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table IV shows the range frequency and the number of sub-ranges to exactly implement the range in TCAM for all four databases. FSP, FDP, TF and NSUB represent the range frequency in the source port, in the destination port, in both ports and the number of sub-ranges to exactly implement the range, respectively.

From table IV, one notes that $RN_1$ is the most popular range, making up about 88.3% of all the ranges. Both databases 1 and 2 have $RN_1$ and $RN_4$ in both source and destination ports. Database 3 has $RN_1$, $RN_5$ and $RN_6$ in its source port and $RN_1$, $RN_7$, $RN_8$, and $RN_9$ in its destination port. Database 4 has $RN_1$, $RN_2$ and $RN_3$ in both source port and destination port, and $RN_{10}$ in the destination port.

Now we apply DRES to all the 4 databases for a 64-bit slot TCAM, widely used in today’s TCAM coprocessors. As stated in Section III, the five-tuple rule has 104 bits; each rule entry takes 2 slots in the TCAM; and each rule entry has 24 free bits, which is much larger than 7, the maximum number of unique ranges found in the 4 databases. Hence, no extra slot is needed for range encoding.

In DRES, to encode ranges in both source and destination port fields, two range tables are needed. We
assume that all the range tables are stored in the TCAM, and will be counted as part of the storage cost for DRES. In each range table, a range is exactly implemented with each sub-range taking one slot (i.e., half of a rule entry). If some ranges in the range table overlap with one another and generate a new sub-range, that new range must be included in the range table. For the four databases, there is only one such range, which appears in the destination range table for database 4, i.e., sub-range \(1024 - 2511\) which is the common range of \(RN_1\) and \(RN_{10}\) and takes 5 slots. For the four databases, the maximum size (in slots) of the range tables is 29. In practice, due to the possible encoded range updates, a range table must be configured to be much larger than 29. Let us assume that 60 slots are allocated for each range table, doubling the maximum size found in the four databases.

Note that in DRES, as soon as either a range table is fully utilized or all the free-bits are exhausted, DRES can stop encoding more ranges to avoid overflowing the range table. In contrast, for CE-BI, all the unique port values for an encoded field must be encoded, each consuming a separate bit in the code vector. Hence the size of the code vector must be larger than the sum of the number of unique port values for each encoded field. Unlike DRES, which has full control over the code vector size, the CE-BI fails if the number

<table>
<thead>
<tr>
<th>Name</th>
<th>Range</th>
<th>FSP</th>
<th>FDP</th>
<th>TF</th>
<th>NSUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN_1</td>
<td>&gt; 1023</td>
<td>149</td>
<td>193</td>
<td>342</td>
<td>6</td>
</tr>
<tr>
<td>RN_2</td>
<td>109-110</td>
<td>10</td>
<td>11</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>RN_3</td>
<td>1645-1646</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>RN_4</td>
<td>33434-33600</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>RN_5</td>
<td>514-1023</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>RN_6</td>
<td>&gt;1024</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>RN_7</td>
<td>33435-33524</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>RN_8</td>
<td>6660-6669</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>RN_9</td>
<td>6000-6099</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>RN_{10}</td>
<td>20-2511</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>
of port values is more than the code vector can accommodate. Therefore, in practice, a code vector with a sufficiently large number of bits must be configured to ensure that the scheme does not fail. CE-BI allows a total number of 40 (16 bits in port field plus 24 free bits) bits to be used by the code vector to encode one port field. If the number of unique port values of that port field exceeds 40, an extra slot must be configured for each rule entry.

From Table III, we see that although the number of unique port values in the source port field is less than 40 in all the databases, the numbers of unique port values in databases 3 and 4 reach 28 and 34, respectively, suggesting that an extra slot should be configured to help ensure that there are no overflows of the code vector space. In the performance evaluation of CE-BI, however, we assume that no extra slot is allocated for encoding the source port field. But an extra slot is assumed to be allocated for encoding the destination port field, and both source and destination port fields, simply because the number of unique port values for the destination port field reaches 40 or more for all the databases. Note that for CE-BI, the TCAM storage overhead due to the range tables is not counted, which is in general very large because all the unique source and destination ports must be included in the range tables.

### TABLE V

**TCAM Storage Expansion Ratio with and without Encoding.**

<table>
<thead>
<tr>
<th>Database set</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without encoding</td>
<td>3.40</td>
<td>3.02</td>
<td>6.20</td>
<td>1.41</td>
</tr>
<tr>
<td>Encoding Source port</td>
<td>DRES 1.50</td>
<td>1.49</td>
<td>2.07</td>
<td>1.4</td>
</tr>
<tr>
<td>Encoding Source port</td>
<td>CE-BI 1.39</td>
<td>1.33</td>
<td>1.95</td>
<td>1.38</td>
</tr>
<tr>
<td>Encoding Destination port</td>
<td>DRES 1.50</td>
<td>1.49</td>
<td>3.00</td>
<td>1.05</td>
</tr>
<tr>
<td>Encoding Destination port</td>
<td>CE-BI 2.09</td>
<td>1.99</td>
<td>4.33</td>
<td>1.54</td>
</tr>
<tr>
<td>Encoding Both ports</td>
<td>DRES 1.22</td>
<td>1.33</td>
<td>1.23</td>
<td>1.04</td>
</tr>
<tr>
<td>Encoding Both ports</td>
<td>CE-BI 1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table V shows the TCAM storage expansion ratio after range encoding. For the ease of comparison, the TCAM storage expansion ratio without range encoding, as listed in Table III, is also listed in Table V. DRES achieves better encoding gain than CE-BI when ranges in either both port fields or the destination port field only are encoded. CE-BI achieves better encoding gain when the source port field only is encoded.
This, however, is based on the assumption that in CE-BI, no extra slots are needed for the source port field encoding. The relatively reduced encoding gain in DRES is due to the added overhead of a range table in the TCAM. Note that due to the lack of control of the code vector size, CE-BI may lead to negative encoding gain, as is the case for the encoding of the destination port field and both port fields for database 4. We also note that the TCAM storage expansion ratio for CE-BI is lower bounded at 1.5 when one extra slot has to be added to each rule entry. This is simply because each rule entry takes 2 slots and adding one extra slot expands the TCAM by 50% even if all the ranges are encoded, as is the case in Table V when both ports are encoded. In contrast, DRES can reduce the TCAM storage expansion ratio to close to 1, especially when the rule table size is relatively large compared with the range table sizes, as is the case for database 4.

In summary, DRES can significantly improve the overall TCAM storage efficiency for range matching. The key to achieve this performance gain is to use a hybrid encoding scheme, rather than a complete encoding scheme.

VI. ANALYTICAL PERFORMANCE EVALUATION

The performance analysis in the previous section is based on existing real-world databases, which may not capture the worst-case scenarios that can occur in the future as the Internet becomes more and more policy based. In this section, we use a probabilistic model to analyze the performance of DRES in a wide range of parameters. This probabilistic model allows us to perform a rather general analysis of DRES without having to resort to simulation, which is often tedious and can only sample a limited number of parameter settings.

In our model, each rule has $K$ fields and each field can be a range or an exact number. To avoid modeling unnecessary details, we use compact ranges instead of both exact numbers and compact ranges. For example, the source port numbers 80 and 23 and the source port range $<1024$ are instances of the compact range for the source port field.

The following parameters are defined in the model:

- $N_r$: total number of rules in the PF table.
- $N_k$: total number of unique non-compact ranges in field $k$.
- $p_k$: probability for non-compact ranges to occur in field $k$.
- \( p_{k,j_k} \): probability for the \( j_k \)-th unique range to occur among all the possible ranges (including both the compact and non-compact range) in field \( k \).
- \( p_{k,j_k}^r \): probability for the \( j_k \)-th unique non-compact range to occur among all the possible non-compact ranges in field \( k \).
- \( m_{k,j_k} \): number of sub-ranges to exactly implement the \( j_k \)-th range in field \( k \) in TCAM.
- \( W(p_{k,j_k}, m_{k,j_k}) \): total number of TCAM rule entries required to implement all the rules as a function of \( p_{k,j_k} \) and \( m_{k,j_k} \).
- \( \eta \): TCAM storage expansion ratio.

Without loss of generality, the compact range in field \( k \) is specified as \( j_k = 0 \). In other words, \( p_{k,0} \) is the probability for the compact range to appear in field \( k \) and \( m_{k,0} = 1 \) (\( k = 1, 2, ..., K \)). Then \( p_{k,j_k} \) for the \( k \)-th field can be expressed as

\[
p_{k,j_k} = \begin{cases} p_k p_{k,j_k}^r & j_k = 1, ..., N_k \\ 1 - p_k & j_k = 0 \end{cases}
\]  

(1)

For simplicity, we assume that the range distributions for different fields are independent of each other. Then we have:

\[
W(\{p_{k,j_k}, m_{k,j_k}\}) = N_r \sum_{j_1=0}^{N_1} ... \sum_{j_K=0}^{N_K} \prod_{k=1}^{K} m_{k,j_k} p_{k,j_k}
\]

(2)

Note that this expression applies to a PF table with or without range encoding. The only difference is that \( m_{k,j_k} \) value changes to 1 after the \( j_k \)-th range in the field \( k \) is encoded while \( p_{k,j_k} \) remains unchanged. The TCAM storage expansion ratio \( \eta \) can be expressed as:

\[
\eta = W(\{p_{k,j_k}, m_{k,j_k}\})/N_r.
\]

(3)

Note that \( \eta \) is a function of \( p_{k,j_k} \), which is further dependent on \( p_k \) and \( p_{k,j_k}^r \) as shown in Eq. (1). As shown in Table III, \( p_k \) can take a wide spectrum of values, ranging from 1.4% to as high as 43.2%. In our numerical studies, \( p_k \) is set to 20% and 30% in two different cases. On the other hand, \( p_{k,j_k}^r \) is highly concentrated on a few popular ranges. The statistics in Table IV indicates that \( \{ >1023 \} \) is the most popular...
one, appearing with a frequency of about 88.3%. The second popular one is \{109-110\}, which appears with a frequency of about 6%. This popularity distribution closely follows Zipf-like distribution with $z=3$ [3]:

$$p_{k,j_k}^r = c/rank^z(j_k), \quad j_k = 1, \ldots, N_k$$

(4)

where $z$ is the Zipf coefficient and $c$ is a normalization factor. In this distribution, $p_{k,j_k}^r$ is proportional to the popularity rank of range $j_k$. This distribution is used in our model to characterize $p_{k,j_k}^r$. Note that DRES becomes more efficient as $z$ gets larger because the popular ranges are concentrated on fewer ranges as $z$ increases.

If the number of unique non-compact range is smaller than the number of ranges that can be encoded, the performance analysis of DRES is trivial. Hence, we consider the situation when $N_k$ is large, e.g., $10^2$ to $10^4$. Moreover, we consider a wide range of $z$ values, e.g., from 0.5 to 3.0. To allow one to get a feel for what exactly these parameters mean in terms of numbers, Figure 9 plots the frequencies for the top 10 popular non-compact ranges at $z = 0.5$, 1.0, and 2.0, and $N_k = 100$ and 1000. First, one notes that for $z = 2.0$, the curves for $N_k = 100$ and 1000 are almost identical and cannot be distinguished from each other in the plot. This is due to the fact that as $z$ gets larger, the top most few popular ranges become more and more dominant. As a result, $N_k$ value has less effect on the relative frequencies of the popular ranges. Second, as $z$ goes as low as 0.5, the frequencies for even the most popular range constitute less than 7% of the total range appearance frequencies at both $N_k = 100$ and 1000 and the frequencies reduce very slowly as the range popularity rank drops. Therefore, we believe that the parameter range $z = 0.5$ to 3 should be wide enough to cover most of the worst-case scenarios that may occur.

We again consider a PF table with 104-bit 5-tuple rules and the TCAM slot size of 64 bits. Assume that only the two ports may have ranges, and the range distributions are the same for both ports. As one bit is reserved for encoded range updating, 23 out of 24 free bits are available for range encoding, with 12 bits for the source port ranges and 11 for the destination port ranges. $m_{k,j_k}$ is set to 6 for all the ranges, which is the average value found in the four real-world databases.

Apparently, as the number of unique port numbers becomes large, a complete encoding scheme becomes unviable due to the need to accommodate large range tables in TCAM and potentially oversized encoded rules. Hence, in this study, we do not compare a hybrid encoding scheme (used in DRES) with any complete encoding scheme. Instead, in this study, we compare two hybrid encoding schemes that may be adopted
by DRES, i.e., HE-BI and HE-PC. Note that in the previous section, the performance of HE-PC is not studied because DRES using HE-BI has already given the best possible performance in terms of TCAM storage efficiency (one rule per TCAM rule entry) and DRES using HE-PC cannot further improve the performance. Also note that while a comprehensive solution for DRES using HE-BI has been described in this paper, whether a comprehensive solution for DRES using HE-PC can be developed is yet to be studied. The key challenges to develop such a solution include the design of a heuristic dynamic range selection algorithm and a lock-free dynamic encoded range update algorithm. Since the study in this section is only concerned with the TCAM storage efficiency, we simply assume that a comprehensive solution for DRES using HE-PC already exists.

With 23 bits used for range encoding, HE-BI can encode 23 ranges. The number of encoded ranges using HE-PC is dependent on the range structure. In the best case, when no range overlaps with any other range, a total number of $2^{12} - 1 = 4095$ ranges in the source port and $2^{11} - 1 = 2047$ ranges in the destination port can be encoded. Such a large number is usually large enough to encode all the ranges that may appear in a PF table. In the worst case, when any range can overlap with any other range, a total number of 23 ranges can be encoded, the same as in HE-BI. In the average case, assume that all the selected ranges can be put into three layers such that any range from a given layer does not overlap with any other range from the same layer, and each layer has the same number of ranges (for more information on the concept of layer, please refer to [16]). Then a total number of $(2^4 - 1) \times 3 = 45$ ranges in the source port and $(2^4 - 1) \times 2 + 2^3 - 1$
= 37 ranges in the destination port can be encoded. In summary, in our numerical analysis, a total number of 23 (12 from the source and 11 from the destination port fields) ranges are encoded for HE-BI and a total number of 82 ranges (45 from the source and 37 from the destination port fields) are encoded for HE-PC.

In our numerical analysis, we neglect the TCAM overhead for accommodating the two range tables. This approximation becomes more and more accurate as the rule table size gets larger. For example, assume that each encoded range generates one extra sub-range that needs to be encoded as well. Then a total number of \((45+37) \times (1+1) \times 6 = 984\) slots or 492 rule entries are needed for all the range tables. This constitutes only 4.9% of the total TCAM memory used to support a rule table with 10 K rule entries. It further drops to 0.49% if a rule table with 100 K rule entries is supported. As the TCAM resource is likely to be constrained only when the rule table size becomes moderately large, this approximation should be a good one in practice.

A. Impact of the Zipf coefficient

We now study the impact of Zipf coefficient on the performance of DRES. \(p_k\) is set at 20% for this case study. Figure 10 shows the TCAM storage expansion ratios \(\eta\) without range encoding, encoding by HE-BI and encoding by HE-PC at \(N_k = 100\) and 1000, respectively. Without range encoding, \(\eta = 4\). Note that without range encoding, \(\eta\) is a function of \(p_k\) only and independent of other parameters in our model. For HE-BI, \(\eta\) is reduced to about 2.9 and 3.6 at \(z=0.5\), saving 28\% and 10\% of TCAM resources, at \(N_k = 100\) and 1000, respectively. About 50\% and 40\% of the TCAM resource are saved at \(z=1.0\) and \(N_k = 100\) and 1000, respectively. \(\eta\) further reduces to less than 1.5 at \(z=1.5\) and quickly converges to 1 as \(z\) further increases, independent of \(N_k\). This is because as \(z\) increases, the top 10 popular ranges become so dominant such that the rest of the ranges do not contribute much to the TCAM expansion. The storage expansion using HE-PC is reduced by up to 40\% from that of HE-BI. The results indicate that HE-PC is much more efficient than HE-BI in general.

B. Impact of the number of unique ranges

We set \(p_k = 30\%\). Figure 11 plots \(\eta\) without range encoding as well as with range encoding at \(z = 1.0\) and 1.5 as a function of \(N_k\).

From Figure 11, we can see that DRES significantly reduces \(\eta\) throughout the wide range of \(N_k\), e.g., from \(10^2\) to \(10^4\). Even at \(z=1.0\) and \(N_k = 10^4\), the TCAM storage space is saved by 30\% for HE-BI and
Fig. 10. Storage expansion ratio versus Zipf coefficient.

Fig. 11. Storage expansion ratio versus the number of unique ranges in each port field.

50% for HE-PC. The TCAM saving increases fast and becomes less sensitive to $N_k$ as $z$ further increases. $\eta$ becomes less than 1.7 for $z = 1.5$. HE-PC gives better performance than HE-BI by further saving about 30% TCAM storage space throughout the parameter range. The above numerical results indicate that DRES can significantly improve the TCAM storage efficiency in a wide spectrum of parameter ranges.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, a Dynamic Range Encoding Scheme (DRES) is proposed to improve TCAM storage efficiency in support of range matching. As a bottom-up approach, DRES is designed to solve the range-
matching issue for existing network processors using a TCAM coprocessor for policy/firewall filtering. DRES includes all the necessary ingredients for implementation in a network processor and its TCAM coprocessor with only a software upgrade.

A salient feature of DRES is its ability to have full control over the encoded rule size and to exploit the TCAM structure for maximizing the encoding gain. DRES provides a range selection algorithm to allow dynamic selection of ranges for encoding so that the maximum encoding gain is maintained whenever the rules are updated. Moreover, DRES uses a lock-free encoded range update algorithm that allows encoded ranges to be updated without impacting the rule-matching process. The performance on real-world databases shows that DRES can significantly reduce TCAM storage expansion from 6.20 to as low as 1.23. Statistical analysis on a probabilistic model demonstrates that DRES can significantly improve TCAM storage efficiency in a wide spectrum of parameter ranges.

To allow for overall simple design, DRES adopts the bit-map intersection encoding scheme, which is not the most effective range-encoding scheme. This seems to be sufficient for today’s PF databases as we showed in Section V. However, as demonstrated in Section VI, using the P^2C encoding scheme rather than the bit-map intersection encoding scheme in DRES can be much more effective in terms of TCAM storage efficiency. Hence, our future work will focus on developing a comprehensive solution for DRES using P^2C for range encoding. The key challenges in achieving this goal include the design of an efficient dynamic range-selection heuristic and a lock-free encoded-range update algorithm.

Acknowledgements:
The authors would like to express their gratitude to Professor Jonathan Turner and Mr. David Tylor from Washington University in St. Louis for kindly sharing their real-world databases and the related statistics with us. We also thank the anonymous reviewers and professor Matthew Wright for their constructive comments which greatly helped improve the paper quality.

REFERENCES