Scalable Multi-match Packet Classification Using TCAM and SRAM

Yu-Chieh Cheng, Pi-Chung Wang

Abstract—Packet classification is an enabling technology for various network services. Fast single-match packet classification can be achieved by using ternary content addressable memory (TCAM) because of the superior speed performance. TCAM has some drawbacks including incapability to store arbitrary ranges, confined TCAM capacity and limited choices of entry lengths. Moreover, TCAM only reports the first matching entry to impose a limitation on supporting multi-match packet classification, which requires all matching rules. The existing algorithms deal with the issues of TCAM-based multi-match packet classification by burdening TCAM with extra entries and/or accesses. In this work, we offload the overhead of TCAM to static random access memory (SRAM) to achieve efficient multi-match packet classification. Our scheme synthesizes TCAM compatible entries by using binary decision trees and employs SRAM for further comparisons. Each synthesized entry can be stored in one TCAM entry to significantly reduce TCAM consumption and fulfill low power consumption. The experimental results show that our scheme can lower the demand of TCAM to improve both search latency and energy efficiency. The scalability of TCAM-based multi-match packet classification can thus be improved drastically.

Index Terms—Packet classification, ternary CAMs, multi-match, range.

1 Introduction

Packet classification is one of the important functions in packet forwarding engines embedded by Internet routers to classify packets into network flows. It enables many services such as firewall packet filtering, quality of services, and intrusion detection. Packet classification is based on rules which define multiple fields of packet headers. These fields include source and destination IP addresses, source and destination ports, and protocol. The value of each field can be a prefix, a range, or an exact value. Different services may use different fields in a packet header. A field of a rule can be ignored by specifying a wildcard. A rule matches a packet if all fields of the rule match the corresponding fields of the incoming packet. Each rule is associated with an action to process matching packets. Some network services, such as firewall and quality of services, perform single-match packet classification, which only yields the best matching rule. The best matching rule could be the rule with the highest priority or the least cost. The services such as deep packet inspection, transparent monitoring and usage-based accounting require multi-match packet classification, which reports all matching rules [1]–[3].

Multi-match packet classification can be treated as a generalization of the single-match alternate because the highest-priority matching rule can always be extracted from all matching rules. Since one single instance of multi-match packet classification is usually faster than multiple instances of single-match packet classification, multi-match packet classification can also be used by multifunction devices that perform single-match packet classification for each function [4].

Currently, ternary content addressable memories (TCAMs), an extension of CAM, have been widely used for packet classification. They are embedded in line cards to act as forwarding engines (or coprocessors) to accelerate the process of packet forwarding. Each TCAM cell can store **0**, **1**, and **"don't care"**. In other words, TCAM can store binary strings with arbitrary bit masks (i.e. ternary strings). Each entry of a commodity TCAM chip can be configured to have a width of 72, 144, 288, or 576 bits. TCAM performs parallel searching upon all entries and only needs one access to accomplish a search. TCAM has several drawbacks including limited capacity, high cost and high power consumption. The extra hardware for implementing **"don't care"** state includes six transistors for the mask bit and four transistors for the match logic. As a result, each TCAM cell needs 16 transistors, which is 2.7 times larger than a standard SRAM cell [5]. In particular, TCAM costs about 30 times more per bit of storage than SRAM and consumes 150 times more power per bit than SRAM [3]. The extra logic and capacitive loading of TCAM also result in tripling the access time of SRAM [3]. Because all of these issues are directly associated with the number of TCAM entries used, the storage efficiency of TCAM becomes critical.

Similar to CAM, TCAM reports only the first matching entry indicating that it is inherently suitable for single-match packet classification. To support multi-match packet classification without using proprietary hardware, either extra TCAM entries or accesses, or both, is inevitable in the existing algorithms [1], [6]–[8]. Another obstacle of TCAM for performing packet classification is that ternary strings cannot represent arbitrary ranges efficiently. In a trivial range-to-prefix

The authors are with the Department of Computer Science and Engineering, National Chung Hsing University, Taichung, Taiwan 402, ROC. E-mail: pcwang@nchu.edu.tw

conversion, at most 900 TCAM entries are generated for a rule with two 16-bit port ranges [3]. Although novel range encoding algorithms have been proposed to alleviate (or avoid) the cost of range representation, speed or storage penalties are inevitable. In summary, the original design of TCAM is not directly adaptable to performing multi-match packet classification. Further work is necessary to address the issue.

In this paper, we present a scalable algorithm for TCAM-based multi-match packet classification. Since TCAM only reports the first matching rule and suffers from the cost of range representation, an efficient TCAM algorithm which can achieve multi-match classification and eliminate the need of range representation is desirable. The existing algorithms deal with these two issues independently. For example, SSA [7] and MUD [6] only address the storage problem of multi-match packet classification. Although the multi-match algorithms can combine with an efficient range representation algorithm, e.g. MUD and DIRPE [6], to minimize the overall TCAM entries, extra cost is still incurred. A traditional TCAM access includes **one TCAM search** and **one SRAM access**[1]. To offload TCAM's overheads, we use SRAM to store searchable data structures. Our algorithm employs binary decision trees to categorize rules geometrically. Then, an index rule for each rule subset is generated and stored in TCAM. The index rules are TCAM friendly, which means that each index rule occupies exactly one TCAM entry. The original rules covered by an index rule are stored in the associated SRAM entry for linear search. Our algorithm uses a bitwise discriminator to yield all matching index rules in TCAM. Since SRAM is more energy efficient and faster than TCAM, the extra accesses to SRAM do not incur high access latency and power consumption. Instead, by reducing the number of TCAM entries, energy consumption can be significantly reduced. The experimental results demonstrate that our scheme improves both power efficiency and search latency of multi-match packet classification. As a result, the scalability of TCAM-based multi-match packet classification is effectively improved.

In the rest of this paper, we provide an overview of previous schemes, including those aimed for multi-match packet classification and for other TCAM issues, in Section 2. Section 3 describes the main idea of our scheme. Section 4 presents our scheme in detail. The experimental results are reported and discussed in Section 5. Finally, we conclude this work in Section 6.

2 Related Works

In this section, we first review previous approaches on TCAM-based multi-match packet classification. Then, we introduce several related studies which address some issues of TCAM-based packet classification and review some research on software-based packet classification.

1. In a TCAM-based packet classifier, the content in the TCAM is compared, but not the case for the coupled SRAM.

2.1 TCAM-based Multi-match Packet Classification

Some commercial TCAMs support multiple matching by appending a valid bit to each TCAM entry. The valid bit indicates whether the corresponding entry is valid or not to be compared to the input search key. Initially, the valid bits of all entries are set so that the first matching rule (if any) will be reported in the first cycle. The valid bit of the previous matching entry is unset and another lookup with the same search key is issued again. The process continues until no matching rule is reported. All the valid bits must be reset for the next search key. This approach requires $m+1$ TCAM read and $2m$ write operations for each multi-match packet classification, where m is the number of matching rules.

Lakshminaryanan et al. propose MUD, which uses discriminators to retrieve multiple matching entries from TCAM [6]. MUD exploits unused bits in each TCAM entry to store an index, which satisfies that the TCAM entries with the same index value do not overlap with each other. The TCAM entries are sorted according to their indices in an ascending order. In the search procedure, a discriminator is appended to the search key for determining the set of TCAM entries for a comparison. Initially, the discriminator is set to wildcard to compare all TCAM entries. After identifying a matching entry with the index value i, the search procedure changes the discriminator to **"greater than i"** to exclude previous matching entries. The process repeats until no match is found. MUD modifies the discriminator of the search key instead of changing TCAM valid bits, and, consequently, achieves better search performance. However, MUD must convert a range which is larger than a certain value into multiple ternary strings. Since one TCAM access is required for each string, MUD may need more TCAM accesses than the number of matching rules. MUD also incurs high power consumption because each multi-match classification compares all TCAM entries multiple times [7].

Generating and storing all matching conditions of a rule set in TCAM is another approach. In [1], a set of geometric intersections of rules is determined for a rule set. Each intersection corresponds to a pseudo rule and each pseudo rule is associated with a list of rule identifiers, or match list, to indicate the matching rules. The pseudo rules are inserted into TCAM along with the original rules. A pseudo rule must be positioned in front of those rules in its match list to ensure the correctness. The geometric intersection (GI) scheme only needs one TCAM access to yield all matches; however, the intersections in a rule set could be too large to be stored in TCAM. Theoretically, there are O(N^k) pseudo rules, where N is the number of original rules and k is the number of fields. SSA [7] reduces the number of pseudo rules based on the observation that the number of intersections can be significantly reduced by splitting a rule database into several subsets. For each subset, the rules and their pseudo rules are stored in an independent

TCAM. Each multi-match packet classification accesses all TCAMs to yield all the matching rules. SSA does not consider the influence of extra TCAM entries caused by range-to-prefix transformation for the range fields. Its speed performance is tied to the number of subsets to fit into TCAM storage. In [4], a dual-phase scheme for rule set partitioning is presented. In the first phase, the rules overlapping with each other are gathered in a partition. The second phase further divides the rules in a partition into different blocks in a TCAM chip which can be accessed in parallel. The authors suggested that a TCAM chip needs at least eight blocks for parallel accesses. A recent work uses one TCAM for each field and generates all possible match conditions for the last two fields [8]. These match conditions are stored in SRAM and accessed by using the index identifiers of the last two fields. The performance of this scheme depends on the number of distinct match conditions in a rule set.

Multi-match packet classification can be achieved by modifying TCAM hardware. In [4], a bit vector is used to store the results of all TCAM matchlines, where each bit indicates whether the corresponding TCAM entry is matched. A modified priority encoder is proposed to retrieve the indices of all matching entries from the bit vector, where each cycle outputs one index. BV-TCAM [9] combines TCAM with Bit Vectors (BV) [10] to address the issues of multi-match packet classification. It stores range fields in SRAM to avoid rule expansion in TCAM. The search procedure of BV-TCAM consists of one search in TCAM and two one-dimensional range searches in SRAM. It uses bit vectors to store the results of both TCAM and SRAM searches, where the result from TCAM is directly exported from matchlines without passing through any priority encoder (i.e., priority encoder is removed). The result of multi-match packet classification is yielded by intersecting these bit vectors. FSBV optimizes the combination of TCAM and SRAM for Snort rule sets [11]. It splits port range fields into multiple one-bit fields to avoid the slow range-matching procedures, with the cost of accessing extra bit vectors. FSBV may result in prolonged bit vectors for the rules with arbitrary ranges. Both FSBV and BV-TCAM use FPGA to implement their proprietary architecture.

2.2 Other Research on TCAM

There are numerous algorithms proposed to solve the issues of TCAM-based packet classification, including range representation [12]–[19], rule-set compression [20]–[23], and energy consumption [12], [24]–[27]. The algorithms addressing the first two issues can reduce the requirement of TCAM storage. A smaller TCAM also has better energy efficiency and shorter access latency [25].

In [28], the authors demonstrated that the maximum cost of range-to-prefix conversion is W, where W is the width of the range field. The problem of range-to-prefix conversion can be alleviated by range encoding [13]–[19]. There are two types of range encoding algorithms, database-dependent and database-independent. Database-dependent encoding algorithms have superior efficiency for both TCAM entry length and count, but they require extra memory accesses to map header values of the encoded fields. Database-independent encoding algorithms avoid extra searches, but they may still incur rule expansion. In [29], the port ranges are stored in SRAM with wide words (e.g. 512 bits) to avoid the cost of range representation.

TCAM minimization algorithms are designed for single-matching packet classification, where only the first matching entry will be reported. Some low-priority rules are useless since they will not be reported by TCAM for any packets. Several TCAM minimization algorithms are developed to remove these redundant entries [20]–[23].

Since TCAM always reports the first-matching entry, the insertion of a new rule must ensure that the predefined rule priority is conformed. For single-match packet classification, the priority of each rule is usually defined by its order. Some algorithms of multi-match packet classification also define the priority of each TCAM entry. For example, a pseudo rule of GI [1] or SSA [7] must have a higher priority than its overlapping original rules. For MUD, a rule with a smaller index value has a higher priority [6]. In [30], the authors improve the update performance by moving only the overlapping rules. To avoid TCAM entry movements, another algorithm sorts rules according to their priority values [31]. The search procedure requires at most $\log|Pr|$ TCAM accesses, where $|Pr|$ denotes the number of distinct priority values. CoPTUA is an algorithm which can maintain the consistency of rules without locking the TCAM coprocessor during the update process [32]. In [24], update performance is improved by partitioning rules into two subsets.

2.3 Software-based Packet Classification

Multi-match packet classification can also be achieved by using slower software implementations. These implementations may use different types of data structures with optimized construction and search algorithms [3]. Decision tree has been considered as an effective data structure for packet classification [33]–[35]. While the algorithms using only a single decision tree may incur high storage requirement for heavily overlapping rules, the storage issue can be addressed by employing multiple decision trees. In [36], the authors present BSOL2, binary-search-on-levels with two rule subsets, where the first subset stores the rules with source prefix whose length is less than five, and the remaining rules form the second subset. Each subset is stored in one decision tree. In some cases, this solution can significantly reduce the memory requirement. However, it lacks of flexibility and may not be suitable for all rule databases. EffiCuts [35] generates one decision tree for rules with the same combination of wildcard fields. Because there are at most 26 combinations, 26 decision trees are generated in the worst case indicating deterioration of the search performance. The authors proposed selective tree merging to reduce the number of decision trees. Two trees with one distinct wildcard field could be merged and five to six trees are still required. In [37], an extension to BSOL is proposed to generate new decision trees based on the number of total rule replicas presented. This approach generates a new decision tree if the number of total rule replicas is higher than a predefined threshold value, namely *expansion factor*. The rules replicated first are stored in the new decision tree. The new decision tree may lead to the construction of another one according to the characteristics of the stored rules.

3 Motivation and Challenges

Some of the existing schemes require complex hardware configurations to achieve multi-match packet classification. Several schemes use proprietary TCAM architecture [4], [9] or extra hardware [9], [11], and some require multiple TCAMs [7], [8]. Some schemes also use SRAM to avoid the problem of range representation in TCAM [9], [11]. Although these schemes may have superior performance, the complex configuration may decrease resource utilization due to the different characteristics of a rule set. For example, the number of TCAM partitions and the size of each partition in [4] could vary for different rule sets. A proprietary architecture may also incur high implementation cost. To simplify the hardware configuration and lower the implementation cost, we focus on a succinct TCAM architecture with only one single nonproprietary TCAM chip.

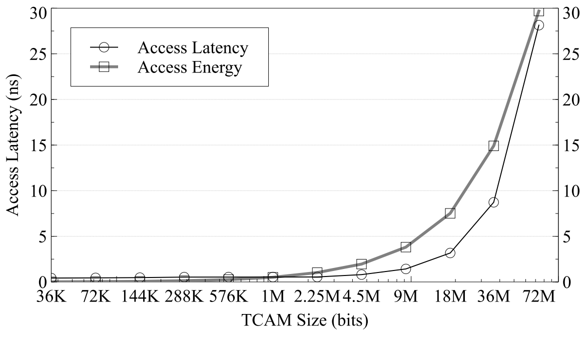
We observed that the state-of-the-art solutions for multi-match packet classification based on native TCAMs use extra bits [6] or extra TCAM entries [1], [7] to yield all matching rules. Several algorithms even yield redundant TCAM accesses [6], [7]. We summarize the cost of these algorithms in Table 1, where the number of unique discriminator values for MUD is optimized to m, the number of matching rules in the worst case. In addition to the cost of yielding all matching rules, range representation in TCAM requires extra bits [6] or extra TCAM entries [1], [6], [7].

Each TCAM chip is usually coupled with an off-chip SRAM, where each TCAM entry is mapped to an SRAM word. Once the TCAM chip is searched, a priority encoder determines and encodes the location of the highest-priority match. The encoded address is then used to retrieve the corresponding data in the SRAM.

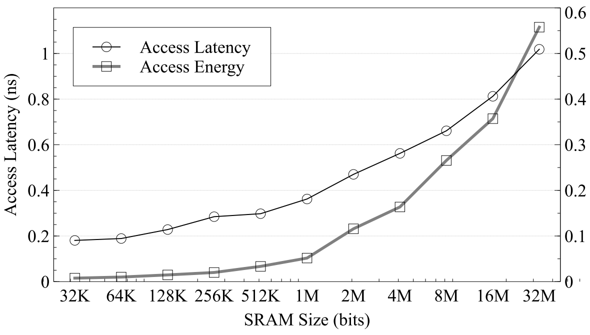
TABLE 1
Cost Comparisons for Multi-Match Packet Classification Algorithms Using Non-proprietary TCAM

	Entries	Extra Bits	TCAM Accesses
GI [1]	O(N^k)	0	1
SSA [7]	O($(N/S)^k$)	0	S
MUD [6]	O(N)	$\log_2 m\frac{(2^r-1)}{r}$	$1+\frac{\log_2 m^{m-1}}{r}$

N: the number of rules, k: the number of inspected fields
S: the number of rule subsets, m: the maximum number of matching rules
r: the number of subfields after splitting a range field



(a) TCAM parameters generated by TCAM-Model [38].



(b) SRAM parameters generated by CACTI6 [39].

Fig. 1. Access Latency and Energy of TCAM and SRAM.

We use two notable software, TCAM-Model [38] and CACTI6 [39], to generate the access latency and energy of TCAM and SRAM with different sizes in Fig. 1. The basic configurations involve 32nm process, 144-bit TCAM entry width, and 512-bit SRAM entry width. The number of TCAM entries varies from 256 to 524,288 and the number of SRAM entries varies from 64 to 65,536. As shown in Fig. 1(a), TCAM have scalability issues since both access latency and energy consumption increase exponentially when memory capacity increases. In contrast, Fig. 1(b) shows that although increments can also be found as SRAM increases in size, the increment is moderate as compared to TCAM. The difference lies in the design of the match line, search line, and priority encoder of TCAM [38].

In this study, we improve the performance of TCAM-based packet classification by incorporating SRAM in the search procedure. Because of the presence of SRAM in a TCAM-based architecture, incorporating both memory technologies is advantageous in tackling the issues of multi-match packet classification. Take the sample parameters in Fig. 1 as an example, where large TCAM chips, e.g. 4.5 Mbits or larger, consume ten times or more energy than the same-size SRAM chips. Given a packet forwarding engine with a fixed power consumption budget, the power consumption can be reduced if one TCAM access is replaced by less than ten SRAM accesses. Accordingly, we alleviate the overhead of multi-match packet classification by storing all overlapping rules in one or several SRAM words while TCAM stores the union of these overlapping rules. When the TCAM entry matches an incoming packet, the rules in the corresponding SRAM words are compared to yield the matching ones. However, the number of overlapping rules in different rule sets is usually different. In an extreme case, all rules overlap with each other and result in a pure linear search. In summary, our proposal faces two challenges. First, the union of all overlapping rules should be TCAM friendly so that the TCAM storage efficiency can be optimized. Second, the number of overlapping rules corresponding to a TCAM entry should be controllable to avoid a long search latency.

4 Multi-match Using Sram-assisted Tcam (MUST)

We address both challenges mentioned above geometrically. Assume that the rules in a database have k fields, and each field can be an arbitrary range. These rules can be treated as hyper-rectangles in a k-dimensional space, where the width of each dimension is equal to the corresponding range. The inspected header fields of each packet can be treated as a point in this space.

To yield all matching rules efficiently, the rules that match the same or similar packets should be gathered in the same SRAM entry. The rules matching the same packet must overlap with each other, i.e. the corresponding hyper-rectangles of the rules that match the same packet must overlap with each other in the k-dimensional space. We split the k-dimensional space into several disjoint subspaces to reduce the number of possibly matching rules. Since a packet matches at most one subspace, we only need to access the rules in a subspace. We create one index rule for each subspace and the rules in a subspace are stored in SRAM. To perform packet classification, the search key is compared with the index rules stored in TCAM to yield the first matching TCAM entry. Then, the original rules stored in the corresponding SRAM entry are retrieved by a linear search to determine the matching rules. By properly controlling the number of rules in a subspace, we can keep the cost of extra SRAM accesses reasonable. In the following, we describe the proposed algorithm in detail.

4.1 Space Decomposition Using a Single Decision Tree

We use the data structure of a single decision tree to serve the purpose of generating index rules. Each leaf node of the decision tree corresponds to an index rule. Because the index rules are stored and searched in TCAM, the construction of our decision tree only focuses on storage efficiency. The tree depth is not of interest in our construction procedure, unlike the algorithmic solutions based on decision trees [33]–[35]. We limit the number of branches of a node to two to construct a binary decision tree. We also consider that each field of an index rule should be represented as a prefix so that each index rule only occupies one TCAM entry.

In our proposed binary decision tree, the space of an internal node is equally divided by its two child nodes and the two subspaces differ only in one of the dimensions. The two contrasting sides correspond to the upper and lower half of a selected dimension of their parent node. Each node of the decision tree has a list of rules which overlap with the node's subspace.

When constructing a binary decision tree, the first step is to have a root node corresponding to the whole k-dimensional space, and all rules are stored in the list of the node. The rule list of a leaf node is allowed to have at most bin-threshold (*binth*) rules. If the length of a rule list is more than *binth*, then two child nodes will be appended to the node by halving the length of a selected dimension. Each rule in the parent node will be inserted into a child node if its hyper-rectangle overlaps with the space of the child node. The procedure of building a decision tree stops when no new child nodes are generated. In this procedure, a rule could be inserted into both new child nodes to cause rule replication. Assume that there are n rules whose i_{th} field is a wildcard, $1 \leq i \leq k$, associated with a decision-tree node. The value of n is larger than *binth*, and two child nodes are generated accordingly. If the i_{th} field is selected for dividing the space of the node, then these n rules will overlap with both child nodes to result in n replicas. Moreover, the number of rules in both child nodes remains the same as their parent node to result in more child nodes and a higher decision tree. In the worst case, these rules replicate exponentially to result in explosive storage. To alleviate the problem of rule replication, the field generating minimum replicas is selected.

We show the pseudo codes of generating index rules for a rule set in Fig. 2. Initially, the function, *construct*, assigns all rules of rule set, *R*, in the *root* node. It then calls another recursive function, *divide*, to generate leaf nodes as mentioned above. The function, *D(node, i)*, calculates the number of replicated rules for the node by using the i_{th} field for partitioning. The value of i should minimize the value of *D(node, i)* so that the *divide* function can separate the rules into two child nodes with the fewest rule replicas in each iteration.

We use a set of 11 two-field rules as an example to explain the above procedure. The binary decision tree for the rules in Table 2 is shown in Fig. 3, where the *binth* value is two. The associated rules are listed in each node. The leaf nodes are denoted either in bold or in dotted-bold. The procedure of constructing a decision tree is as follows. First, all rules are associated with the root node. Next, the number of the distinct specifications for each field is derived. Since there are *eight* distinct field specifications for f_1 and *seven* for f_2, f_1 is selected for rule categorization. Accordingly, the rules are divided into two buckets where one bucket corresponds to the subspace $\langle 0*, * \rangle$ and the other corresponds to the subspace $\langle 1*, * \rangle$. If the hyper-rectangle of a rule overlaps with the subspace of a bucket, then the rule is

```
RuleSet construct(RuleSet R) {
    root=new TreeNode;
    node_list=new TreeNodeList;
    index_rules=new RuleSet;
    root.rulelist=R;
    node_list=divide(root, node_list);
    for (each node leaf_node in node_list) {
        index_rule=new Rule;
        index_rule.space=leaf_node.space;
        index_rule.rulelist=leaf_node.rulelist;
        append index_rule to index_rules;
    }
    return index_rules;
}

TreeNodeList divide(TreeNode node, TreeNodeList node_list) {
    if (|node.left.rulelist| ≤ binth)
        append node to node_list;
    return node_list;
    node.left=new TreeNode;
    node.right=new TreeNode;
    select dimension $i$ with minimum D(node, i);
    node.left.space=lower half of node.space on dimension $i$;
    node.right.space=upper half of node.space on dimension $i$;
    for (each rule $r$ in node.rulelist) {
        if (r.space ∩ node.left.space ≠ ∅)
            append $r$ to node.left.rulelist;
        if (r.space ∩ node.right.space ≠ ∅)
            append $r$ to node.right.rulelist;
    }
    node_list=divide(node.left, node_list);
    node_list=divide(node.right, node_list);
    return node_list;
}
```

Fig. 2. Pseudo codes for generating index rules based on single decision tree.

TABLE 2
An example with eleven rules on two fields.

Rule	f_1	f_2	Rule	f_1	f_2	Rule	f_1	f_2
F_1	$*$	0010	F_5	$00*$	$11*$	F_9	$111*$	$00*$
F_2	$*$	0111	F_6	$000*$	$01*$	F_{10}	0111	$*$
F_3	$0*$	$1*$	F_7	$111*$	$11*$	F_{11}	1111	$*$
F_4	$0*$	$01*$	F_8	$101*$	$01*$			

associated with the bucket. In this example, F_1 and F_2 are duplicated and inserted into both buckets. The above procedure repeats for each bucket until the number of rules in each bucket is less than or equal to *two*. Notice that there are more than *two* rules in the leaf nodes with dotted bold lines. These leaf nodes correspond to the overlapping areas completely covered by more than two rules. Since the overlapping rules always match all the points in the area, a further space decomposition is no longer necessary.

An index rule is synthesized for each leaf node. Each index rule consists of two parts, rule specification stored in TCAM and the associated original rules stored in SRAM. The rule specification of a leaf node is generated by collecting its ranges of all dimensions. Because the space of any node is exactly half of its parent node, each dimension of a subspace can be represented as a prefix. As a result, the rule specification of an index rule can be recorded by a single TCAM entry. For each index rule, the specifications of the associated rules

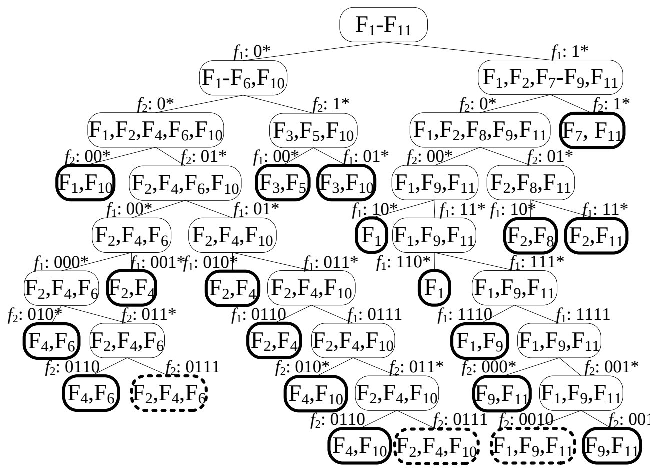


Fig. 3. Decision tree for the rules in Table 2. ($binth = 2$)

are stored in SRAM. In Fig. 3, there is an index rule, "$\langle 000*, 010*\rangle \rightarrow F_4\langle 0*, 01*\rangle, F_6\langle 000*, 01*\rangle$", where the rule specifications of F_4 and F_6 are stored in SRAM for further comparison. The specifications of the original rules could be ignored in some leaf nodes, e.g. the leaf nodes with dotted bold lines in Fig. 3, because the search keys matching these nodes always match the corresponding original rules. The index rules of these leaf nodes only record the indices of the associated rules. One index rule of a dotted-bold leaf node in Fig. 3 is "$\langle 000*, 0111\rangle \rightarrow F_2, F_4, F_6$", where only the indices of F_2, F_4 and F_6 are stored in SRAM. When an index rule in the TCAM is matched, the rules in the list of the corresponding leaf node will be fetched from SRAM to determine the matching rules. Because the index rules do not overlap with each other, each packet classification only accesses TCAM once. This architecture has two advantages. First, the storage efficiency of TCAM is promoted because index rules rather than the original rules with range fields are stored in TCAM, where each index rule occupies exactly one TCAM entry. Second, because the subspaces of all leaf nodes in a decision tree are disjoint to each other, each multi-match packet classification is performed by using one TCAM search and several SRAM searches upon a rule list.

The existence of the dotted-bold nodes usually leads to a space-inefficient decision tree such as the one with more leaf nodes than the original rules as illustrated in Fig. 3. A proper field selection for space partitioning can reduce the probability of rule replication only for some cases. Rule replication is especially difficult to avoid for the rules with wildcards. Increasing the *binth* value is an approach to eliminating the dotted-bold nodes. For example, we can use a larger *binth* value, four, to construct the decision tree for the rules in Table 2. As shown in Fig. 4, the number of leaf nodes is greatly reduced as compared to the previous decision tree in Fig. 3. However, it is difficult to determine a proper *binth* value for a rule set since not all rules heavily overlap with each other. The speed performance may

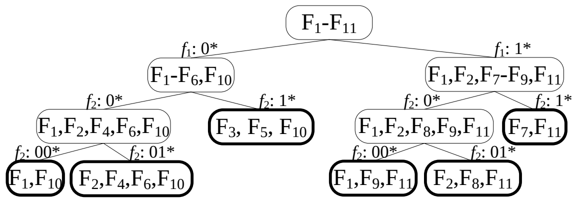


Fig. 4. Decision tree for the rules in Table 2. ($binth = 4$)

also degrade since the number of SRAM accesses is proportional to the *binth* value.

4.2 Space Decomposition Using Multiple Decision Trees

For the heavily overlapping rules, space decomposition with a single decision tree is inefficient. Although only one TCAM access is required for a multi-match packet classification, the storage performance of both SRAM and TCAM is unmanageable. To leverage both speed and storage performance, we allow a moderate increment of TCAM accesses to achieve better storage performance. Although extra TCAM accesses may degrade speed performance, the degradation would not be obvious since a memory chip with smaller capacity usually has shorter access latency, which will be discussed in Section 5. Since each decision tree requires one TCAM access, multiple decision trees are employed to organize heavily overlapping rules.

We extend the approach presented in [37] to serve our purpose of decomposing a search space. The approach is developed for the specialized decision tree of BSOL whose search procedure performs binary search to determine the matching leaf node. The idea of dynamically generating new decision tree is applied to the binary decision tree in our scheme. Unlike the previous approach that reduces both the number of rule replicas and the height of a BSOL decision tree, our new approach attempts to reduce the number of leaf nodes and keep the number of decision trees low.

Our approach determines whether a rule should be stored in a different decision tree according to the number of its replicas. We set an expansion factor to control the number of replicas of a rule to be stored in a decision tree. For each rule, there is a counter which is initially set to zero. When a node is split into two child nodes, each rule of the parent node is tested to identify the presence of overlapping child nodes. If a rule overlaps more than one child node, then the counter value is increased by one. When the number of replicas is larger than the expansion factor, this rule will be removed from the decision tree and stored in a new rule set. The procedure of tree construction proceeds for all of the rules. If the new rule set is non-empty after constructing a decision tree, then the above procedure repeats to generate another decision tree until all rules are stored in a decision tree. In the end, multiple decision trees might be constructed that depend on the characteristics of the

```
RuleSet new_construct(RuleSet R) {
    node_list=new TreeNodeList;
    index_rules=new RuleSet;
    $i = 0$;
    while ($R \neq$ NULL) {
        root[$i$]=new TreeNode;
        root[$i$].rulelist=R;
        R=NULL;
        reset all counters of all rules in root[i].rulelist;
        node_list=new_divide(root[i], node_list, R);
    }
    for (each node leaf_node in node_list) {
        index_rule=new Rule;
        index_rule.space=leaf_node.space;
        index_rule.rulelist=leaf_node.rulelist;
        append index_rule to index_rules;
    }
    return index_rules;
}

TreeNodeList new_divide(TreeNode node, TreeNodeList node_list, Rule-
Set R) {
    if (|node.left.rulelist| $\leq$ binth)
        append node to node_list;
    return node_list;
    node.left=new TreeNode;
    node.right=new TreeNode;
    select dimension $i$ with minimum D(node, i);
    node.left.space=lower half of node.space on dimension $i$;
    node.right.space=upper half of node.space on dimension $i$;
    for (each rule $r$ in node.rulelist) {
        if (r.space $\cap$ node.left.space $\neq \emptyset$) {
            append $r$ to node.left.rulelist;
            r.counter++;
        }
        if (r.space $\cap$ node.right.space $\neq \emptyset$)
            append $r$ to node.right.rulelist;
            r.counter++;
        }
        r.counter$--$;
        if (r.counter $>$ expansion_factor) {
            remove $r$ from the current decision tree;
            append $r$ to R;
        }
    }
    node_list=new_divide(node.left, node_list, R);
    node_list=new_divide(node.right, node_list, R);
    return node_list;
}
```

Fig. 5. Pseudo codes for generating index rules based on multiple decision trees.

original rule set. The leaf nodes of all decision trees are then extracted to generate the index rules.

The pseudo codes for generating multiple decision trees and the corresponding index rules are shown in Fig. 5. The function, *new_divide*, traces the number of replicas for each rule and determines whether a rule should be stored in a new decision tree. The function, *new_construct*, repeatedly calls the *new_divide* function if there are rules to be stored in a new decision tree.

We use the same example in Table 2 to illustrate the new procedure for generating index rules by setting both expansion factor and *binth* to two. Fig. 6 shows the initial decision tree before the removal of the highly replicated rules. This decision tree has the same first three layers as the one in Fig. 3 since none of the rules has more than two replicas. In the fourth layer, both rules, F_{10} and F_{11}, generate the third replicas to activate rule removal. Both

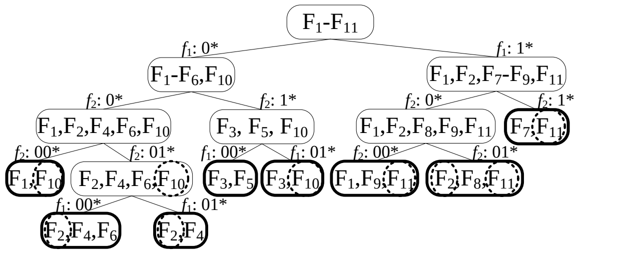


Fig. 6. The initial decision tree for the rule set in Table 2 before removing highly replicated rules (denoted by dotted circles).

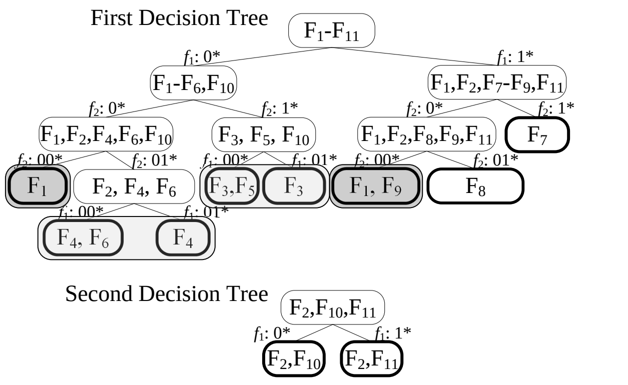


Fig. 7. Decision trees for the rule set in Table 2. The light shade represents sibling nodes to be merged into one leaf node. The dark shade represents two non-sibling leaf nodes to be merged.

rules are removed from the nodes in the fourth layer and leaf nodes in the above layers. For example, F_{11} is removed from two fourth-layer nodes which originally have three rules. Both nodes become leaf nodes after removing F_{11}. F_{11} is also removed from a leaf node in the third layer, and, as a result, only one node in the fourth layer is going to be split further. In the fifth layer, F_2, which has a replica, is to be removed. Finally, the construction of the first decision tree is complete with eight original rules stored, as shown in Fig. 7. In summary, three rules are removed from the first decision tree, namely, F_2, F_{10} and F_{11}, and are relocated to the second decision tree.

After removing some rules from a decision tree, several leaf nodes could be merged to decrease the number of index rules. The procedure of merging two leaf nodes involves comparing two leaf nodes in the same layer. The following two conditions are satisfied for the merge to occur. First, the field specifications of the index rules of both leaf nodes can be merged into one ternary string. Second, the total number of distinct rules in both nodes is less than the *binth* value.

As shown in the first decision tree in Fig. 7, the nodes in light shade can be merged since the total number of distinct rules is equal to two. These nodes can be detected easily because they have the same parent nodes. Another two leaf nodes (denoted by dark shade), which correspond to index rules, "$\langle 0*, 00*\rangle \rightarrow F_1\langle *, 0010\rangle$" and "$\langle 1*, 00*\rangle \rightarrow F_1\langle *, 0010\rangle, F_9\langle 111*, 00*\rangle$", can also be merged into "$\langle *, 00*\rangle \rightarrow F_1\langle *, 0010\rangle, F_9\langle 111*, 00*\rangle$", although they originate from different parent nodes. In Fig. 7, the number of leaf nodes can be reduced to seven, where the optimal number of leaf nodes is six.

The index rules of leaf nodes from different layers have at least one field with different bit masks. In other words, merging the leaf nodes cannot yield a TCAM-friendly index rule, e.g. the leaf nodes storing F_7 and F_8 in Fig. 7. A technique of rule expansion presented in [40] can be used to lower the cost of storing non-TCAM-friendly index rules. For example, the index rules of F_7 and F_8 are merged into one: "$\langle 1*, *\rangle \rightarrow F_7\langle 111*, 11*\rangle, F_8\langle 101*, 01*\rangle$", whose f_2 field is expanded from range [0100:1111] to [0000:1111]. The new index rule partially overlaps with the index rule of F_1 and F_9. To preserve the search accuracy, the new index rule must be placed behind that of F_1 and F_9. Since an expanded index rule could overlap with many others, this approach would lead to a considerable cost of maintaining the sequence of index rules. In this study, we do not implement the rule expansion technique.

Because of a heavy reliance on decision trees, our scheme sometimes requires extra TCAM accesses. Although the tradeoff between storage efficiency and speed performance is inevitable, we believe that maintaining storage efficiency is particularly critical for scalable multi-match packet classification. The reason is that storage efficiency also affects speed performance and energy consumption in practice because a larger memory chip, especially for TCAM, usually has worse performance.

4.3 Search Procedure

Our procedure of generating multiple decision trees ensures that each rule is only stored in one decision tree. As a result, the index rules of all decision trees must be searched for all possible matches. The index rules from different decision trees may overlap with each other. To produce all matching index rules stored in a TCAM chip, we need a multi-match approach to search for the index rules in TCAM. Unlike performing multi-match packet classification upon the rules with varied number of matches, the number of matching index rules is always equal to the number of generated decision trees and is usually less than the maximum number of matching rules.

We use a bitmap to distinguish the index rules from different decision trees. Assume that there are d decision trees. Each bitmap has d bits which are initially set to "don't care". For the i_{th} decision tree, the i_{th} bit of its bitmap is set to one. The bitmap is then appended to all index rules of the i_{th} decision tree.

To yield all matching index rules, one d-bit bitmap is appended to the search key in each TCAM access. Initially, all bits of the bitmap are set to one to compare all index rules. Assume that an index rule from the j_{th}

TABLE 3
Index rules for the rule set in Table 2.

Index Rule	TCAM	SRAM
I_1	$\langle *, 00*, 1* \rangle$	$F_1\langle *, 0010 \rangle, F_9\langle 111*, 00* \rangle$
I_2	$\langle 0*, 01*, 1* \rangle$	$F_4\langle 0*, 01* \rangle, F_6\langle 000*, 01* \rangle$
I_3	$\langle 0*, 1*, 1* \rangle$	$F_3\langle 0*, 1* \rangle, F_5\langle 00*, 11* \rangle$
I_4	$\langle 1*, 01*, 1* \rangle$	$F_8\langle 101*, 01* \rangle$
I_5	$\langle 1*, 1*, 1* \rangle$	$F_7\langle 111*, 11* \rangle$
I_6	$\langle 0*, *, *1 \rangle$	$F_2\langle *, 0111 \rangle, F_{10}\langle 0111, * \rangle$
I_7	$\langle 1*, *, *1 \rangle$	$F_2\langle *, 0111 \rangle, F_{11}\langle 1111, * \rangle$

decision tree matches the search key. In the next iteration, the j_{th} bit of the bitmap is set to zero so that the same index rule will not match the search key again. Since there is exactly one matching index rule in a decision tree, each packet classification requires d iterations of TCAM accesses. The original rules of all matching index rules stored in SRAM are then compared to determine the matching rules.

The comparisons of original rules stored in SRAM can be carried out by additional hardware or the processor embedded in a line card. In [29], a circuit for processing 512-bit SRAM words is synthesized using a 0.18μm library. The circuit includes multiple prefix and range comparators. Its consumed energy is 0.12 nJ per operation, which is about the equivalent to one 2-Mbit SRAM access in Fig. 1. The rule-matching circuit for our scheme can be yielded by modifying the previous circuit. On the other hand, a line card could be equipped with a processor for assisting packet processing [41] or performing software-based packet forwarding [42]. The processor retrieves the addresses of the possibly matching rules from the TCAM-based forwarding engine and accesses the original rules stored in the SRAM of the line card for further comparisons.

In Table 3, we list the index rules from both decision trees in Fig. 7. Each index rule stored in TCAM consists of three fields: f_1, f_2 and a bitmap for identifying its decision tree. The original rules along with their specifications are stored in the corresponding SRAM word. For an incoming packet $\langle 0111, 0111 \rangle$, the search procedure generates a search key, $\langle 0111, 0111, 11 \rangle$, to access TCAM. Assume that the index rules are stored in TCAM in an ascending order of their identifiers. An index rule, I_2, matches the search key. Its rules, F_4 and F_6, stored in the corresponding SRAM word are then compared, where only F_4 matches the search key. Since I_2 belongs to the first decision tree, the search key is updated to $\langle 0111, 0111, 01 \rangle$ to avoid matching I_2 again. In the second TCAM access, I_6 matches the search key so that F_2 and F_{10} stored in SRAM are compared. Finally, three matching rules, F_4, F_2 and F_{10}, are yielded.

The index rules generated by our algorithm have the same length as a typical five-field rule, which occupies 104 bits. There are 40 unused bits in a 144-bit TCAM entry. Our scheme usually generates three to seven decision trees and does not require any extra bit for range encoding. There are quite enough spare bits reserved for rule updates or specifying new fields. Our approach of using a bitmap for searching different set of index rules also avoids the cost of moving TCAM entries in updating rules because of two properties. First, the index rules of the same decision tree do not overlap with each other. They can be stored in TCAM arbitrarily. Second, the index rules from different decision trees can also be arbitrarily stored in TCAM since the bits for identifying different decision trees do not have any implicit ordering. Both properties could simplify the procedure of updating rules since the insertion and deletion of index rules do not involve the other TCAM entries.

4.4 Update Procedure

The procedure for updating rules stored in the proposed data structure is described in this section. The rules in a router vary to accommodate updated configurations. Rule updates include insertion and deletion. The proposed scheme can fulfill rule deletion by removing the deleted rules from the corresponding SRAM entries without modifying TCAM entries. After removing a rule, the update procedure should check whether the original index rule can be merged with another one, as mentioned in Section 4.2. If the answer is affirmative, then both index rules are merged into one and their original rules are collected in the same SRAM word. Merging two index rules could ensure that the storage efficiency of the SRAM is always higher than 50%.

The update procedure for rule insertion is then described. To facilitate a rule insertion, the first step starts by randomly generating an address which matches the inserted rule to access TCAM. For each decision tree, the address has exactly one matching index rule. By comparing the inserted rule and the matching index rules, we can determine whether the new rule is enclosed by an index rule. If there exists such index rules, then the new rule can be stored in the available space of the SRAM entry corresponding to the enclosure index rule. If free SRAM space is not available, then the *divide* function described in Fig. 2 is used to test whether the rules in the SRAM entry and the new rule can be properly stored in two child nodes. If the answer is positive, then two new index rules for the new leaf nodes are generated and the one corresponding to the original leaf node is removed. The new index rules can be stored in any empty TCAM entries indicating that no TCAM entry movement is required.

If there does not exist any enclosure index rule or the *divide* function cannot generate new child nodes for storing the new rule, then the new rule is inserted into the root node of a new decision tree to generate a new index rule like "$\langle *, *, \ldots, * \ldots 1 \rangle \rightarrow new\_rule$". None of the existing TCAM entry is affected. The new decision tree can also be used for storing new rules so that the consumption of spare bits can be smoothed. All data structure should be reconstructed when the update procedure runs out of all spare bits.

5 Performance Evaluation

To conduct the performance evaluation, we implement our algorithm by using GNU C++ to yield the performance metrics of interests. The performance metrics include TCAM and SRAM storage requirements, and search latency and energy consumption per packet classification in the worst case. We also use the energy-delay product to consider energy consumption and speed performance simultaneously. The TCAM storage requirement is measured by the number of occupied 144-bit TCAM entries and the SRAM storage requirement counts the number of stored rules. Each rule stored in SRAM consumes 20 bytes. To keep the cost of SRAM low, the expansion factor is fixed to two. We assume that there are 512 bits in an SRAM word, and each word stores three rules and one pointer for chaining another SRAM word. Multiple sequential SRAM accesses are required if there are more than three rules to be compared. We extract the search latency and energy consumption per access of TCAM and SRAM for different memory sizes in Fig. 1. For each rule set, we choose the parameters of the smallest chip which is larger than the required storage for calculation. With the number of TCAM and SRAM accesses per classification, we can calculate the search latency and energy consumption.

Our performance evaluation consists of two parts. In the first part, we use Snort rules [43] downloaded from http://www.snort.org to depict the performance of our scheme. In the second part, we evaluate the performance of our scheme by using the five-field rule sets acquired from [44]. These rule sets can be downloaded from http://www.arl.wustl.edu/~hs1/PClassEval.html. In both parts, we compare our scheme, MUST, with three previous algorithms, MUD with DIRPE [6], SSA [7], and GI [1]. The storage performance of SSA can be improved by splitting the original rule set into more sets [7]. In our experiment, we collect the numerical results of SSA-2 and SSA-4, where SSA-2 and SSA-4 split the original rule set into two and four sets, respectively. To simplify the comparison, all sets of SSA are stored in a TCAM chip. We use the bitmap mentioned in Section 4.3 to distinguish the rules of different sets. Similar to our scheme, SSA must access each set to yield all matching rules. We do not compare our scheme with the algorithms using proprietary TCAM architectures because both access latency and energy of these architectures cannot be estimated. In addition, the implementation cost of a proprietary TCAM architecture could be too high to conduct an appropriate comparison.

5.1 Snort Rule Sets

There are four versions of Snort rules, 2.9.0.0, 2.9.1.0, 2.9.2.0, and 2.9.3.0, used in our experiments. The numbers of original rules for different versions vary from 888 to 1,147, where a larger set of rules has more rules overlapping with each other. These Snort rules specify few ranges and do not suffer from the cost of range representation in TCAM. However, these rules may specify negation values, e.g. \$EXTERNAL_NET for IP address prefix or !80 for port number. The negation values may result in a great expense of TCAM entries [1]. In [1], a negation removal scheme is proposed to avoid the problem of negation representation. All schemes in our experiments apply negation removal for storage saving.

The first experiment determines a proper *binth* value by testing the Snort rule sets. Since each SRAM access fetches three rules, we set the *binth* value from 9 to 18 with a step value of three. Table 4 summarizes storage performance for Snort rule sets with varying *binth* values, including the numbers of TCAM entries (TCAM.E), rules stored in SRAM (SRAM.R), and decision trees (DT). The number of rules stored in SRAM shows the number of replicated rules and is related to the required SRAM capacity. The number of decision trees determines the bitmap length and the number of TCAM accesses in a classification. Among these rule sets, version 2.9.3.0 has the highest number of overlapping rules and results in the greatest number of decision trees. Snort v2.9.1.0 requires more TCAM entries than v2.9.2.0 and v2.9.3.0 for certain *binth* values. The fluctuation of storage performance is caused by the proposed decision tree which can only half a space in each tree node. The limitation could lead to more leaf nodes if the number of rules overlapping with each other is slightly more than the *binth* value. In general, a large *binth* value can reduce both the numbers of TCAM entries and decision trees. In most cases, a larger *binth* value also reduces the number of rule replicas, but the slower search performance is the tradeoff. For example, a decision tree with a *binth* value 18 requires one TCAM access and six SRAM accesses. As shown in Table 4, all Snort rule sets require only 3 TCAM accesses but 18 SRAM accesses. According to the parameters depicted in Fig. 1, the total latencies of TCAM and SRAM are 1.28ns and 5.12ns, respectively. This setting results in biased access latencies for TCAM and SRAM in a classification. To minimize the overall latency and maximize the benefit of pipelining implementation (if any), we select the smallest *binth* value (9) to balance the access latencies of TCAM and SRAM in a classification. We use the same *binth* value in all experiments.

The numbers of TCAM entries for different algorithms are shown in Fig. 8. GI always consumes TCAM entries to the greatest extent due to the large number of intersections among Snort rules. In some cases, it may need more than 524K TCAM entries (or 72-Mbit TCAM capacity). Both configurations of SSA (SSA-2 and SSA-4) can drastically reduce the number of TCAM entries by splitting a rule set, but they still need a more considerable number of entries than MUD and our scheme. MUD incurs extra TCAM entries because of range representation, even when DIRPE is employed. Among these algorithms, our scheme needs the least number of TCAM entries because our scheme merges geometrically close-by rules, and only index rules are stored in TCAM to result in fewer

TABLE 4
Storage performance for Snort rule sets with different *binth* values.

SNORT	Original	Max.	*binth*=9			*binth*=12			*binth*=15			*binth*=18		
Version	Rules	Matches	TCAM.E	SRAM.R	DT	TCAM.E	SRAM.R	DT	TCAM.E	SRAM.R	DT	TCAM.E	SRAM.R	DT
2.9.0.0	888	18	396	1,742	4	304	1,802	3	169	1,403	3	113	1,152	3
2.9.1.0	1,058	19	577	2,142	6	456	2,233	4	363	2,121	3	225	1,608	3
2.9.2.0	1,097	21	557	2,152	6	493	2,263	5	407	2,311	5	191	1,817	3
2.9.3.0	1,147	22	413	1,648	7	475	2,294	5	342	2,199	4	244	2,243	3

TCAM.E: TCAM Entries, SRAM.R: Rules stored in SRAM, DT: Decision Trees

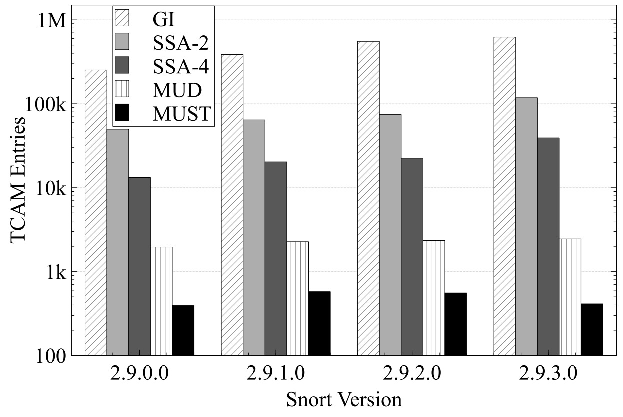


Fig. 8. TCAM entries of different algorithms.

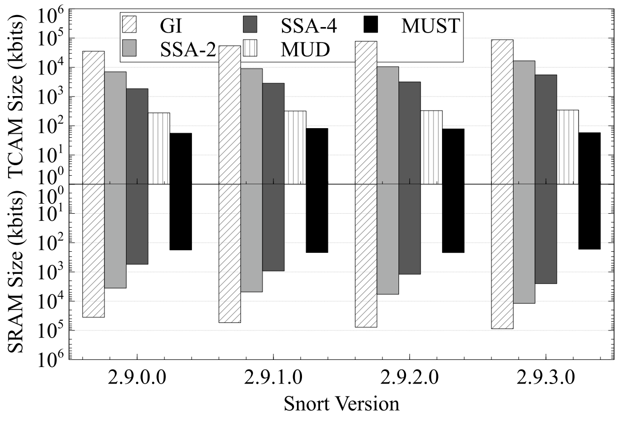


Fig. 9. Memory requirements of different algorithms.

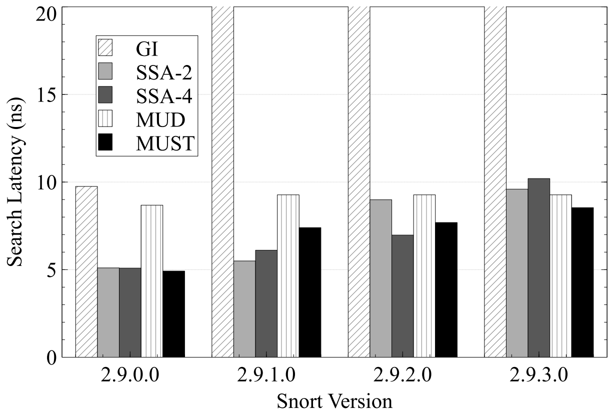


Fig. 10. Search latency of different algorithms.

TCAM entries. Although the existing algorithms only need TCAM accesses to accomplish packet classification, their overall costs are higher than our scheme.

We further depict the required memory space for TCAM and SRAM in Fig. 9, where the required TCAM size correlates with the number of required TCAM entries. The previous algorithms only store rule indices in SRAM, but their SRAM requirements are quite different. GI and SSA need to store one rule list for each TCAM entry. When a rule overlaps with numerous rules, the rule index is repeated in different rule lists to incur considerable storage. Fig. 9 shows that the SRAM requirements of GI and SSA are highly related to their TCAM requirements. The SRAM requirement of MUD is too small to be illustrated in Fig. 9 because it only stores one rule index in SRAM for each TCAM entry. Our algorithm stores the overlapping rules in one or more SRAM entries without generating any match condition. The advantage of our scheme is obvious when there are many overlapping rules. Although our algorithm needs more SRAM storage than MUD, MUD needs fourfold or more TCAM storage than our scheme. As a result, the overall storage performance of our scheme is better than that of MUD.

The search latency of each algorithm is shown in Fig. 10. Although GI only needs one TCAM access and one SRAM access to yield all matching rules, it has the longest search latency because of the slow speed of a large TCAM chip. SSA drastically reduces the number of TCAM entries of GI to curtail the latency of each TCAM access. Although SSA needs more TCAM accesses than GI, the search latency of SSA is much shorter than that of GI. The search latency of SSA-2 might be longer than that of SSA-4 also because SSA-2 needs a much larger TCAM chip than SSA-4 does. Among these algorithms, MUD needs the most TCAM accesses. The minimum number of TCAM accesses for MUD is the same as the number of matching rules for a packet. MUD may require multiple TCAM accesses for a certain range of TCAM entries to further extend the search latency. The number of TCAM accesses of our scheme is tied to the number of generated decision trees. As shown in Table 4, our scheme generates four to seven decision trees. Since one TCAM access is required for each tree, our scheme may need more TCAM accesses and search latency than SSA for some Snort versions (2.9.1.0 and 2.9.2.0). In the worst case, our scheme is 25% slower than SSA. In the

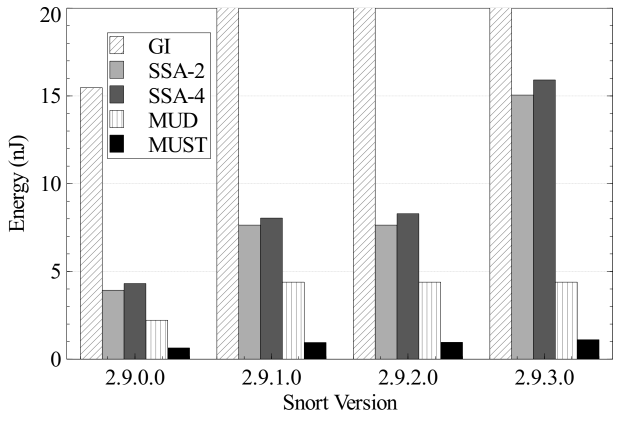


Fig. 11. Energy consumption of different algorithms.

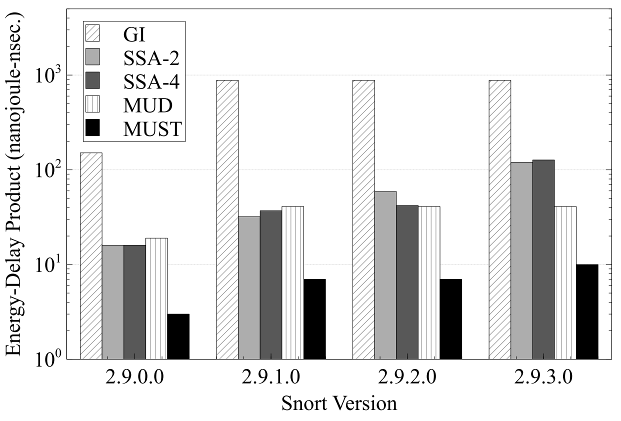


Fig. 12. Energy-delay product of different algorithms.

cases where our scheme outperforms SSA, the rate of improvement varies from 3 to 20%.

We illustrate the energy consumption of different algorithms in Fig. 11. Both TCAM chip size and number of accesses dominate the power consumption in each classification. Similarly, while GI still consumes the highest level of energy among all algorithms, SSA significantly reduces energy consumption. SSA-2 always consumes less energy than SSA-4 because of fewer TCAM accesses. Although MUD needs the most TCAM accesses, its low memory requirements also keep the overall energy consumption low. Our scheme consumes the lowest level of energy among these algorithms because it needs fewer TCAM entries and accesses than MUD.

We use a metric, energy-delay product (EDP), to investigate the energy efficiency of different algorithms by taking into account of search latency and energy consumption per classification [45]. A smaller EDP value indicates a better overall performance. As shown in Fig. 12, GI has the worst EDP performance because of large search latency and energy consumption. Although SSA-2 always consumes less energy than SSA-4, SSA-4 outperforms SSA-2 by lowering search latency in Snort v2.9.2.0. In general, both SSA-2 and SSA-4 have similar EDP performance. Although both search latency and energy consumption of MUD are quite different from those of SSA, their EDP performance is similar for most cases, except for Snort v2.9.3.0. Our scheme has the best performance throughout all cases by providing shorter search latency and lower energy consumption simultaneously. The results demonstrate that although our algorithm requires extra SRAM accesses, the extra latency and energy consumption can be easily offset by the benefits of a smaller TCAM size.

5.2 ClassBench Rule Databases

Next, we use ClassBench rule databases for performance evaluation. There are three different types of rule sets, access control list (ACL), firewalls (FW) and IP chain (IPC). Each type includes one real database and three synthetic ones generated by ClassBench [46]. The real databases have 752, 269 and 1550 rules. The sizes of the synthetic databases vary from 1K to 10K. The rules of the FW databases specify the most wildcard fields and those of the ACL databases have only wildcard specifications in the source-port field. The detailed characteristics of these rule databases can be found in [44]. Unlike the Snort rules, the rules in these databases do not specify negation but indicate many arbitrary ranges with high costs of range-to-prefix conversion. Since these databases vary in size and overlapping rules, they can be used to test the scalability of our scheme.

We show both SRAM and TCAM memory storage in Fig. 13. Because the ACL databases have the least overlapping rules, their TCAM entries are mainly contributed by range-to-prefix conversion. The FW databases have the most intersections. Even SSA-4 cannot store the 10K-rule FW database in a 72-Mbit TCAM chip. Although MUD uses DIRPE to decrease the cost of range representation, the cost is inevitable. Our approach effectively reduces the number of TCAM entries by synthesizing index rules and avoiding the cost of range representation. Our scheme needs more SRAM storage than GI and SSA for the synthetic ACL databases because of storing complete rule specifications. When the number of overlapping rules increases in the other databases, our scheme benefits from lowering the number of TCAM entries to reduce SRAM storage by avoiding the costs of range representation and intersection rules. MUD needs fewer SRAM storage and more TCAM entries than our scheme. By assuming the cost of TCAM is thirty times of that of SRAM [3], our scheme requires only $5 \sim 54$ percent (30 percent in average) storage cost of MUD.

We use energy-delay product in Fig. 14 to evaluate and compare the performance of different schemes. When the number of overlapping rules is small, GI has the best EDP performance since it only accesses TCAM once. However, when the number of overlapping rules increases, both increment of search latency and energy consumption would sharply increase the EDP value of GI. SSA can smooth the performance degradation by drastically reducing pseudo rules, which means it has better scalability than GI. The search latency and energy

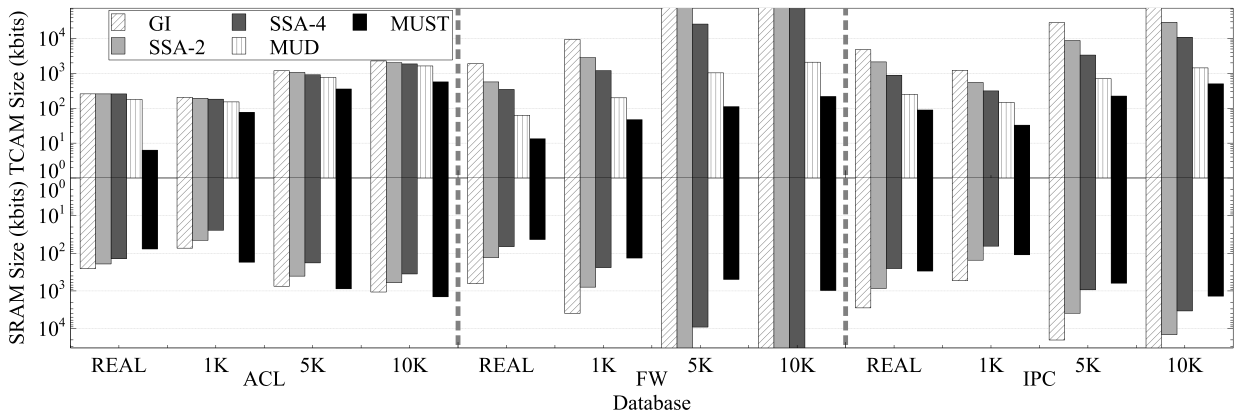


Fig. 13. Memory requirements of different algorithms for ClassBench databases.

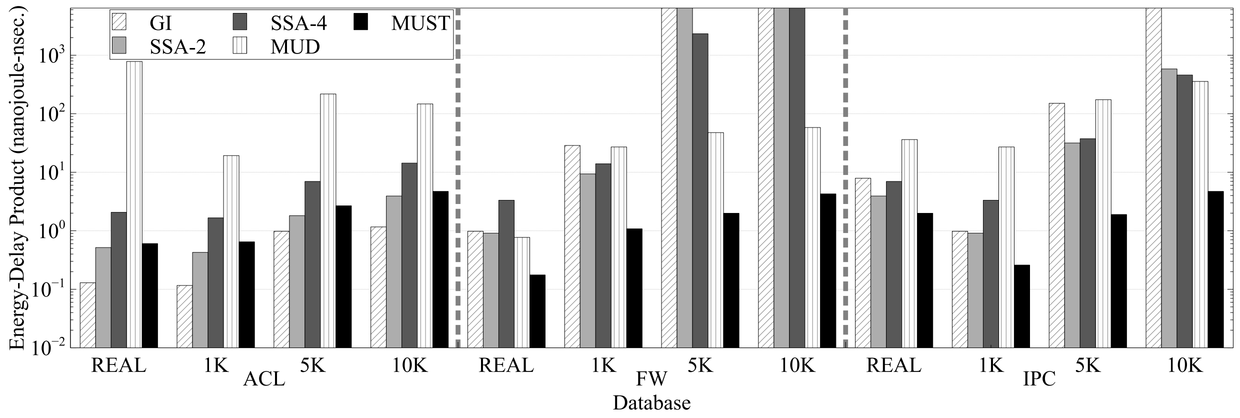


Fig. 14. Energy-delay product of different algorithms for ClassBench databases.

consumption of MUD is tied to the number of TCAM accesses. As a result, MUD cannot benefit from its efficient TCAM storage to outperform GI and SSA in most cases. Our scheme has both advantages of low TCAM storage and moderate number of TCAM accesses. Although our scheme has worse EDP performance than that of GI and SSA for ACL databases, it outperforms both schemes for FW and IPC databases, which have more overlapping rules. The results demonstrate that a scheme based on pseudo rules, e.g. GI or SSA, is suitable for the databases with only few overlapping rules since the extra TCAM entries may not incur significant increment of access latency. Our scheme has better performance for the databases where a packet may match numerous rules, e.g. Snort and FW/IPC databases.

6 Conclusions

Multi-match packet classification enables the services such as deep packet inspection, transparent monitoring and usage-based accounting. Fulfilling multi-match packet classification solely based on TCAM may suffer from extra TCAM entries, accesses, or both. In this paper, we present a novel scheme by combing TCAM and SRAM to reduce TCAM entries. Our scheme is based on the observation that SRAM is larger, faster, and more power efficient than TCAM. We offload TCAM's overheads by using SRAM to store rules for comparison. We use binary decision trees to generate TCAM-compatible index rules and store the original rule specifications in SRAM. Our scheme thus avoids the cost of range representation. With the algorithm which generates multiple decision trees, the number of rule replicas can be manipulated so that both TCAM and SRAM storage can be maintained at a reasonable level. While both access latency and power consumption of TCAM may expand exponentially as the chip capacity increases, the experimental results show that our scheme achieves significant improvement for the required TCAM entries. In summary, our scheme can effectively improve speed performance and energy efficiency simultaneously, which makes it feasible and scalable for multi-match packet classification. In our future work, we attempt to apply multi-match packet classification in OpenFlow switches, where multiple flow tables in a pipeline can be merged to shorten the search latency.

Acknowledgments

This work was supported in part by the Ministry of Science and Technology of Taiwan, R.O.C., under Grant MOST 103-2221-E-005-061 and Grant MOST 103-2218-E-002-033.

References

[1] F. Yu, R. H. Katz, and T. Lakshman, "Efficient Multimatch Packet Classification and Lookup with TCAM," *IEEE Micro*, vol. 25, no. 1, pp. 50–59, 2005.

[2] *Cisco 4700 Series Application Control Engine Appliance Device Manager GUI Configuration Guide*, Cisco, 2010. [Online]. Available: http://www.cisco.com/en/US/docs/app_ntwk_services/data_center_app_services/ace_appliances/vA4_1_0/configuration/device_manager/guide/dmgui_cfgd.pdf

[3] D. E. Taylor, "Survey and Taxonomy of Packet Classification Techniques," *ACM Computing Survey*, vol. 37, no. 3, pp. 238–275, 2005.

[4] M. Faezipour and M. Nourani, "Wire-Speed TCAM-Based Architectures for Multimatch Packet Classification," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 5–17, 2009.

[5] R. K. Montoye, "Apparatus for storing Don't Care in a content addressable memory cell." U.S. Patent 5,319,590, 1994.

[6] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 193–204, 2005.

[7] F. Yu, T. V. Lakshman, M. A. Motoyama, and R. H. Katz, "Efficient Multimatch Packet Classification for Network Security Applications," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, pp. 1805–1816, 2006.

[8] R. Shen, X. Li, and H. Li, "A space- and power-efficient multi-match packet classification technique combining tcams and srams," *The Journal of Supercomputing*, pp. 1–20, 2014.

[9] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *Proceedings of the ACM/SIGDA FPGA 2005*, pp. 238–245.

[10] T. Lakshman and D. Stidialis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM SIGCOMM'98*, pp. 203–214.

[11] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using fpgas," in *Proc. SPAA'2009*, pp. 188–196.

[12] A. Kesselman, K. Kogan, S. Nemzer, and M. Segal, "Space and Speed Tradeoffs in TCAM Hierarchical Packet Classification," in *Proceedings of the IEEE Sarnoff Symposium 2008*, pp. 1–6.

[13] Z. Kai, H. Che, W. Zhijun, L. Bin, and Z. Xin, "DPPC-RE: TCAM-based Distributed Parallel Packet Classification with Range Encoding," *IEEE Transactions on Computers*, vol. 55, no. 8, pp. 947–961, 2006.

[14] H. Che, Z. Wang, K. Zheng, and B. Liu, "DRES: Dynamic Range Encoding Scheme for TCAM Coprocessors," *IEEE Transactions on Computers*, vol. 57, no. 7, pp. 902–915, 2008.

[15] Y.-K. Chang, C.-I. Lee, and C.-C. Su, "Multi-field Range Encoding for Packet Classification in TCAM," in *Proceedings of the IEEE INFOCOM 2011*, pp. 196–200.

[16] X. He, J. Peddersen, and S. Parameswaran, "LOP_RE: Range Encoding for Low Power Packet Classification," in *Proceedings of the IEEE LCN 2009*, pp. 137–144.

[17] A. Bremler-Barr and D. Hendler, "Space-Efficient TCAM-Based Classification Using Gray Coding," in *Proceedings of the IEEE INFOCOM 2007*, pp. 1388–1396.

[18] A. Bremler-Barr, D. Hay, and D. Hendler, "Layered interval codes for tcam-based classification," in *Proceedings of the IEEE INFOCOM 2009*, pp. 1305–1313.

[19] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems," in *Proceedings of the USENIX Security 2010*, pp. 8–8.

[20] R. Wei, Y. Xu, and H. J. Chao, "Block Permutations in Boolean Space to Minimize TCAM for Packet Classification," in *Proceedings of the IEEE INFOCOM 2012*, pp. 2561–2565.

[21] C. R. Meiners, A. X. Liu, and E. Torng, "Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 20, no. 2, pp. 488–500, 2012.

[22] R. Cohen and D. Raz, "Simple Efficient TCAM Based Range Classification," in *Proceedings of the IEEE INFOCOM 2010*, pp. 1–5.

[23] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel, "Split: Optimizing Space, Power, and Throughput for TCAM-Based Classification," in *Proceedings of the ACM/IEEE ANCS 2011*, pp. 200–210.

[24] T. Banerjee-Mishra, S. Sahni, and G. Seetharaman, "PC-DUOS: Fast TCAM lookup and update for packet classifiers," in *Proceedings of the IEEE ISCC 2011*, pp. 265–270.

[25] T. Banerjee-Mishra and S. Sahni, "PETCAM-A Power Efficient TCAM Architecture for Forwarding Tables," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 3–17, 2012.

[26] W. Lu and S. Sahni, "Low-Power TCAMs for Very Large Forwarding Tables," *IEEE/ACM Transactions on Networking*, vol. 18, no. 3, pp. 948–959, 2010.

[27] T. Banerjee-Mishra and S. Sahni, "DUOS - Simple Dual TCAM Architecture for Routing Tables with Incremental Update," in *Proceedings of the IEEE ISCC 2010*, pp. 503–508.

[28] O. Rottenstreich and I. Keslassy, "Worst-case TCAM rule expansion," in *Proc. IEEE INFOCOM'2010*, pp. 456–460.

[29] T. Banerjee, S. Sahni, and G. Seetharaman, "PC-TRIO: A Power Efficient TCAM Architecture for Packet Classifiers," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 1104–1118, 2015.

[30] D. Shah and P. Gupta, "Fast Updating Algorithms for TCAMs," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, Jan. 2001.

[31] H. Song and J. S. Turner, "Fast filter updates for packet classification using tcam," in *Proc. GLOBECOM'06*, pp. 1–6.

[32] Z. Wang, H. Che, M. Kumar, and S. K. Das, "CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1602–1614, Dec. 2004.

[33] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, 2000.

[34] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM'2003*, pp. 213–224.

[35] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: Optimizing Packet Classification for Memory and Throughput," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 207–218, 2010.

[36] H. Lu and S. Sahni, "O(log W) Multidimensional Packet Classification," *IEEE/ACM Transactions on Networking*, vol. 15, no. 2, pp. 462–472, 2007.

[37] Y.-C. Cheng and P.-C. Wang, "Packet Classification Using Dynamically Generated Decision Trees," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 582–586, 2015.

[38] B. Agrawal and T. Sherwood, "Ternary CAM Power and Delay Model: Extensions and Uses," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 5, pp. 554–564, 2008.

[39] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A Tool to Understand Large Caches," HP Tech Report HPL-2009, Tech. Rep., 2009.

[40] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet Classifiers in Ternary CAMs Can Be Smaller," in *Proc. SIGMETRICS 2006*, pp. 311–322.

[41] *The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor*, Cisco, 2008. [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.pdf

[42] R. Ohlendorf, "A Network Processor Architecture with Application-optimized Reconfigurable Processing Paths (FlexPath NP)," Ph.D. dissertation, Technische Universitt Mnchen, 2010.

[43] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proceedings of LISA 1099*, Berkeley, pp. 229–238.

[44] H. Song and J. S. Turner, "Toward advocacy-free evaluation of packet classification algorithms," *IEEE Transactions on Computers*, vol. 60, no. 5, pp. 723–733, 2011.

[45] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok, "Energy and Performance Evaluation of Lossless File Data Compression on Server Systems," in *Proc. SYSTOR 2009*, pp. 4:1–4:12.

[46] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," in *Proceedings of the IEEE INFOCOM 2005*, vol. 3, pp. 2068–2079.