# Bit Weaving: A Non-prefix Approach to Compressing Packet Classifiers in TCAMs

Chad R. Meiners Alex X. Liu Eric Torng
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48823, U.S.A.
{meinersc, alexliu, torng}@cse.msu.edu

***Abstract*—Ternary Content Addressable Memories (TCAMs) have become the de facto standard in industry for fast packet classification. Unfortunately, TCAMs have limitations of small capacity, high power consumption, high heat generation, and high cost. The well-known range expansion problem exacerbates these limitations as each classifier rule typically has to be converted to multiple TCAM rules. One method for coping with these limitations is to use compression schemes to reduce the number of TCAM rules required to represent a classifier. Unfortunately, all existing compression schemes only produce prefix classifiers. Thus, they all miss the compression opportunities created by non-prefix ternary classifiers.**

**In this paper, we propose bit weaving, the first non-prefix compression scheme. Bit weaving is based on the observation that TCAM entries that have the same decision and whose predicates differ by only one bit can be merged into one entry by replacing the bit in question with *. Bit weaving consists of two new techniques, *bit swapping* and *bit merging*, to first identify and then merge such rules together. The key advantages of bit weaving are that it runs fast, it is effective, and it is composable with other TCAM optimization methods as a pre/post-processing routine.**

**We implemented bit weaving and conducted experiments on both real-world and synthetic packet classifiers. Our experimental results show the following: (i) bit weaving is an effective stand-alone compression technique (it achieves an average compression ratio of $23.6\%$) and (ii) bit weaving finds compression opportunities that other methods miss. Specifically, bit weaving improves the prior TCAM optimization techniques of TCAM Razor and Topological Transformation by an average of $12.8\%$ and $36.5\%$, respectively.**

## I. Introduction

### A. Background on TCAM-Based Packet Classification

Packet classification is the core mechanism that enables many networking devices, such as routers and firewalls, to perform services such as packet filtering, virtual private networks (VPNs), network address translation (NAT), quality of service (QoS), load balancing, traffic accounting and monitoring, differentiated services (Diffserv), etc. The essential problem is to compare each packet with a list of predefined rules, which we call a packet classifier, and find the first (i.e., highest priority) rule that the packet matches. Table I shows an example packet classifier of three rules. The format of these rules is based upon the format used in Access Control Lists (ACLs) on Cisco routers. In this paper we use the terms *packet classifiers*, *ACLs*, *rule lists*, and *lookup tables* interchangeably.

Hardware-based packet classification using Ternary Content Addressable Memories (TCAMs) is now the de facto industry standard [1], [9]. TCAM-based packet classification is widely used because Internet routers need to classify every packet on the wire. Although software based packet classification has been extensively studied (see survey paper [24]), these techniques cannot match the wire speed performance of TCAM-based packet classification systems.

| Rule | Source IP | Dest. IP | Source Port | Dest. Port | Protocol | Action |
|---|---|---|---|---|---|---|
| $r_1$ | 1.2.3.0/24 | 192.168.0.1 | [1,65534] | [1,65534] | TCP | accept |
| $r_2$ | 1.2.11.0/24 | 192.168.0.1 | [1,65534] | [1,65534] | TCP | accept |
| $r_3$ | * | * | * | * | * | discard |

TABLE I
AN EXAMPLE PACKET CLASSIFIER

As a traditional random access memory chip receives an address and returns the content of the memory at that address, a TCAM chip works in a reverse manner: it receives content and returns the address of the *first* entry where the content lies in the TCAM in constant time (*i.e.*, a few clock cycles). Exploiting this hardware feature, TCAM-based packet classifiers store a rule in each entry as an array of 0's, 1's, or *'s (*don't-care* values). A packet header (*i.e.*, a search key) matches an entry if and only if their corresponding 0's and 1's match. Given a search key to a TCAM, the hardware circuits compare the key with all its occupied entries in parallel and return the index (or the content, depending on the chip architecture and configuration,) of the first matching entry.

### B. Motivation for TCAM-based Classifier Compression

Although TCAM-based packet classification is currently the *de facto* standard in industry, TCAMs do have several limitations. First, TCAM chips have limited capacity. The largest available TCAM chip has a capacity of 36 megabits (Mb). Smaller TCAM chips are the most popular due to the other limitations of TCAM chips stated below. Second, TCAMs require packet classification rules to be in ternary format. This leads to the well-known range expansion problem, *i.e.*, converting packet classification rules to ternary format results in a much larger number of TCAM rules, which exacerbates the problem of limited capacity TCAMs. In a typical packet classification rule, the three fields of source and destination IP addresses and protocol type are specified as prefixes (*e.g.*, 1011****) where all the *s are at the end of the ternary string, so the fields can be directly stored in a TCAM. However, the remaining two fields of source and destination port numbers are specified in ranges (*i.e.*, integer intervals such as $[1, 65534]$), which need to be converted to one or more prefixes before being stored in a TCAM. This can lead to a

978-1-4244-4634-6/09/$25.00 ©2009 IEEE

significant increase in the number of TCAM entries needed to encode a rule. For example, 30 prefixes are needed to represent the single range $[1, 65534]$, so $30 \times 30 = 900$ TCAM entries are required to represent the single rule $r_1$ in Table I. Third, TCAM chips consume lots of power. The power consumption of a TCAM chip is about 1.85 Watts per Mb [2]. This is roughly 30 times larger than a comparably sized SRAM chip [10]. TCAMs consume lots of power because every memory access searches the entire active memory in parallel. That is, a TCAM is not just memory, but memory and a (very fast) parallel search system. Fourth, TCAMs generate lots of heat due to their high power consumption. Fifth, a TCAM chip occupies a *large footprint* on a line card. A TCAM chip occupies 6 times (or more) board space than an equivalent capacity SRAM chip [10]. For networking devices such as routers, area efficiency of the circuit board is a critical issue. Finally, TCAMs are *expensive*, costing hundreds of dollars even in large quantities. TCAM chips often cost more than network processors [11]. The high price of TCAMs is mainly due to their large die area, not their market size [10]. Power consumption, heat generation, board space, and cost lead to system designers using smaller TCAM chips than the largest available. For example, TCAM components are often restricted to at most 10% of an entire board's power budget, so a 36 Mb TCAM may not be deployable on many routers due to power consumption reasons.

While TCAM-based packet classification is the current industry standard, the above limitations imply that existing TCAM-based solutions may not be able to scale up to meet the future packet classification needs of the rapidly growing Internet. Specifically, packet classifiers are growing rapidly in size and width due to several causes. First, the deployment of new Internet services and the rise of new security threats lead to larger and more complex packet classification rule sets. While traditional packet classification rules mostly examine the five standard header fields, new classification applications begin to examine addition fields such as classifier-id, protocol flags, ToS (type of service), switch-port numbers, security tags, etc. Second, with the increasing adoption of IPv6, the number of bits required to represent source and destination IP address will grow from 64 to 256. The size and width growth of packet classifiers puts more demand on TCAM capacity, power consumption, and heat dissipation.

To address the above TCAM limitations and ensure the scalability of TCAM-based packet classification, we study the following *TCAM-based classifier compression* problem: given a packet classifier, we want to *efficiently generate a semantically equivalent packet classifier that requires fewer TCAM entries*. Note that two packet classifiers are (semantically) equivalent if and only if they have the same decision for every packet. TCAM-based classifier compression helps to address the limited capacity of deployed TCAMs because reducing the number of TCAM entries effectively increases the fixed capacity of a chip. Reducing the number of rules in a TCAM directly reduces power consumption and heat generation because the energy consumed by a TCAM grows linearly with the number of ternary rules it stores [27]. Finally, TCAM-based classifier compression lets us use smaller TCAMs, which results in less power consumption, less heat generation, less board space, and lower hardware cost.

### C. Limitations of Prior Art

All prior TCAM-based classifier compression schemes (*i.e.*, [3], [6], [7], [13], [16], [23]) suffer from one fundamental limitation: they only produce prefix classifiers, which means they all miss some opportunities for compression. A prefix classifier is a classifier in which every rule is a prefix rule. In a prefix rule, each field is specified as a prefix bit string (*e.g.*, 01**) where *s all appear at the end. In a ternary rule, each field is a ternary bit string (*e.g.*, 0**1) where * can appear at any position. Every prefix rule is a ternary rule, but not vice versa. Because all previous compression schemes can only produce prefix rules, they miss the compression opportunities created by non-prefix ternary rules.

### D. Our Bit Weaving Approach

In this paper, we propose *bit weaving*, a new TCAM-based classifier compression scheme that is not limited to producing prefix classifiers. The basic idea of bit weaving is simple: adjacent TCAM entries that have the same decision and have a hamming distance of one (*i.e.*, differ by only one bit) can be merged into one entry by replacing the bit in question with *. Bit weaving applies two new techniques, *bit swapping* and *bit merging*, to first identify and then merge such rules together. Bit swapping first cuts a rule list into a series of partitions. Within each partition, a single permutation is applied to each rule's predicate to produce a reordered rule predicate, which forms a single prefix where all *'s are at the end of the rule predicate. This single prefix format allows us to use existing dynamic programming techniques [16], [23] to find a minimal TCAM table for each partition in polynomial time. Bit merging then finds and merges mergeable rules from each partition. After bit merging, we revert all ternary strings back to their original bit permutation to produce the final TCAM table. We name our solution bit weaving because it manipulates bit ordering in a ternary string much like a weaver manipulates the position of threads.

The example in Figure 1 shows that bit weaving can further compress a minimal prefix classifier. The input classifier has 5 prefix rules with three decisions (0, 1, and 2) over two fields $F_1$ and $F_2$, where each field has two bits. Bit weaving compresses this minimal prefix classifier with 5 rules down to 3 ternary rules as follows. First, it cuts the input prefix classifier into two partitions which are the first two rules and the last three rules, respectively. Second, it swaps bit columns in each partition to make the two-dimensional rules into one-dimension prefix rules. In this example, in the second partition, the second and the fourth columns are swapped. We call the above two steps *bit swapping*. Third, we treat each partition as a one-dimensional prefix rule list and generate a minimal prefix representation. In this example, the second partition is minimized to 2 prefix rules. Fourth, in each partition, we detect and merge rules that differ by a single bit. In the first partition, the two rules are merged. We call this step *bit merging*. Finally, we revert each partition back to its original bit order. In this example, for the second partition after minimization, we swap

the second and the fourth columns again to recover the original bit order. The final output is a ternary packet classifier with only 3 rules.

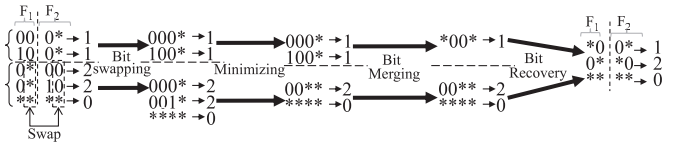


Fig. 1. Example of the bit weaving approach

### E. Technical Challenges

To implement bit weaving, we must solve several challenging technical problems. First, we need to develop an algorithm that partitions a rule list into the least number of partitions. Second, we must develop an algorithm that permutes the bit columns within each partition to produce one-dimensional prefix rule lists. Third, we must adapt existing one-dimensional prefix rule list minimization algorithms (*i.e.*, [16], [23]) to minimize *incomplete* one-dimensional rule lists. A rule list is complete if and only if for any packet, the list has a rule that the packet matches. Finally, we must develop algorithms to detect and then merge mergeable rules within each partition.

### F. Our Contributions

Our bit weaving approach has many significant benefits. First, it is the first TCAM compression method that can create non-prefix classifiers. All previous compression methods [6], [12], [14], [16] generate only prefix classifiers. This restriction to prefix format may miss important compression opportunities. Second, it is the first efficient compression method with a polynomial worst-case running time with respect to the number of fields in each rule. Third, it is orthogonal to other techniques, which means that it can be run as a pre/post-processing routine in combination with other compression techniques. In particular, bit weaving complements TCAM Razor [16] nicely. In our experiments on real-world classifiers, bit weaving outperforms TCAM Razor on classifiers that do not have significant range expansion. Fourth, it supports fast incremental updates to classifiers.

### G. Summary of Experimental Results

We implemented bit weaving and conducted experiments on both real-world and synthetic packet classifiers. Our experimental results show that bit weaving is an effective stand-alone compression technique as it achieves an average compression ratio of $23.6\%$ and that bit weaving finds compression opportunities that other methods miss. Specifically, bit weaving improves the prior TCAM optimization techniques of TCAM Razor [16], and Topological Transformation [18] by an average of $12.8\%$ and $36.5\%$, respectively.

The rest of this paper proceeds as follows. We start by reviewing related work in Section II. We define bit swapping in Section III and bit merging in Section IV. In Section V, we discuss how bit weaving supports incremental updates, how bit weaving can be composed with other compression methods, and the complexity bounds of bit weaving. We show our experimental results on both real-life and synthetic packet classifiers in Section VI, and we give concluding remarks in Section VII.

## II. Related Work

TCAM-based packet classification systems have been widely deployed due to their $O(1)$ classification time. This has led to a significant amount of work that explores ways to efficiently store packet classifiers within TCAMs. Prior work falls into three broad categories: classifier compression, range encoding, and circuit and hardware modification.

### A. Classifier Compression

Classifier compression converts a given packet classifier to another semantically equivalent packet classifier that requires fewer TCAM entries. Several classifier compression schemes have been proposed [3], [6], [7], [12], [16], [23]. The work is either focused on one-dimensional and two dimensional packet classifiers [3], [7], [23], or it is focused on compressing packet classifiers with more than 2 dimensions [6], [12], [14], [16]. Liu and Gouda proposed the first algorithm for eliminating all the redundant rules in a packet classifier [12], and we presented a more efficient redundancy removal algorithm [14]. Dong *et al.* proposed schemes to reduce range expansion by repeatedly expanding or trimming ranges to prefix boundaries [6]. Their schemes use redundancy removal algorithms [12] to test whether each modification changes the semantics of the classifier. We proposed a greedy algorithm that finds locally minimal prefix solutions along each field and combines these solutions into a smaller equivalent prefix packet classifier [16].

Bit weaving differs from these previous efforts in that it is the first classifier minimization algorithm that produces equivalent non-prefix packet classifiers given an arbitrary number of fields and decisions. Furthermore, bit weaving is the first algorithm whose worst-case running time is polynomial with respect to the number of fields within a classifier.

### B. Range Encoding

Range encoding schemes cope with range expansion by developing a new representation for important packets and intervals. For example, a new representation for interval $[1, 65534]$ may be developed so that this interval can be represented with one TCAM entry rather than 900 prefix entries. Previous range encoding schemes fall into two categories: database independent encoding schemes [4], [9], where each rule is encoded according to standard encoding scheme, and database dependent encoding schemes [5], [15], [18], [20], [26], where the encoding of each rule depends on the intervals present within the classifier. While range encoding methods do mitigate the effects of prefix expansion, they require either extra hardware or more per packet processing time.

### C. Circuit and Hardware Modification

Spitznagel *et al.* proposed adding comparators at each entry level to better accommodate range matching [22]. While this research direction is important, our contribution does not require circuit-level modifications to hardware. Zheng *et al.* developed load balancing algorithms for TCAM based systems to exploit chip level parallelism to increase classifier throughput with multiple TCAM chips without having to copy the complete classifier to every TCAM chip [28], [29]. This work may benefit from bit weaving since fewer rules would need to be distributed among the TCAM chips.

## III. Bit Swapping

In this section, we present a new technique called *bit swapping*. It is the first part of our bit weaving approach.

### A. Prefix Bit Swapping Algorithm

**Definition III.1** (Bit-swap)**.** *A* bit-swap $\beta$ *of a length* $m$ *ternary string* $t$ *is a permutation of the* $m$ *ternary bits; that is,* $\beta$ *rearranges the order of the ternary bits of* $t$*. The resulting permuted ternary string is denoted* $\beta(t)$*.* □

For example, if $\beta$ is permutation 312 and string $t$ is $0*1$, then $\beta(t) = 10*$. For any length $m$ string, there are $m!$ different bit-swaps. Bit-swap $\beta$ is a *prefix bit-swap* of $t$ if the permuted string $\beta(t)$ is in prefix format. Let $P(t)$ denote the set of prefix bit-swaps for $t$: specifically, the bit-swaps that move the $*$ bits of $t$ to the end of the string.

A bit-swap $\beta$ can be applied to a list $\ell$ of ternary strings $\langle t_1, \ldots, t_n \rangle$ where $\ell$ is typically a list of consecutive rules in a packet classifier. The resulting list of permuted strings is denoted as $\beta(\ell)$. Bit-swap $\beta$ is a prefix bit-swap for $\ell$ if $\beta$ is a prefix bit-swap for every string $t_i$ in list $\ell$ for $1 \leq i \leq n$. Let $P(\ell)$ denote the set of prefix bit-swaps for list $\ell$. It follows that $P(\ell) = \cap_{i=1}^{n} P(t_i)$.

Prefix bit-swaps are useful for two main reasons. First, we can minimize prefix rule lists using algorithms in [7], [16], [23]. Second, prefix format facilitates the second key idea of bit weaving, bit merging (Section IV). After bit merging, the classifier is reverted to its original bit order, which typically results in a non-prefix format classifier.

Unfortunately, many lists of string $\ell$ have no prefix bit-swaps which means that $P(\ell) = \emptyset$. For example, the list $\langle 0*, *0 \rangle$ does not have a prefix bit-swap. We now give the necessary and sufficient conditions for $P(\ell) \neq \emptyset$ after defining the following notation.

Given that each ternary string denotes a set of binary strings, we define two new operators for ternary strings: $\hat{0}(x)$ and $\sqsubseteq$. For any ternary string $x$, $\hat{0}(x)$ denotes the resulting ternary string where every 1 in $x$ is replaced by 0. For example, $\hat{0}$(1*)=0*. For any two ternary strings $x$ and $y$, $x \sqsubseteq y$ if and only if $\hat{0}(x) \subseteq \hat{0}(y)$. For example, 1*$\sqsubseteq$0* because $\hat{0}$(1*)=0*=$\{00, 01\} \subseteq \{00, 01\}=\hat{0}$(0*).

**Definition III.2** (Cross Pattern)**.** *Given two ternary strings* $t_1$ *and* $t_2$*, a cross pattern on* $t_1$ *and* $t_2$ *exists if and only if* $(t_1 \not\sqsubseteq t_2) \wedge (t_2 \not\sqsubseteq t_1)$*. In such cases, we say that* $t_1$ crosses $t_2$*.* □

We first observe that bit swaps have no effect on whether or not two strings cross each other.

**Observation III.1.** *Given two ternary strings,* $t_1$ *and* $t_2$*, and a bit-swap* $\beta$*,* $t_1 \subseteq t_2$ *if and only if* $\beta(t_1) \subseteq \beta(t_2)$*, and* $t_1 \sqsubseteq t_2$ *if and only if* $\beta(t_1) \sqsubseteq \beta(t_2)$*.* □

**Theorem III.1.** *Given a list* $\ell = \langle t_1, \ldots, t_n \rangle$ *of* $n$ *ternary strings,* $P(\ell) \neq \emptyset$ *if and only if no two ternary strings* $t_i$ *and* $t_j$ *(*$1 \leq i < j \leq n$*) cross each other.* □

*Proof: (implication)* It is given that there exists a prefix bit-swap $\beta \in P(\ell)$. Suppose that string $t_i$ crosses string $t_j$. According to Observation III.1, $\beta(t_i)$ crosses $\beta(t_j)$. This implies that one of the two ternary strings $\beta(t_i)$ and $\beta(t_j)$ has a $*$ before a 0 or 1 and is not in prefix format. Thus, $\beta$ is not in $P(\ell)$, which is a contradiction.

*(converse)* It is given that no two ternary strings cross each other. It follows that we can impose a total order on the ternary strings in $\ell$ using the relation $\sqsubseteq$. Note, there may be more than one total order if $t_i \sqsubseteq t_j$ and $t_j \sqsubseteq t_i$ for some values of $i$ and $j$. Let us reorder the ternary strings in $\ell$ according to this total order; that is, $t'_1 \sqsubseteq t'_2 \sqsubseteq \cdots \sqsubseteq t'_{n-1} \sqsubseteq t'_n$. Any bit swap that puts the $*$ bit positions of $t'_1$ last, preceded by the $*$ bit positions of $t'_2$, ..., preceded by the $*$ bit positions of $t'_n$, finally preceded by all the remaining bit positions will be a prefix bit-swap for $\ell$. Thus, the result follows. ■

Theorem III.1 gives us a simple algorithm for detecting whether a prefix bit-swap exists for a list of ternary strings. If a prefix bit-swap exists, the proof of Theorem III.1 gives us an *algorithm for constructing a prefix bit-swap* as shown in Algorithm 1. The algorithm sorts bit columns in an increasing order by the number of strings that have a $*$ in that column.

Before we formally present our bit swapping algorithm, we define the concepts of *bit matrix* and *decision array* for a possibly incomplete rule list (*i.e.*, there may exist a packet that none of the $n$ rules matches). Any list of $n$ rules defines a bit matrix $M[1..n, 1..b]$ and a decision array $D[1..n]$, where for any $1 \leq i \leq n$ and $1 \leq j \leq b$, $M[i, j]$ is the $j$-th bit in the predicate of the $i$-th rule and $D[i]$ is the decision of the $i$-th rule. Conversely, a bit matrix $M[1..n, 1..b]$ and a decision array $D[1..n]$ also uniquely defines a rule list. Given a bit matrix $M[1..n, 1..b]$ and a decision array $D[1..n]$ defined by a rule list, our bit swapping algorithm swaps the columns in $M$ such that for any two columns $i$ and $j$ in the resulting bit matrix $M'$ where $i < j$, the number of *s in the $i$-th column is less than or equal to the number of *s in the $j$-th column. Figure 2(a) shows a bit matrix and Figure 2(b) shows the resulting bit matrix after bit swapping. Let $L_1$ denote the rule list defined by $M$ and $D$, and let $L_2$ denote the rule list defined by $M'$ and $D$. Usually, $L_1$ will not be equivalent to $L_2$. This is not an issue. The key is that if we revert the bit-swap on any rule list $L_3$ that is equivalent to $L_2$, the resulting rule list $L_4$ will be equivalent to $L_1$.

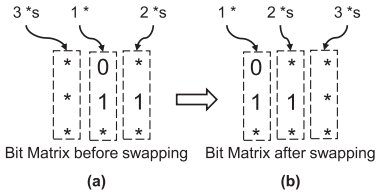


Fig. 2. Example of bit-swapping

### B. Minimal Cross-Free Classifier Partitioning Algorithm

Given a classifier $\mathbb{C}$, if $P(\mathbb{C}) = \emptyset$, we cut $\mathbb{C}$ into partitions where each partition has no cross patterns and thus has a prefix bit-swap. We treat classifier $\mathbb{C}$ as a list of ternary strings by ignoring the decision of each rule.

Given an $n$-rule classifier $\mathbb{C} = \langle r_1, \ldots, r_n \rangle$, a *partition* $\mathbb{P}$ on $\mathbb{C}$ is a list of consecutive rules $\langle r_i, \ldots, r_j \rangle$ in $\mathbb{C}$ for some $i$ and $j$ such that $1 \leq i \leq j \leq n$. A *partitioning*, $\mathbb{P}_1, \ldots, \mathbb{P}_k$, of $\mathbb{C}$ is a series of $k$ partitions on $\mathbb{C}$ such that the concatenation of $\mathbb{P}_1, \ldots, \mathbb{P}_k$ is $\mathbb{C}$. A partitioning is *cross-free* if and only if each

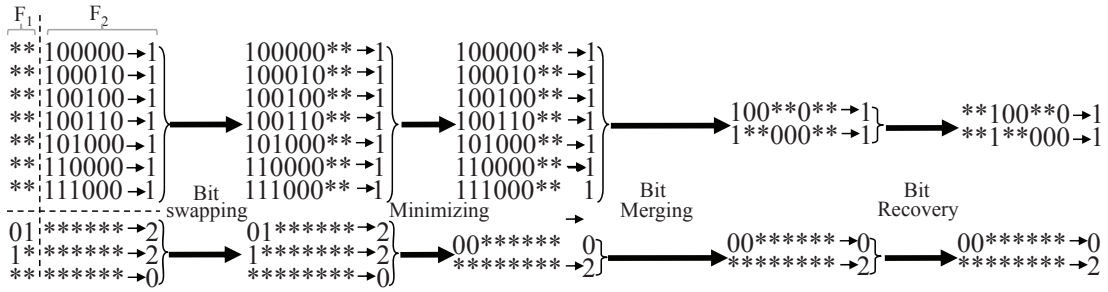


Fig. 3. Applying bit weaving algorithm to an example classifier

**Algorithm 1**: Finds a prefix bit-swap

**Input**: A classifier $C$ of $n$ rules $\langle r_1, \ldots, r_n \rangle$ where each rule has $b$ bits.
**Output**: A classifier $C'$ that is $C$ after a valid prefix bit-swap.

Let $M[1 \ldots n, 1 \ldots b]$ and $D[1 \ldots n]$ be the bit matrix and decision array of $C$;
Let $B = \langle (i, j) | 1 \leq i \leq b$ where $j$ is the number of *'s in $M[1 \ldots n, i] \rangle$;
Sort $B$ in ascending order of each pair's second value;
Let $M'$ be a copy of $M$;
**for** *k := 1 to b* **do**
  Let $(i, j) = B[k]$;
  $M'[1 \ldots n, k] := M[1 \ldots n, i]$;
Output $C'$ defined by $M'$ and $D$;

**Algorithm 2**: Find a minimal partition

**Input**: A list of $n$ rules $\langle r_1, \ldots, r_n \rangle$ where each rule has $b$ bits.
**Output**: A list of partitions.

Let $P$ be the current partition (empty list), and $L$ be a list of partitions (empty);
**for** $i := n$ *to* $1$ **do**
  **if** $r_i$ *introduces a cross pattern in* $P$ **then**
    Append $P$ to the head of $L$;
    $P := \langle r_i \rangle$;
  **else**
    Append $r_i$ to the head of $P$;
return $L$;

partition has no cross patterns. Given a classifier $\mathbb{C}$, a cross-free partitioning with $k$ partitions is *minimal* if and only if any partitioning of $\mathbb{C}$ with $k-1$ partitions is not cross-free. The goal of classifier partitioning is to find a minimal cross-free partitioning for a given classifier. We then apply independent prefix bit-swaps to each partition.

We give an algorithm, depicted in Algorithm 2, that finds a minimal cross-free partitioning for a given classifier. At any time, we have one active partition. The initial active partition is the last rule of the classifier. We consider each rule in the classifier in reverse order and attempt to add it to the active partition. If the current rule crosses any rule in the active partition, that partition is completed, and the active partition is reset to contain only the new rule. We process rules in reverse order to facilitate efficient incremental update (Section V-B). New rules are more likely to be added to the front of a classifier than at the end. It is not hard to prove that this algorithm produces a minimal cross-free partitioning for any given classifier.

The core operation in our cross-free partitioning algorithm is to check whether two ternary strings cross each other. We can efficiently perform this check based on Theorem III.2. For any ternary string $t$ of length $m$, we define the *bit mask* of $t$, denoted $M(t)$, to be a binary string of length $m$ where the $i$-th bit ($0 \leq i < m$) $M(t)[i] = 0$ if $t[i] = *$ and $M(t)[i] = 1$ otherwise. For any two binary strings $a$ and $b$, we use $a \,\&\&\, b$ to denote the resulting binary string of the bitwise logical AND of $a$ and $b$.

**Theorem III.2.** *For any two ternary string $t_1$ and $t_2$, $t_1$ does crosses $t_2$ if and only if $M(t_1) \,\&\&\, M(t_2)$ is different from both $M(t_1)$ and $M(t_2)$.* □

For example, given two ternary strings $t_1 = 01{*}0$ and $t_2 = 101{*}$, whose bit masks are $M(t_1) = 1101$ $M(t_1) = 1110$, we have $M(t_1) \,\&\&\, M(t_2) = 1100$. Therefore, $t_1 = 01{*}0$ crosses $t_2 = 101{*}$ because $M(t_1) \,\&\&\, M(t_2) \neq M(t_1)$ and $M(t_1) \,\&\&\, M(t_2) \neq M(t_2)$.

Figure 3 shows the execution of our bit weaving algorithm on an example classifier. Here we describe the bit swapping portion of that execution. The input classifier has 10 prefix rules with three decisions (0, 1, and 2) over two fields $F_1$ and $F_2$, where $F_1$ has two bits, and $F_2$ has six bits. We begin by constructing a maximal cross-free partitioning of the classifier by starting at the last rule and working upward. We find that the seventh rule introduces a cross pattern with the eighth rule according to Theorem III.2. This results in splitting the classifier into two partitions. Second, we perform bit swapping on each partition, which converts each partition into a list of one-dimensional prefix rules.

### C. Partial List Minimization Algorithm

We now describe how to minimize each bit-swapped partition where we view each partition as a list of 1-dimensional prefix rules. If a list of 1-dimensional prefix rules is complete (*i.e.*, any packet has a matching rule in the list), we can use the algorithms in [7], [23] to produce an equivalent minimal prefix rule list. However, the rule list in a partition is often incomplete; that is, there exist packets that do not match any rule in the partition.

Instead, we adapt the Weighted 1-Dimensional Prefix List Minimization Algorithm in [16]. Given a 1-dimensional packet classifier $\mathbb{C}$ of $n$ prefix rules $\langle r_1, r_2, \cdots, r_n \rangle$, where $\{Decision(r_1), Decision(r_2), \cdots, Decision(r_n)\} = \{d_1, d_2, \cdots, d_z\}$ and each decision $d_i$ is associated with a cost (*i.e.* weight) $Cost(d_i)$ (for $1 \leq i \leq z$), the cost of packet classifier $\mathbb{C}$ is defined as follows: $Cost(\mathbb{C}) =$

$\sum_{i=1}^{n} Cost(Decision(r_i))$. For any packet classifier $\mathbb{C}$, we use $\{\mathbb{C}\}$ to denote the set of all classifiers that are equivalent to $\mathbb{C}$. The problem of weighted one-dimensional TCAM minimization is defined as follows: given a one-dimensional packet classifier $\mathbb{C}_1$ where each decision is associated with a cost, find a prefix classifier $\mathbb{C}_2 \in \{\mathbb{C}_1\}$ such that for any prefix classifier $\mathbb{C} \in \{\mathbb{C}_1\}$, the condition $Cost(\mathbb{C}_2) \leq Cost(\mathbb{C})$ holds.

We adapt the Weighted 1-Dimensional Prefix List Minimization Algorithm in [16] to minimize a partial 1-dimensional prefix rule list $L$ over field $F$ as follows. Let $\{d_1, d_2, \cdots, d_z\}$ be the set of all the decisions of the rules in $L$. Let $\overline{L}$ denote the list of prefix rules that is the complement of $L$ (*i.e.*, any packet has one matching rule in either $L$ or $\overline{L}$, but not both) and each rule in $\overline{L}$ is assigned the same decision $d_{z+1}$ that is not in $\{d_1, d_2, \cdots, d_z\}$). First, we assign each decision in $\{d_1, d_2, \cdots, d_z\}$ a weight of 1 and the decision $d_{z+1}$ a weight of $|D(F)|$, the size of the domain $F$. Second, we concatenate $L$ with $\overline{L}$ to form a complete prefix classifier $L'$, and run the weighted 1-dimensional prefix list minimization algorithm in [16] on $L'$. Since this algorithm outputs a prefix classifier whose sum of the decision weights is the minimum, our weight assignment guarantees that decision $d_{z+1}$ only appears in the last rule in the minimized prefix classifier. Let $L''$ be the resulting minimized prefix classifier. Finally, we remove the last rule from $L''$. The resulting prefix list is the minimal prefix list that is equivalent to $L$.

Continuing the example from Figure 3, we use the partial prefix list minimization algorithm to minimize each partition to its minimal prefix representation. In this example, this step eliminates one rule from the bottom partition.

## IV. Bit Merging

In this section, we present bit merging, the second part of our bit weaving approach. The fundamental idea behind bit merging is to repeatedly find in a classifier two ternary strings that differ only in one bit and replace them with a single ternary string where the differing bit is $*$.

### A. Definitions

Two ternary strings $t_1$ and $t_2$ are *ternary adjacent* if they differ only in one bit, *i.e.*, their hamming distance [8] is one. The ternary string produced by replacing the one differing bit by a $*$ in $t_1$ (or $t_2$) is called the *ternary cover* of $t_1$ and $t_2$. For example, $0{**}$ is the ternary cover of $00{*}$ and $01{*}$. We call the process of replacing two ternary adjacent strings by their cover *bit merging* or just merging. For example, we can merge $00{*}$ and $01{*}$ to form their cover $0{**}$.

We now define how to bit merge (or just merge) two rules. For any rule $r$, we use $\mathbb{P}(r)$ to denote the predicate of $r$. Two rules $r_i$ and $r_j$ are ternary adjacent if their predicates $\mathbb{P}(r_i)$ and $\mathbb{P}(r_j)$ are ternary adjacent. The merger of ternary adjacent rules $r_i$ and $r_j$ is a rule whose predicate is the ternary cover of $\mathbb{P}(r_i)$ and $\mathbb{P}(r_j)$ and whose decision is the decision of rule $r_i$. We give a necessary and sufficient condition where bit merging two rules does not change the semantics of a classifier.

**Theorem IV.1.** *Two rules in a classifier can be merged into one rule without changing the classifier semantics if and only if they satisfy the following three conditions: (1) they can be moved to be positionally adjacent without changing the semantics of the classifier; (2) they are ternary adjacent; (3) they have the same decision.* □

The basic idea of bit merging is to repeatedly find two rules in the same bit-swapped partition that can be merged based on the three conditions in Theorem IV.1. We do not consider merging rules from different bit-swapped partitions because any two bits from the same column in the two bit-swapped rules may correspond to different columns in the original rules.

### B. Bit Merging Algorithm (BMA)

*1) Prefix Chunking:* To address the first condition in Theorem IV.1, we need to quickly determine what rules in a bit-swapped partition can be moved together without changing the semantics of the partition (or classifier). For any 1-dimensional minimum prefix classifier $\mathbb{C}$, let $\mathbb{C}^s$ denote the prefix classifier formed by sorting all the rules in $\mathbb{C}$ in decreasing order of prefix length. We prove that $\mathbb{C} \equiv \mathbb{C}^s$ if $\mathbb{C}$ is a 1-dimensional minimum prefix classifier in Theorem IV.2.

Before we introduce and prove Theorem IV.2, we first present Lemma IV.1. A rule $r$ is *upward redundant* if and only if there are no packets whose first matching rule is $r$ [12]. Clearly, upward redundant rules can be removed from a classifier with no change in semantics.

**Lemma IV.1.** *For any two rules $r_i$ and $r_j$ $(i < j)$ in a prefix classifier $\langle r_1, \cdots, r_n \rangle$ that has no upward redundant rules, $\mathbb{P}(r_i) \cap \mathbb{P}(r_j) \neq \emptyset$ if and only if $\mathbb{P}(r_i) \subset \mathbb{P}(r_j)$.* □

**Theorem IV.2.** *For any one-dimensional minimum prefix packet classifier $\mathbb{C}$, we have $\mathbb{C} \equiv \mathbb{C}^s$.* □

*Proof:* Consider any two rules $r_i, r_j$ $(i < j)$ in $\mathbb{C}$. If the prefixes of $r_i$ and $r_j$ do not overlap (*i.e.*, $\mathbb{P}(r_i) \cap \mathbb{P}(r_j) = \emptyset$), changing the relative order between $r_i$ and $r_j$ does not change the semantics of $\mathbb{C}$. If the prefixes of $r_i$ and $r_j$ do overlap (*i.e.*, $\mathbb{P}(r_i) \cap \mathbb{P}(r_j) \neq \emptyset$), then according to Lemma IV.1, we have $\mathbb{P}(r_i) \subset \mathbb{P}(r_j)$. This means that $\mathbb{P}(r_i)$ is strictly longer than $\mathbb{P}(r_j)$. This implies that $r_i$ is also listed before $r_j$ in $\mathbb{C}^s$. Thus, the result follows. ■

Based on Theorem IV.2, given a minimum sized prefix bit-swapped partition, we first sort the rules in decreasing order of their prefix length. Second, we further partition the rules into *prefix chunks* based on their prefix length. By Theorem IV.2, the order of the rules within each prefix chunk is irrelevant.

*2) Bit-Mask Grouping:* To address the second condition in Theorem IV.1, we need to quickly determine what rules are ternary adjacent. Based on Theorem IV.3, we can significantly reduce our search space by searching for mergeable rules only among the rules which have the same bit mask and decision.

**Theorem IV.3.** *Given a list of rules such that the rules have the same decision and no rule's predicate is a proper subset of another rule's predicate, if two rules are mergeable, then the bit masks of their predicates are the same.*

*Proof:* Suppose in such a list there are two rules $r_i$ and $r_j$ that are mergeable and have different bit masks. Because they are mergeable, $\mathbb{P}(r_i)$ and $\mathbb{P}(r_j)$ differ in only one bit. Because the bit masks are different, one predicate, say $\mathbb{P}(r_i)$,

must have a $*$ and the other predicate, $\mathbb{P}(r_j)$, must have a 0 or 1 in that bit column. Thus, $\mathbb{P}(r_j) \subset \mathbb{P}(r_i)$, which is a contradiction. ■

*3) Algorithm and Optimality:* The bit merging algorithm (BMA) works as follows. BMA takes as input a minimum, possibly incomplete prefix classifier $\mathbb{C}$ that corresponds to a cross-free partition generated by bit swapping. BMA first creates classifier $\mathbb{C}^s$ by sorting the rules of $\mathbb{C}$ in decreasing order of their prefix length and partitions $\mathbb{C}^s$ into prefix chunks. Second, for each prefix chunk, BMA groups all the rules with the same bit mask and decision together, eliminates duplicate rules, and searches within each group for mergeable rules. The second step repeats until no group contains rules that can be merged. Let $\mathbb{C}'$ denote the output of the algorithm.

Figure 4 demonstrates how BMA works. On the leftmost side is the first partition from Figure 3. On the first pass, eight ternary rules are generated from the original seven. For example, the top two rules produce the rule 1000*0** $\rightarrow$ 1. These eight rules are grouped into four groups with identical bit masks. On the second pass, two unique rules are produced by merging rules from the four groups. Since each rule is in a separate group, no further merges are possible and the algorithm finishes. Algorithm 3 shows the general algorithm for BMA.

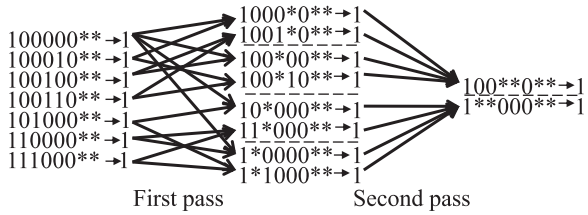

Fig. 4. Example of Bit Merging Algorithm Execution

**Algorithm 3**: Bit Merging Algorithm

**Input**: A list $I$ of $n$ rules $\langle r_1, \ldots, r_n \rangle$ where each rule has $b$ bits.
**Output**: A list of $m$ rules.

```
Let S be the set of rules in I;
Let C be the partition of S such that each partition contains a maximal set of rules in S where each rule has an identical bitmask and decision;
Let OS be an empty set;
for each c = {r'_1, ..., r'_m} ∈ C do
    for i := 1 to m − 1 do
        for j := i + 1 to m do
            if P(r'_i) and P(r'_j) are ternary adjacent then Add the ternary cover of P(r'_i) and P(r'_j) to OS;
Let O be OS sorted in decreasing order of prefix length;
if S = O then
    return O;
else
    return the result of BMA with O as input;
```

The correctness of this algorithm, $\mathbb{C}' \equiv \mathbb{C}$, is guaranteed because we only combine mergeable rules. We now prove that BMA is locally optimal as stated in Theorem IV.4.

**Lemma IV.2.** *During each execution of the second step, BMA never introduces two rules $r_i$ and $r_j$ such that $\mathbb{P}(r_i) \subset \mathbb{P}(r_j)$ where both $r_i$ and $r_j$ have the same decision.* □

**Lemma IV.3.** *Consider any prefix chunk in $\mathbb{C}^s$. Let $k$ be the length of the prefix of this prefix chunk. Consider any rule $r$ in $\mathbb{C}'$ that was formed from this prefix chunk. The $k$th bit of $r$ must be 0 or 1, not $*$.* □

**Theorem IV.4.** *The output of BMA, $\mathbb{C}'$, contains no pair of mergeable rules.*

*Proof:* Within each prefix chunk, after applying BMA, there are no pairs of mergeable rules for two reasons. First, by Theorem IV.3 and Lemma IV.2, in each run of the second step of the algorithm, all mergeable rules are merged. Second, repeatedly applying the second step of the algorithm guarantees that there are no mergeable rules in the end.

We now prove that any two rules from different prefix chunks cannot be merged. Let $r_i$ and $r_j$ be two rules from two different prefix chunks in $\mathbb{C}'$ with the same decision. Suppose $r_i$ is from the prefix chunk of length $k_i$ and $r_j$ is from the prefix chunk of length $k_j$ where $k_i > k_j$. By Lemma IV.3, the $k_i$-th bit of $r_i$'s predicate must be 0 or 1. Because $k_i > k_j$, the $k_i$-th bit of $r_j$'s predicate must be $*$. Thus, if $r_i$ and $r_j$ are mergeable, then $r_i$ and $r_j$ should only differ in the $k_i$-th bit of their predicates, which means $\mathbb{P}(r_i) \subset \mathbb{P}(r_j)$. This conflicts with Lemma IV.2. ■

Continuing the example in Figure 3, we perform bit merging on both partitions to reduce the first partition to two rules. Finally, we revert each partition back to its original bit order. After reverting each partition's bit order, we recover the complete classifier by appending the partitions together. In Figure 3, the final classifier has four rules.

## V. Discussion

### A. Redundancy Removal

Our bit weaving algorithm uses the redundancy removal procedure [12] as both the preprocessing and postprocessing step. We apply redundancy removal at the beginning because redundant rules may introduce more cross patterns. We apply redundancy removal at the end because our incomplete 1-dimensional prefix list minimization algorithm may introduce redundant rules across different partitions.

### B. Incremental Classifier Updates

Classifier rules periodically need to be updated when networking services change. When classifiers are updated manually by network administrators, timing is not a concern and rerunning the fast bit weaving algorithm will suffice. When classifiers are updated automatically in an incremental fashion, fast updates may be very important.

Bit weaving supports efficient incremental classifier updates by confining change impact to one cross-free partition. An incremental classifier change is typically inserting a new rule, deleting an existing rule, or modifying a rule. Given a change, we first locate the cross-free partition where the change occurs by consulting a precomputed list of all the rules in each partition. Then we rerun the bit weaving algorithm on the affected partition. We may need to further divide the partition into two cross-free partitions if the change introduces a cross pattern. Note that deleting a rule never introduces cross patterns. We generated our partitions by processing rules in

reverse order because new rules are most likely to be placed at the front of a classifier.

The experimental data used in Section VI indicates that only $2.7\%$ of partitions have more than 32 rules and $0.6\%$ of partitions have more than $128$ rules for real life classifiers. For synthetic classifiers, these percentages are $17.3\%$ and $0.9\%$, respectively. For these classifiers, incremental classifier updates are fast and efficient. To further evaluate the incremental update times, we divided each classifier into a top half and a bottom half. We constructed a classifier for the bottom half and then incrementally added each rule from the top half classifier. Using this test, we found that incrementally adding a single rule takes on average 2ms with a standard deviation of $4$ms for real world classifiers, and 6ms with a standard deviation of $5$ms for synthetically generated classifiers.

### C. Composability of Bit Weaving

Bit weaving, like redundancy removal, never returns a classifier that is larger than its input. Thus, bit weaving, like redundancy removal, can be composed with other classifier minimization schemes. Since bit weaving is an efficient algorithm, we can apply it as a postprocessing step with little performance penalty. As bit weaving uses techniques that are significantly different than other compression techniques, it can often provide additional compression.

We can also enhance other compression techniques by using bit weaving, in particular bit merging, within them. Specifically, multiple techniques [5], [16]–[18], [20] rely on generating single field TCAM tables. These approaches generate minimal prefix tables, but minimal prefix tables can be further compressed by applying bit merging. Therefore, every such technique can be enhanced with bit merging (or more generally bit weaving).

For example, TCAM Razor [16] compresses multiple field classifiers by converting a classifier into multiple single field classifiers, finding the minimal prefix classifiers for these classifiers, and then constructing a new prefix field classifier from the prefix lists. A natural enhancement is to use bit merging to convert the minimal prefix rule lists into smaller non-prefix rule lists. In our experiments, bit weaving enhanced TCAM Razor yields significantly better compression results than TCAM Razor alone.

Range encoding techniques [5], [15], [18], [20], [26] can also be enhanced by bit merging. Range encoding techniques require lookup tables to encode fields of incoming packets. When such tables are stored in TCAM, they are stored as single field classifiers. Bit merging offers a low cost method to further compress these lookup tables. Our results show that bit merging significantly compresses the lookup tables formed by the topological transformation technique [18].

### D. Complexity Analysis

The most computationally expensive stage of bit weaving is bit merging. With the application of the binomial theorem, we arrive at a worst case time complexity of $O(\mathrm{b}\times n^{\frac{5}{2}})$ where $b$ is the number of bits within a rule predicate, and $n$ is the number of rules in the input. Therefore, bit weaving is the first polynomial-time algorithm with a worst-case time complexity that is independent of the number of fields in that classifier. This complexity analysis excludes redundancy removal because redundancy removal is an optional pre/post-processing step. The space complexity of bit weaving is dominated by finding the minimum prefix list. For a complete complexity analysis of bit weaving, see [19].

## VI. Experimental Results

In this section, we evaluate the effectiveness and efficiency of bit weaving on both real-world and synthetic packet classifiers. First, we compare the relative effectiveness of Bit Weaving (BW) and the state-of-the-art classifier compression scheme, TCAM Razor (TR) [16]. Then, we evaluate how much additional compression results from enhancing prior compression techniques TCAM Razor and Topological Transformation (TT) [18] with bit weaving.

### A. Evaluation Metrics and Data Sets

We first define the following notation. We use $C$ to denote a classifier, $|C|$ to denote the number of rules in $C$, $S$ to denote a set of classifiers, $A$ to denote a classifier minimization algorithm, $A(C)$ to denote the classifier produced by applying algorithm $A$ on $C$, and $Direct(C)$ to denote the classifier produced by applying direct prefix expansion on $C$.

We define six basic metrics for assessing the performance of classifier minimization algorithm $A$ on a set of classifiers $S$ as shown in Table II. The *improvement ratio of $A'$ over $A$* assesses how much additional compression is achieved when adding $A'$ to $A$. $A$ does well when it achieves small compression and expansion ratios and large improvement ratios.

Compression Ratio

| Average | Total |
|---|---|
| $\frac{\Sigma_{C\in S}\frac{\lvert A(C)\rvert}{\lvert Direct(C)\rvert}}{\lvert S\rvert}$ | $\frac{\Sigma_{C\in S}\lvert A(C)\rvert}{\Sigma_{C\in S}\lvert Direct(C)\rvert}$ |

Expansion Ratio

| Average | Total |
|---|---|
| $\frac{\Sigma_{C\in S}\frac{\lvert A(C)\rvert}{\lvert C\rvert}}{\lvert S\rvert}$ | $\frac{\Sigma_{C\in S}\lvert A(C)\rvert}{\Sigma_{C\in S}\lvert C\rvert}$ |

Improvement Ratio

| Average | Total |
|---|---|
| $\frac{\Sigma_{C\in S}\frac{\lvert A(C)\rvert-\lvert A'(C)\rvert}{\lvert A(C)\rvert}}{\lvert S\rvert}$ | $\frac{\Sigma_{C\in S}\lvert A(C)\rvert-\lvert A'(C)\rvert}{\Sigma_{C\in S}\lvert A(C)\rvert}$ |

TABLE II
METRICS FOR $A$ ON A SET OF CLASSIFIERS $S$

We use $RL$ to denote a set of 25 real-world packet classifiers that we performed experiments on. The classifiers range in size from a handful of rules to thousands of rules. We obtained $RL$ from several network service providers where many classifiers from the same provider are structurally similar varying only in the IP prefixes of some rules. When we run any compression algorithms on these structurally similar classifiers, we get essentially identical results. We eliminated the resulting double counting of results that would bias the resulting averages by randomly choosing a single classifier from each set of structurally similar classifiers to be in $RL$. We then split $RL$ into two groups, $RLa$ and $RLb$, where the expansion ratio of direct expansion is less then 2 in $RLa$ and the expansion ratio of direct expansion is greater than $40$ in $RLb$. We have no classifiers where the expansion ratio of direct expansion is between 2 and $40$. It turns out $|RLa| = 12$ and $|RLb| = 13$. By separating these classifiers into two groups, we can determine how well our techniques work on classifiers

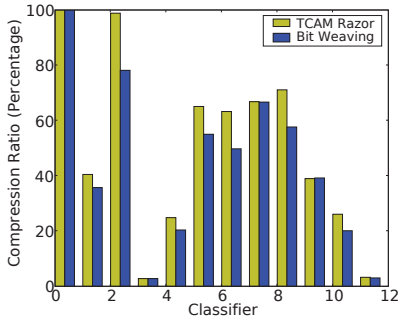


Fig. 5. Compression ratio for $RLa$

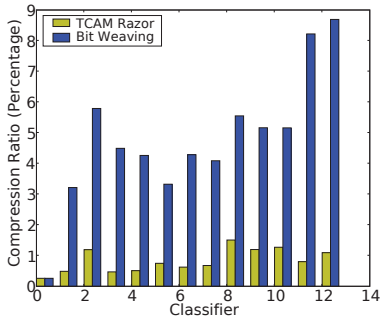


Fig. 6. Compression ratio for $RLb$

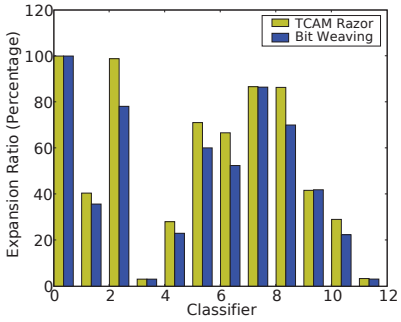


Fig. 7. Expansion ratio for $RLa$

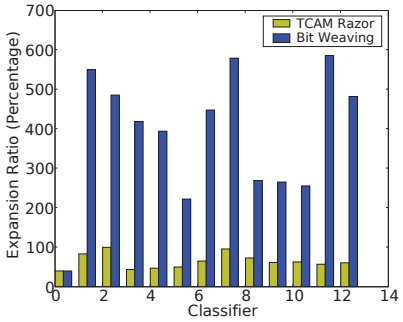


Fig. 8. Expansion ratio for $RLb$

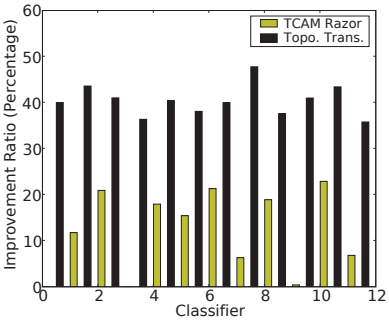


Fig. 9. Improvement for $RLa$

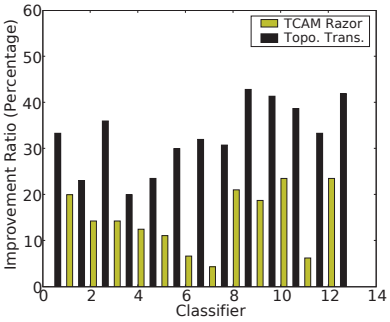


Fig. 10. Improvement for $RLb$

that do suffer significantly from range expansion as well as those that do not.

Due to security concerns, it is difficult to acquire a large quantity of real-world classifiers. We generated a set of 150 synthetic classifiers $SYN$ with the number of rules ranging from 250 to 8000. The predicate of each rule has five fields: source IP, destination IP, source port, destination port, and protocol. We based our generation method upon Singh *et al.*'s [21] model of synthetic rules. We also performed experiments on $TRS$, a set of 490 classifiers produced by Taylor&Turner's Classbench [25]. These classifiers were generated using the parameters files downloaded from Taylor's web site (http://www.arl.wustl.edu/~det3/ClassBench/index.htm). To represent a wide range of classifiers, we chose a uniform sampling of the allowed values for the parameters of smoothness, address scope, and application scope.

To stress test the sensitivity of our algorithms to the number of classifier decisions, we created a set of classifiers $RL_U$ (and thus $RLa_U$ and $RLb_U$) by replacing the decision of every rule in each classifier by a unique decision. Similarly, we created the set $SYN_U$. Thus, each classifier in $RL_U$ (or $SYN_U$) has the maximum possible number of distinct decisions.

### B. Effectiveness of Bit Weaving Alone

Table III shows the average and total compression ratios, and the average and total expansion ratios for TCAM Razor and Bit Weaving on all nine data sets. Figures 5 and 6 show the specific compression ratios for all of our real-world classifiers, and Figures 7 and 8 show the specific expansion ratios for all of our real-world classifiers. Clearly, bit weaving is an effective algorithm with an average compression ratio of 23.6% on our real-world classifiers and 34.6% when these classifiers have unique decisions. This is very similar to TCAM Razor, the previous best known-compression method.

One interesting observation is that *TCAM Razor and bit weaving seem to be complementary techniques.* That is, TCAM Razor and bit weaving seem to find and exploit different compression opportunities. Bit weaving is more effective on $RL_a$ while TCAM Razor is more effective on $RL_b$. TCAM Razor is more effective on classifiers that suffer from range expansion because it has more options to mitigate range expansion including introducing new rules to eliminate bad ranges. On the other hand, by exploiting non-prefix optimizations, bit weaving's ability to find rules that can be merged is more effective than TCAM Razor on classifiers that do not experience significant range expansion.

### C. Improvement Effectiveness of Bit Weaving

Table III shows the improvement to average and total compression and expansion ratios when TCAM Razor and Topological Transformation are enhanced with bit weaving on all nine data sets. Figures 9 and 10 show how bit weaving improved compression for each of our real-world classifiers.

Our results for enhancing TCAM Razor with bit weaving is actually the best result from three different possible compositions: bit weaving alone, TCAM Razor followed by bit weaving, and a TCAM Razor algorithm that uses bit merging to generate non-prefix classifiers. Topological Transformation is enhanced by performing bit merging on each of its encoding tables. We do not perform bit weaving on the encoded classifier because the nature of Topological Transformation produces encoded classifiers that do not benefit from non-prefix encoding. Therefore, for Topological Transformation, we report only the improvement to storing the encoding tables.

*Bit weaving significantly improves both TCAM Razor and Topological Transformation with an improvement ratio of* 12.8% *and* 38.9%*, respectively.* TCAM Razor and bit weaving

| | Compression Ratio | | | | Expansion Ratio | | | | Improvement Ratio | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Average | | Total | | Average | | Total | | Average | | Total | |
| | TR | BW | TR | BW | TR | BW | TR | BW | TR | TT | TR | TT |
| $RL$ | 24.5 % | 23.6 % | 8.8 % | 10.7 % | 59.8 % | 222.9 % | 30.1 % | 36.8 % | 12.8 % | 36.5 % | 12.8 % | 38.9 % |
| $RLa$ | 50.1 % | 44.0 % | 26.7 % | 23.7 % | 54.6 % | 48.0 % | 29.0 % | 25.7 % | 11.9 % | 40.4 % | 12.7 % | 39.9 % |
| $RLb$ | 0.8 % | 4.8 % | 0.8 % | 5.0 % | 64.7 % | 384.3 % | 65.1 % | 397.7 % | 13.6 % | 32.8 % | 14.3 % | 34.7 % |
| $RL_U$ | 31.9 % | 34.6 % | 13.1 % | 17.1 % | 146.2 % | 465.5 % | 45.0 % | 58.8 % | 3.5 % | 35.6 % | 2.8 % | 38.2 % |
| $RLa_U$ | 62.9 % | 61.6 % | 36.0 % | 35.0 % | 68.7 % | 67.2 % | 39.1 % | 38.0 % | 2.0 % | 40.6 % | 2.9 % | 39.3 % |
| $RLb_U$ | 3.3 % | 9.6 % | 3.0 % | 9.2 % | 217.7 % | 833.1 % | 237.6 % | 732.6 % | 4.8 % | 30.9 % | 2.1 % | 32.7 % |
| $SYN$ | 10.4 % | 9.7 % | 7.8 % | 7.3 % | 12.3 % | 11.5 % | 9.3 % | 8.7 % | 8.1 % | 42.0 % | 9.4 % | 43.2 % |
| $SYN_U$ | 42.7 % | 41.2 % | 38.4 % | 36.2 % | 50.8 % | 49.0 % | 45.8 % | 43.2 % | 3.9 % | 43.3 % | 5.7 % | 44.1 % |
| $TRS$ | 13.8 % | 7.8 % | 20.6 % | 10.3 % | 41.7 % | 21.3 % | 45.2 % | 22.7 % | 32.9 % | 34.0 % | 49.7 % | 33.6 % |

TABLE III
EXPERIMENTAL RESULTS FOR ALL DATA SETS

exploit different compression opportunities so they compose well. Bit weaving significantly helps Topological Transformation because each encoding table of Topological Transformation is essentially the projection of the classifier along the given field. Thus each encoding table contains every relevant range in that field. This leads to non-adjacent intervals with the same decision that can benefit from bit merging.

### D. Efficiency

We implemented all algorithms on Microsoft .Net framework 2.0. Our experiments were carried out on a desktop PC running Windows XP with 8G memory and a single 2.81 GHz AMD Athlon 64 X2 5400+. All algorithms used a single processor core. On $RL$, the minimum, mean, median, and maximum running times of our bit weaving algorithm (excluding the time of running the redundancy removal algorithm before and after running our bit weaving algorithm) were $0.0002$, $0.0339$, $0.0218$, and $0.1554$ seconds, respectively; on $RL_U$, the minimum, mean, median, and maximum running times of our bit weaving algorithm were $0.0003$, $0.2151$, $0.0419$, and $1.7842$ seconds, respectively. On synthetic rules, the running time of bit weaving grows linearly with the number of rules in a classifier, where the average running time for classifiers of 8000 rules is $0.2003$ seconds.

## VII. CONCLUSION

*Bit weaving* is the first TCAM compression method that can create non-prefix field classifiers and runs in polynomial time regardless of the number of fields in each rule. It also supports fast incremental updates to classifiers, and it can be deployed on existing classification hardware. Given its speed and its ability to find different compression opportunities than existing compression schemes, bit weaving should always be used, either by itself or as a postprocessing routine, whenever TCAM classifier compression is needed.

## REFERENCES


[1] Integrated Device Technology, Inc. Content addressable memory. http://www.idt.com/.

[2] B. Agrawal and T. Sherwood. Modeling tcam power for next generation network devices. In *Proc. IEEE Int. Symposium on Performance Analysis of Systems and Software*, pages 120– 129, 2006.

[3] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang. Compressing rectilinear pictures and minimizing access control lists. In *Proc. (SODA)*, January 2007.

[4] A. Bremler-Barr and D. Hendler. Space-efficient TCAM-based classification using gray coding. In *Proc. (Infocom)*, May 2007.

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

[6] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *Proc. ACM Sigmetrics*, pages 311–322, 2006.

[7] R. Draves, C. King, S. Venkatachary, and B. Zill. Constructing optimal IP routing tables. In *Proc. IEEE INFOCOM*, pages 88–97, 1999.

[8] R. W. Hamming. Error detecting and correcting codes. *Bell Systems Technical Journal*, 29:147–160, April 1950.

[9] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *Proc. ACM SIGCOMM*, pages 193 – 204, August 2005.

[10] C. Lambiri. Senior staff architect IDT, private communication. 2008.

[11] P. C. Lekkas. *Network Processors - Architectures, Protocols, and Platforms*. McGraw-Hill, 2003.

[12] A. X. Liu and M. G. Gouda. Complete redundancy detection in firewalls. In *Proc. 19th Annual IFIP Conf. on Data and Applications Security, LNCS 3654*, pages 196–209, August 2005.

[13] A. X. Liu and M. G. Gouda. Complete redundancy removal for packet classifiers in tcams. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, to appear.

[14] A. X. Liu, C. R. Meiners, and Y. Zhou. All-match based complete redundancy removal for packet classifiers in TCAMs. In *Proc. IEEE Infocom*, April 2008.

[15] H. Liu. Efficient mapping of range classifier into Ternary-CAM. In *Proc. Hot Interconnects*, pages 95– 100, 2002.

[16] C. R. Meiners, A. X. Liu, and E. Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. In *Proc. IEEE ICNP*, pages 266–275, October 2007.

[17] C. R. Meiners, A. X. Liu, and E. Torng. Algorithmic approaches to redesigning tcam-based systems [extended abstract]. In *Proc. ACM SIGMETRICS*, June 2008.

[18] C. R. Meiners, A. X. Liu, and E. Torng. Topological transformation approaches to optimizing tcam-based packet processing systems. In *Proc. ACM SIGCOMM (poster session)*, August 2008.

[19] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in tcams. Technical Report MSU-CSE-09-1, Department of Computer Science and Engineering, Michigan State University, January 2009.

[20] D. Pao, P. Zhou, B. Liu, and X. Zhang. Enhanced prefix inclusion coding filter-encoding algorithm for packet classification with ternary content addressable memory. *Computers & Digital Techniques, IET*, 1:572–580, April 2007.

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

[22] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended TCAMs. In *Proc. IEEE ICNP*, pages 120– 131, November 2003.

[23] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35:287–300, 2003.

[24] D. E. Taylor. Survey & taxonomy of packet classification techniques. *ACM Computing Surveys*, 37(3):238–275, 2005.

[25] D. E. Taylor and J. S. Turner. Classbench: A packet classification benchmark. In *Proc. IEEE Infocom*, March 2005.

[26] J. van Lunteren and T. Engbersen. Fast and scalable packet classification. *IEEE Journals on Selected Areas in Communications*, 21(4):560– 571, 2003.

[27] F. Yu, T. V. Lakshman, M. A. Motoyama, and R. H. Katz. SSA: A power and memory efficient scheme to multi-match packet classification. In *Proc. ANCS*, pages 105–113, October 2005.

[28] K. Zheng, H. Che, Z. Wang, B. Liu, and X. Zhang. DPPC-RE: TCAM-based distributed parallel packet classification with range encoding. *IEEE Transactions on Computers*, 55(8):947–961, 2006.

[29] K. Zheng, C. Hu, H. Lu, and B. Liu. A TCAM-based distributed parallel ip lookup scheme and performance analysis. *IEEE/ACM Transactions on Networking*, 14(4):863–875, 2006.