GenMatcher: A Generic Clustering-Based Arbitrary Matching Framework

PING WANG, LUKE MCHALE, PAUL V. GRATZ, and ALEX SPRINTSON,

Department of Electrical and Computer Engineering, Texas A&M University

Packet classification methods rely upon packet content/header matching against rules. Thus, throughput of matching operations is critical in many networking applications. Further, with the advent of Software Defined Networking (SDN), efficient implementation of software approaches to matching are critical for the overall system performance.

This article presents¹ GenMatcher, a generic, software-only, arbitrary matching framework for fast, efficient searches. The key idea of our approach is to represent arbitrary rules with efficient prefix-based tries. To support arbitrary wildcards, we rearrange bits within the rules such that wildcards accumulate to one side of the bitstring. Since many non-contiguous wildcards often remain, we use multiple prefix-based tries. The main challenge in this context is to generate efficient trie groupings and expansions to support all arbitrary rules. Finding an optimal mix of grouping and expansion is an NP-complete problem.

Our contribution includes a novel, clustering-based grouping algorithm to group rules based upon their bit-level similarities. Our algorithm generates near-optimal trie groupings with low configuration times and provides significantly higher match throughput compared to prior techniques. Experiments with synthetic traffic show that our method can achieve a 58.9X speedup compared to the baseline on a single core processor under a given memory constraint.

CCS Concepts: • **Networks** → **Network architectures**; **Network algorithms**; **Packet classification**; **Network simulations**; **Network performance analysis**; *Packet-switching networks*;

Additional Key Words and Phrases: Arbitrary matching, correlation clustering, performance tradeoff

ACM Reference format:

Ping Wang, Luke Mchale, Paul V. Gratz, and Alex Sprintson. 2018. GenMatcher: A Generic Clustering-Based Arbitrary Matching Framework. *ACM Trans. Archit. Code Optim.* 15, 4, Article 51 (November 2018), 22 pages. <https://doi.org/10.1145/3281663>

1 INTRODUCTION

Packet classification is an enabling function for a variety of applications within networking, including Quality of Service (QoS), security, monitoring, and multimedia communications. The emerging distributed computing and big data applications require the imposition of strict requirements for

¹New article; not an extension of a conference paper.

Authors' addresses: P. Wang, L. McHale, P. V. Gratz, and A. Sprintson, Department of Electrical and Computer Engineering, Texas A&M University, 400 Bizzell St, College Station, TX, 77840; emails: freesummer@tamu.edu, luke.mchale@tamu.edu, pggratz@gratz1.com, spalex@tamu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 1544-3566/2018/11-ART51

<https://doi.org/10.1145/3281663>

data processing delays and throughput. On the other hand, the emergence of Software Defined Networking (SDN) and a push toward more general purpose networking hardware is driving the need for optimized software approaches to high throughput matching. Accordingly, in this article, we present a generic and efficient pure-software mechanism for arbitrary matching. Our algorithm can be used in a broad range of packet classification application, including SDN packet processing pipeline.

In general networking applications, packet classification typically includes bit-wise matching of a *key* against a predefined rule set [13, 15–17, 42]. The key is typically a subset of packet header fields, but might also include other meta-data derived from the packet header.

The matching functions used in packet classification typically come in four forms:

- **Exact match:** An exact match rule corresponds to exactly one matching key (i.e., no wildcard bits).
- **Range match:** A range match rule defines a sequential set of possible matching keys. This form of matching is occasionally used in defining a rule that covers multiple matching ports (e.g., from 1–30 inclusive).
- **Arbitrary match:** An arbitrary match rule contains wildcards (i.e., a wildcard bit can be either 0 or 1) at any bit position, matching 2^n possible keys for each wildcarded bit.
- **Prefix match:** In prefix matches, all the wildcards must uniformly cover the low-order bits of the rule.

While there exists a large body of research on packet matching, most of this work focuses on prefix and range matching. With the developing trends toward new SDN services, big data, and High Performance Computing (HPC), matching is no longer performed with five standard header fields. New classification applications examine additional fields, increasing rule flexibility. Further, parallel computing techniques for machine learning and big data analytics are blooming and are able to process increasing volumes of data simultaneously. Thus, more generic packet matching methods are required to deal with all the various matching requirements.²

Prefix matching has been most heavily studied in prior work [40]. An important improvement for software-based prefix matching is the trie data structure [40]. A trie is a binary-tree data structure, which significantly accelerates matching by bounding the search complexity to the tree depth instead of total rule count.

While tries can greatly accelerate matching for situations where prefix matches are defined, ultimately, this approach is insufficient to meet all matching needs. In particular, tries cannot be directly used in non-prefix, arbitrary match scenarios where wildcards may appear at any bit-position. Unfortunately, as we will describe, in most current applications, while prefix matches may be defined for particular fields of a rule, when multiple fields of type other than exact match are defined in a rule, the problem becomes one of arbitrary matching. Accordingly, this article proposes a software framework for arbitrary matching, which considers the performance tradeoff between search time and memory cost. The goal of this article is to achieve the highest search throughput, under a given memory constraint, for generic arbitrary matching.

We propose *GenMatcher*, a generic clustering-based arbitrary matching framework and a software-based arbitrary matching approach. The basic idea of *GenMatcher* is to transform the rule set into a minimal (under a given memory bound), optimized set of prefix format rule groups. All rules are carefully organized into a minimal number of groups, such that each group constructs a trie—transforming the group’s subset of arbitrary match rules into prefix match rules.

²Range matching can be trivially transformed into a small number of prefix matches, thus we do not discuss it further.

Creating a minimal set of prefix rule groups from the initial arbitrary rules requires a careful transformation. *GenMatcher* uses a correlation clustering-based grouping algorithm. To enable this clustering grouping algorithm, *GenMatcher* finds the relationships among all rules and allocates them into separate groups according to their similarities. For a given number of rules N , the complexity of *GenMatcher* grouping algorithm is $O(N^2)$. This is a large improvement versus the complexity of a brute force search for the optimal grouping, which we show has a complexity of $O(N^N)$.

After *GenMatcher* finds the best grouping of rules, we employ a bit swapping algorithm based on prior work [32] to rearrange the bit order, such that the largest possible prefix match is constructed. After the bit swapping operation, if there are any rules with wildcard bits outside of the prefix, we expand these rules to ensure all rules become prefix rules. Because of this rule expansion, memory may rapidly exceed the available memory in the system. Thus, one objective of *GenMatcher* is to improve search time under a given memory bound. Finally, *GenMatcher* inserts these transformed prefix rules into trie data structures, one trie per group. Then, for each incoming key, we traverse all the tries to find a match.

Challenges: *GenMatcher* employs its grouping algorithm to allocate all the rules into minimal groups under a given memory constraint threshold. In general, clustering [2, 3, 5, 9, 28, 35, 44, 45] is a powerful tool for finding the underlying structure of large datasets [19, 30]. Our goal is to apply the clustering result to build a data structure for efficient arbitrary matching. To implement the efficient arbitrary matching, we must address several challenging technical problems:

- We need to determine a similarity function to correlate and group rules.
- We need to develop a grouping algorithm to maximumly group similar rules together based on the similarity function.
- We must guarantee that the grouping algorithm will produce a data structure that can satisfy performance requirements while remaining within a given memory constraint.

Contributions: *GenMatcher* is a generic arbitrary matching approach that can deal with arbitrary rules without requiring human specified hints or other configuration metadata. Our *GenMatcher* has the following features:

- We introduce *GenMatcher*, a generic arbitrary matching framework, which can process any form of matching used in packet classification. This is the first framework we are aware of for optimizing general, bit-wise arbitrary matching under a given memory bounds.
- We develop a novel similarity function for use in correlation clustering-based grouping. To the best of our knowledge, this similarity function is the first such function to be used in arbitrary matching. The similarity function is a bit-wise based, efficient means to extract the relationship between rules. Any two rules in a rule table have a different similarity value based on the overlapping wildcard distributions. The similarity value is used to allocate all the rules into minimal groups under a memory constraint in order to balance search time with memory cost.
- Unlike prior arbitrary matching techniques, *GenMatcher* does not require specialized hardware. It is a pure software approach that can be applied to any general matching problem.

2 BACKGROUND

Much prior work exists in both software and hardware-based approaches to improve lookup performance for tables containing network IP addresses [39, 46]. IP lookup is typically defined as a composite prefix matching function, as illustrated in the left table in Figure 1. In this table, there are five rules ($R1-R5$) that define the circumstances under which packets must undergo one of

Rules	Field 1	Field 2	Priority	Action
R1	10**	11**	1	A
R2	1***	101*	2	B
R3	00**	10**	3	A
R4	10**	1***	4	C
R5	100*	110*	5	B

Rules	Field 1	Priority	Action
R1	10**11**	1	A
R2	1***101*	2	B
R3	00**10**	3	A
R4	10**1***	4	C
R5	100*110*	5	B

Fig. 1. Arbitrary packet matching.

three actions (A, B, or C). Here, rules are defined using two fields (*Field1* and *Field2*) where each bit is exact (0s or 1s in the table) or wildcarded (* in the table). If a given key is covered by rule's *Field1* and *Field2*, then the key is considered matching and returns the defined action. Here, as is typical in networking applications, each field is a prefix match, thus all wildcards are constrained to the least-significant bit positions of each field.

2.1 Motivation for Arbitrary Matching

Arbitrary matching is the most general form of matching, covering all the other types of matching patterns, including the exact, range, and prefix match. Further, while traditional IP packet classification is typically defined in terms of prefix matches on an individual field of the packet header, when multiple sets of such prefix defined match fields are defined together in each rule, the matching function becomes arbitrary. In Figure 1, each rule's *Fields* have different prefix masks defined (i.e., there are different numbers of **s in each rule for each *Field*). While each field could be processed individually as a prefix match with a trie as described above, this would require a separate trie traversal for each field, with the match not determined until the results from all trie traversals are completed and combined. This approach restricts opportunities to optimize the data structures across the entire rule.

Alternately, matching might be accelerated by combining all *Fields* into a single match function, as shown on the right table of Figure 1. This approach means that the entire set of extracted header *Fields* can be compared against the rules in one operation, which does lend itself to optimizations as the entire rule is considered as whole. Doing so, however, changes the matching function to arbitrary match as shown in Figure 1. Thus, tries may not be used directly, motivating the need for new approaches to accelerate arbitrary matching.

We further note that arbitrary matches have applications beyond traditional network classification. With the emergence of big data, more and more applications need to leverage bit-wise matching. In HPC systems, the Message Passing Interface (MPI) standard defines a set of rules, known as tag matching [4, 20]. Tag matching is designed for matching source-send operations to destination-receives. Instead of matching the packet header of the switch network, there are many MPI message fields that must match a sender and its corresponding receive functions: (1) the communicator;³ (2) the user tag, including a wildcard specified by the receiver; (3) the source rank,⁴ including a wildcard specified by the receiver; and (4) the destination rank. Thus, the tag matching mechanism is a form of arbitrary matching.

2.2 Relationship with Prior Art

While prefix matching is heavily studied, arbitrary matching, particularly in software, remains a highly under-examined area. Among the few works in this area, Meiners et. al proposed Bitweaving [32], a non-prefix approach to compressing packet classifiers. We note that their approach was designed to compress rules for use with a ternary content-addressable memory

³In HPC system, the communicator is the process group ID in multi-core system.

⁴The rank is the stack address of a message that shows where the message comes from.

Fig. 2. GenMatcher framework.

(TCAM) [25]. Reference [32] proposed a bit swapping algorithm to transform non-prefix rules into prefix rules. This was combined with a bit merging operation to reduce the number of rules. As this mechanism is intended to reduce the number of rules that need to be stored in a TCAM [24], the algorithm makes no further optimization to the rule set.

In this article, in order to avoid the hardware and energy cost of TCAMs, and to provide a general software-based solution, we propose a generic, software, arbitrary matching mechanism while keeping the door open to hardware acceleration.

3 GENMATCHER

In this section, we present an overview of our design. GenMatcher is a generic clustering-based arbitrary matching framework for software packet processing with the following design goals:

- **Performance:** It processes packets at high speed under a limited memory cost.
- **Generality:** It supports any type of bit-wise matching with rules of arbitrary length and wildcards.

To achieve these goals, GenMatcher leverages prior work wherein efficient trie data structures are generated from sets of prefix rules. To this end, GenMatcher consists of three phases: map, group, and build trie, as shown in Figure 2. The goal of these phases is to generate subsets of rules (groups), setup as prefix matches, from which tries can be generated through a combination of bit swapping (rearranging the bits in the rules) and rule expansion (wherein wildcards are enumerated). Our framework aims to keep the number of groups and amount of expansion to a minimum to keep performance high while minimizing memory cost. As the full space of all possible groupings with expansion is untenable for any reasonable number of rules, our framework is a heuristic that aims to achieve optimal grouping and expansion without requiring a full search. These phases are detailed in this section.

3.1 Map Phase

In map phase, first, we parse rules into four fields, represented by the rule.value, rule.mask, rule.priority, and rule.action fields.

Definition 3.1 (Rule expression). Each rule has four properties: value, mask, priority, and action, as shown in Figure 2. These properties have the following definitions:

- **Rule.value:** Represents the non-wildcarded component of the desired match with the wildcard portions set to 0. Generated by parsing all *'s (wildcards) in the rule and replacing them with 0s, keeping all other bits unchanged.
- **Rule.mask:** The bit-mask used to clear the wild-card components of the key for comparison against the Rule.value. Generated by parsing all *'s and replacing them with 1's, clearing all other bits.

- **Rule.priority:** Where there are multiple rules that match a given key, we choose the rule with the highest priority (1 being the highest). This is provided as input or inferred by the order of rules initially given.
- **Rule.action:** Action to take place as the result of a match, enumerated. E.g., forward the packet to port A, B, or C. Provided as input.

Priority is an important property in matching applications. In packet classification, priority is used to preserve the order semantic of the rules in a rule table, which determines the order of insertion into the trie data structure. The smaller the order value of the rule in the rule table, the higher the priority of the rule. Thus, the rules with higher priority will be inserted earlier, which eliminates the insertion of redundant information. In MPI tag matching, the order semantic of the messages must be preserved. Here, order is represented by the priority property. If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message if the first one is still pending [8].

After being processed, each rule is now expressed as a set: {value, mask, priority, action}. Given this set, we define the matching process following the operations described in Definition 3.2.

Definition 3.2. (Match function).

$$f(\text{key}, \text{mask}, \text{value}) = (\text{AND}(!\text{mask}, \text{key}) = \text{value}).$$

$$\begin{cases} \text{match} = \text{true}, & \text{if } \text{AND}(!\text{mask}, \text{key}) = \text{value}; \\ \text{match} = \text{false}, & \text{else.} \end{cases}$$

Example 3.3. Assume a rule $R1$ is $1^{**}0^{**}11$, then its Mask is 01101100, and its value is 10000011. Also assume a key 10001011.

$10001011 \& 01101100 = 10001011 \& 10010011 = 10000011$, which equals Rule.value.

$\therefore$ Key 10001011 matches with $R1$.

Instead, assume the incoming key is 01101011.

$01101011 \& 01101100 = 01101011 \& 10010011 = 00000011$, which does not equal Rule.value.

$\therefore$ Key 01101011 does not match with $R1$.

If multiple rules match, we always choose the one with the highest priority. The match result consists of the priority of the matched rule and its corresponding action. Upon match, the matched rule's action attribute will be returned to the application.

3.2 Group Phase

In the *group* phase, we exploit the correlation between rules, observing the similarity of their wildcard patterns, with the goal of transforming all rules into prefixes. Here, we have the goal of minimizing the number of groups (since more groups require more search time) while minimizing the amount of rule expansion necessary to construct a trie from each group. Thus, before continuing, we formally define rule expansion:

Definition 3.4 (Expand operation). Expand the non prefix wildcard bits into 1's and 0's.

Example 3.5. Assume rule $R1$ is $10^{*}011$. Here, $R1$ is a non-prefix rule.

After the expansion operation, the rule will now be expanded to two new rules without wildcards: 100011 and 101011.

Instead, assume rule $R2$ is $1^{**}0^{**}$, where the rightmost ** s form a prefix in common with other rules in the group.

Fig. 3. Grouping examples.

After expansion, this will result in four rules: 1000^{**} , 1010^{**} , 1100^{**} , and 1110^{**} .

Thus, the resulting number of expanded rules is determined by the number of non-prefixed * bits. We define the number of * in the middle of a rule as E . Thus, the number of expanded rules is 2^E .

Different grouping algorithms generate different resultant groupings. For the rule table shown in Figure 2, there are many groupings possible; Figure 3 illustrates two possible groupings. The first group result is the single group $G1$. The second group result is the two groups $G1'$ and $G2'$. We note, as we will show, building a trie from the single group, $G1$ will require some expansion because a single prefix match is not possible for this set of rules, while building tries for each group $G1'$ and $G2'$ will not require expansion.

Since the goal of the group phase is to minimize the number of groups while minimizing the amount of rule expansion necessary to construct a trie for each group, we first attempt to put all rules into one group. If this does not cause the the memory cost to go beyond the given threshold, the group result is the entire rule table; otherwise, we employ our proposed GenMatcher grouping algorithm to split the rules into multiple groups. In the GenMatcher grouping policy, we assign a rule to a group based on the similarity between the rule and the core of the group. Even if the rule's similarity is qualified, we need to check the memory cost status of the group if it accepts the new rule. If the memory cost is still within the threshold after the group has absorbed this rule, the rule will be assigned to this group; otherwise, the rule will be assigned to a new group. Thereafter, the number of the group is increased.

3.3 Build Phase

In the trie build phase, we employ the bit swapping algorithm [32] to rearrange the bits within the rules in each group such that the wildcards accumulate to the right-most position of the bit string. After swap operations, if some of the rules in each group are still not in a prefix format, we need to expand the non-prefix rules into prefix rules.

In Figure 4(a), there is only one group $G1$ generated in the group phase. After the swap operation, three rules, 1^*10 , 1^*01 , and 1^*1^{**} , remain that are not in prefix format. Thus, we must expand the three rules into six rules in order to allow a single trie to be built for this group. As shown in Figure 4(b), we must construct two different swap operations to produce prefix matches for each group $G1'$ and $G2'$. After the swap operations, all the rules in each group are in a prefix format. Thus, this second set of groups requires no expansion. As we can observe from Figures 3 and 4, the groupings chosen in the group phase determine the number of tries and the number of expanded rules.

3.4 Objectives and Challenges

As discussed previously, to utilize the efficiency of a trie data structure in performing search operations, each rule must be a prefix match. Thus, rules should be split into groups where each

Fig. 4. Build trie examples.

group can be bit-swapped to make the largest possible prefix matches. Since each trie must be traversed sequentially, a larger number of groups will result in a longer time to finish searching than a smaller number.⁵ Thus, our strategy is to employ rule expansion to decrease the number of groups, which also tends to reduce the search time. A potential problem with this approach, however, is that the expansion increases the number of trie nodes in the trie data structure, which increases memory utilization. Ultimately, if memory utilization exceeds the system memory limit, the search performance will dramatically suffer.

Objective: Since we must transform arbitrary rules into prefix rules, we may have multiple groups and many rule expansions. Our objective is to minimize the search time while not exceeding the memory threshold, defined thus:

- **Search time:** determined by the number of groups and the trie data structure for each group.
- **Memory cost:** determined by the number of trie nodes, which is largely increased through rule expansion.

Challenges: Different grouping results lead to different trie data structures, represented by the number of groups and the number of trie nodes. Thus, we need to resolve the tradeoff between the number of groups and the number of trie nodes to obtain an optimal performance balance. In this article, we propose the GenMatcher grouping algorithm to form optimized groupings for prefix match searching to achieve high search throughput with a fixed, given memory cost threshold.

4 THE GENMATCHER GROUPING ALGORITHM

In this section, we present the correlation clustering-based (GenMatcher) grouping algorithm in detail.

4.1 Similarity Function

Recalling the goal of our application, we want to minimize the search time while not exceeding the memory threshold. In order to minimize the search time, the grouping algorithm needs to generate as few groups as possible, while keeping expansion to a minimum, thus reducing the required memory. We therefore must maximumly group similar rules together with similarity defined as having the most wildcard positions in common.

Since the number of groups and the number of trie nodes are mainly determined by the wildcard distribution in each rule, we can observe the similarity from the *rule.mask* field. Considering two rules R_1 and R_2 , we define the similarity function $s(R_1, R_2)$ as follows.

⁵The size/depth of each trie itself also impacts search time. Since the complexity of trie search, however, grows as $O(h)$ where $h \sim \log n$ typically, the number of tries tends to have a greater effect on search time than the size of the trie.

Definition 4.1. The similarity between rules R_1 and R_2 is collected from their wildcard distribution, as defined in Equation (1).

$$s(R_1, R_2) = \begin{cases} M(R_1.mask \& R_2.mask), & \text{if } R_1.mask \neq 0 \wedge R_2.mask \neq 0 \\ 1, & \text{if } R_1.mask = 0 \parallel R_2.mask = 0, \end{cases} \quad (1)$$

where $M(\cdot)$ counts the number of 1's set in a bitstring.

Since expansion is determined by the wildcard distribution, the intersection of similarities among rules with non-zero mask is the key factor in grouping. If $R_1.mask \neq 0 \wedge R_2.mask \neq 0$, the similarity equals to the number of * that appear on both R_1 and R_2 ; If $R_1.mask = 0 \parallel R_2.mask = 0$, which means that we always can group these two rules together since there is no expansion. Because rules that have mask = 0 (i.e., no wildcards) cause no rule expansion and can be placed equally in any group, so we set the similarity to 1⁶ when $R_1.mask = 0 \parallel R_2.mask = 0$. If the rule is 64-bit, the similarity belongs to $[0, 64]$, where 0 denotes no similarity between two rules.

4.2 GenMatcher Grouping Algorithm

We now present our correlation clustering-based GenMatcher grouping algorithm for the group phase in GenMatcher. Given a set $\mathcal{D}$ of rules, our GenMatcher algorithm generates a set $\mathcal{O}$ of groups. It requires a parameter θ , the given memory threshold.

The pseudocode of GenMatcher grouping is shown in Algorithm 1. The algorithm consists of two steps. In the first step (lines 2–10), the *calculateMatrix* function computes the similarity of any two rules in the rule set $\mathcal{D} = \{R_1, R_2, \dots, R_N\}$ utilizing Definition 4.1. In the second step (lines 12–27), the *assignCluster* function assigns rules into groups according to the similarity matrix $S_{N \times N}$, calculated by the *calculateMatrix* function.

In the second step, the rule that has the most correlation with other rules in the rule set is chosen as the core of a new group. The procedure is shown in line 13. The *index* is the index of the core of the new group. If the similarity between a rule and the core is positive and under the given memory limit, the rule is added to the vector $H[1]$; otherwise, it is added to the vector $H[2]$. $H[1]$ is a new group of the output $\mathcal{O}$. $H[2]$ is a vector storing all the rules not qualified in $H[1]$. $H[2]$ will be processed in function *assignCluster* until empty. After we obtain the grouping result $\mathcal{O}$, we can build the trie data structure, as shown in Figure 2. Thereafter, we process the binary search operations for all the incoming keys.

We now present, in Lemma 4.2, the time complexity of the GenMatcher algorithm. In Lemma 4.3, we examine the determinant factors of search performance for our GenMatcher algorithm.

LEMMA 4.2. *The time complexity of our GenMatcher grouping algorithm is $O(N^2)$, where N is the number of rules.*

PROOF. For the function *calculateMatrix* (lines 2–10), the time complexity is $O(N^2)$. For the function *assignCluster* (lines 12–27), the time complexity is $O(N * num)$, where *num* is the number of groups. Thus, the time complexity of the GenMatcher algorithm is dominated by the function *calculateMatrix*, which is $O(N^2)$. $\square$

LEMMA 4.3. *For our GenMatcher grouping algorithm, the search performance is primarily determined by the number of groups; the fewer groups, the better the search performance. When the number of groups is fixed, fewer trie nodes always perform better.*

⁶In our GenMatcher grouping policy, if the similarity between a rule and the core of a group is 1, the rule will be assigned to a group if it does not break the memory limitation. We arbitrarily choose the value 1 because it is small enough that it does not affect the selection of the core of a group, which we want to have the greatest summation of similarities between itself and any other rules.

ALGORITHM 1: GenMatcher Grouping Algorithm.

Require: $\mathcal{D} = \{R_1, R_2, \dots, R_N\}$
Ensure: $\mathcal{O} = \{G_1, G_2, \dots, G_{num}\}$

```

1: Set  $\theta = 64\text{MB}$ 
2: function CALCULATEMATRIX( $\mathcal{D}$ )
3:    $S_{N \times N} \leftarrow \emptyset, L_{N \times 1} \leftarrow \emptyset$ 
4:   for  $i=1; i \leq N; i++$  do
5:     for  $j=1; j \leq N; j++$  do
6:        $S_{ij} \leftarrow s(R_i, R_j)$ 
7:        $L_i \leftarrow L_i + S_{ij}$ 
8:     end for
9:   end for
10: end function
11:
12: function ASSIGNCLUSTER( $\mathcal{D}, S_{N \times N}, L_{N \times 1}$ )
13:    $index \leftarrow \underset{i \in \mathcal{D}}{\operatorname{argmax}}(L_{N \times 1})$ 
14:   for  $i=1; i \leq \mathcal{D}.size(); i++$  do
15:     if  $S_{index, i} > 0 \wedge \text{total memory cost} < \theta$  then
16:        $H[1] \leftarrow R_i$ 
17:     else
18:        $H[2] \leftarrow R_i$ 
19:     end if
20:   end for
21:    $\mathcal{O} \leftarrow H[1]$  /* a new group is generated */
22:   if  $H[2].size() == 0$  then
23:     return  $\mathcal{O}$ 
24:   else
25:      $assignCluster(H[2], S_{N \times N}, L_{N \times 1})$ 
26:   end if
27: end function

```

PROOF. Let us compare two different groupings constructed using an identical set of rules. Assuming that each trie generated by the two groupings are balanced, the performance will not be impacted by bias keys. The first grouping results in N trie nodes contained within a single group. The second grouping results in M trie nodes across two groups; one group has M_1 trie nodes, the other one has M_2 trie nodes. The average search attempts⁷ for the first grouping is $\log_2(N)$. Since we assume tries must be searched sequentially to get the matched result, the average search attempts for the second grouping is $\log_2(M_1) + \log_2(M_2)$, where $M_1 + M_2 = M$. Thus, the search performance of a single group performs better than two groups when

$$\begin{aligned} \log_2(N) &< \log_2(M_1) + \log_2(M_2) = \log_2(M_1 M_2) \\ &\Rightarrow N < M_1 M_2 \end{aligned}$$

If $M_1 = M_2$, $\log_2(N) < 2 \log_2(\frac{M}{2}) \Rightarrow N < \frac{M^2}{4}$. If the number of groups is G , then the statement is true when $N < M_1 M_2 \cdots M_G$. As G increases, $M_1 M_2 \cdots M_G$ is more than likely larger than N ,

⁷The number of search attempts for a given key is the depth that a key needs to traverse in a trie data structure before a match can be determined.

especially when the G groups have an even number of trie nodes, that is, $N < \frac{M^G}{G^G}$. In general, $M \gg G$. Therefore, as G increases, M^G will always grow faster than G^G ; thus, we can say that the search performance is primarily determined by the number of groups.

Let us now consider two groupings with an identical group size, generated by the same rule set. If the keys are unbiased, the grouping result with the least number of trie nodes has fewer search attempts. As the search time is determined by the number of search attempts, the search time is proportional to the number of trie nodes. If the keys are biased, a grouping with more trie nodes might perform better when the trie data structure is unbalanced. As needed, we can employ tree balance techniques [6, 11, 18, 23, 41, 43] to rebalance the data structure. Thus, when the number of groupings is fixed, fewer trie nodes always perform better. $\square$

5 EVALUATION

In this section, we evaluate *GenMatcher* on rule sets generated from packet capture (PCAP) traces. We first present the evaluation methodology, followed by the correctness and scalability of the *GenMatcher* grouping algorithm. Finally, we compare the results obtained versus previous techniques.

5.1 Methodology

We program *GenMatcher* framework in C++, leveraging our in-house developed binary trie data structure and the linear vector data structure in C++ standard library (STL). A rule generation heuristic developed by McHale et al. [31] was used to synthesize a set of rules relevant to a given PCAP trace. Rules used in this project consisted of IPv4 source and destination addresses, resulting in a key width of 64-bits. Keys used to measure matching performance were constructed using the source and destination IPv4 address of every packet extracted from the given PCAP trace. The PCAP dataset (equinix-sanjose.dirA.20120119-125903) examined in this article was taken from an Internet backbone link and provided by CAIDA [1]. CAIDA's anonymization preserves relative flows across packets in a trace. These traces provide a more realistic dataset with less entropy compared to randomly generated rules and keys.

For the rule tables, we use our generator to generate 11,000 rules based on the CAIDA PCAP data [1]. We randomly pick 10 different sets of rule samples with six different rule table sizes (256, 512, 1,024, 2,048, 4,096, 8,192). We extract 28,744,877 keys from the traces as previously described. We set a memory cost threshold of 64 MB. We evaluate our *GenMatcher* on a real system with 512 GB DRAM and Intel Xeon E5-2697A V4 32-core 2.6GHz processor. L1d cache is 32k, L1i cache is 32k, L2 cache is 256k, and L3 cache is 40,960k. All tests are executed with one single thread.

We evaluate the performance in terms of search time, configuration time, and memory cost. First, in order to show the correctness and scalability of the *GenMatcher* grouping algorithm, we compare *GenMatcher* against a brute force group search algorithm. For the brute force algorithm, the objective is to find the optimal group results, which result in the minimal number of groups with minimal number of trie nodes while not exceeding the memory threshold. The time complexity of the brute force algorithm is $O(N^N)$, where N is the number of rules.

Second, we compare *GenMatcher* against two arbitrary matching algorithms: the Linear arbitrary matching algorithm, which serves as a baseline, and a software-only version of the Bitweaving [32] algorithm. Linear arbitrary matching employs C++ STL vectors to build a rule table and search the rules sequentially. The goal of Bitweaving is the compression and reformatting of rules such that they could be best implemented in a TCAM. Their proposed technique is composed of a bit swapping algorithm and a bit merging algorithm. In the bit swapping stage, they employ a grouping policy. First, they sort the rules by the wildcard positions and the number

Table 1. The Number of Trie Nodes for BF and GenMatcher

<i>Rule#</i>	10	11	12	13	14	15	16	17	18	19	20
BF	338	435	452	464	535	587	666	708	671	724	734
Gen	338	435	452	464	535	587	666	708	671	724	734

Table 2. The Number of Groups for BF and GenMatcher

<i>Rule#</i>	10	11	12	13	14	15	16	17	18	19	20
BF	2	2	2	2	2	2	2	2	2	2	2
Gen	2	2	2	2	2	2	2	2	2	2	2

of wildcards in each rule. After sorting, the rules with similar wildcard distribution are moved next to each other. Second, they start grouping rules from the beginning of the sorted rule list. They define a cross-free condition to determine if the adjacent rules can be grouped together. If the rules' wildcard distribution has intersection and all the wildcards can be swapped to the right side of the rules by applying a same permutation pattern, the rules can be grouped together; otherwise, they will make another group. The grouping process will not stop until the last rule in the rule list has been examined. Within each group, a single permutation is applied to each rule's bits to produce a reordered rule. Bitweaving employs its grouping policy to transform all the non-prefix rules into prefix rules. After grouping, all the rules in each group are in a prefix format.

5.2 Comparison with Brute Force Grouping

In Lemma 4.3, we prove that the search performance is determined by the number of groups and trie nodes. Thus, if the **GenMatcher** algorithm generates the same groups and number of trie nodes compared with the brute force (**BF**) algorithm, the **GenMatcher** algorithm is achieving the optimal grouping. Since the **BF** algorithm is an NP-complete problem, we can only evaluate it on small rule samples due to the runtime limitation. We randomly choose 10 different sets of rule samples ranging from 10 rules to 20 rules from the overall rule set.

Tables 1 and 2 show the number of trie nodes and the number of groups generated by the **GenMatcher** algorithm and the brute force algorithm on selected rule samples.⁸ As shown in Tables 1 and 2, we observe that **GenMatcher** and **BF** generate the same number of trie nodes (**Gen** represents **GenMatcher**). In each case, the same number of groups were generated as well. Since the search performance is determined by the number of trie nodes, the search performance of the two algorithms are the same.

With respect to the time complexity, the **BF** is much worse than the **GenMatcher** algorithm. The average process time for grouping 10 to 20 rules is shown in Figure 5. The process time for **GenMatcher** is ineligible at only $1\mu s$ since the size of the rule samples is very small. However, the process time for **BF** increases exponentially with the number of rules. The process time of generating the group results for **BF** is nearly 13 minutes when the rule sample size = 20.

Overall, the performance of our proposed **GenMatcher** algorithm for this range of rule counts matches optimal. Further, the time complexity of **GenMatcher** is much better than the **BF** algorithm.

⁸Since the rules were chosen randomly, sometimes a larger set of rules produces a slightly larger prefix (and smaller trie), than a smaller set of rules. Thus, 18 rules has less nodes than 17.

Fig. 5. Time complexity comparison.

Fig. 6. **GenMatcher** grouping algorithm scalability.

5.3 Scalability

To show the scalability of **GenMatcher**, we evaluate its configuration time by scaling up the number of rules from 128 to 8,192. Figure 6 shows the configuration time (i.e., the time to generate all tries) of **GenMatcher** when scaling the number of rules. The configuration time consists of two parts. One is the time for processing **GenMatcher** to get the group results. The other one is to build the trie data structure based on the group results. Thus, the configuration time is a summation of the process time and the build time, where the build time is the summation of rule rearrange time and rule insertion time, which are happening during the build trie phase. The rule rearrange time is the time cost during the bit swapping operation. The rule insertion time is the time cost for inserting all the rules into the trie data structure. As expected, configuration time increases with the number of rules. However, for a 8,192 rules, the configuration time of **GenMatcher** is less than one second. Thus, **GenMatcher** incurs little overhead for a large set of rules.

Since **Bitweaving** and **GenMatcher** both have group phase and build trie phase, their configuration time elements are the same. Figure 7 shows the configuration time comparisons in terms of build time and process time with respect to **Bitweaving** and **GenMatcher**. **Bit** represents **Bitweaving**, and **Gen** represents **GenMatcher**. Here, we see that **Bit_build** and **Gen_build** both take less time than **Bit_process** and **Gen_process**. On the 8,192nd case, we observe that **Gen_build** is larger than **Bit_build**. This is because **GenMatcher** generates less groups than **Bitweaving** by trading off memory for performance. The insertion time of **GenMatcher** is larger than **Bitweaving** due to the rule expansion. For the process time comparison, since the time complexity of **Bitweaving**'s grouping is $O(N)$, which is smaller than $O(N^2)$, **Bit_process** is smaller than **Gen_process**. Although the configuration time of **GenMatcher** is larger than **Bitweaving**,

Fig. 7. Configuration time comparisons between Bitweaving and GenMatcher.

Table 3. Grouping Result on *Rule num* = 4,096

S#	1	2	3	4	5	6	7	8	9	10
Gen	2	1	1	1	1	2	2	2	2	2
Bit	62	1	1	1	1	62	62	62	60	62

the resulting search performance is much better than **Bitweaving**, which is the goal of our *GenMatcher* framework. The search performance is described in Section 5.4.

5.4 Performance Comparisons

We compare **GenMatcher** against **Linear** and **Bitweaving** in terms of search time and memory cost.

5.4.1 Search Time. Figure 8 shows the search time speedup with respect to **Linear**, **Bitweaving** and **GenMatcher**. For all the six different rule sizes (256, 512, 1,024, 2,048, 4,096, 8,192), **GenMatcher** achieves the best performance. In the figure, for each rule size, performance of 10 different scenarios (randomly chosen rules) are shown.

In the figure, we see that **GenMatcher** generally provides greater speedups for larger numbers of rules. This is because the performance of **Linear** degrades (linearly) with increasing numbers of rules, while the performance of both **GenMatcher** and **Bitweaving** tend to be dominated by the number of groups each technique creates.

Between **Bitweaving** and **GenMatcher**, since the search time is determined by the number of groups and the number of trie nodes within the given memory threshold, the performance is always better with fewer groups. Because **Bitweaving** generates more groups than **GenMatcher**, **GenMatcher** always outperforms **Bitweaving**.

The *Rule num* = 4,096th case, shown in Figure 8(e), shows an interesting behavior. In the figure, we see that both **Bitweaving** and **GenMatcher** produce substantially better results for scenarios 2–5 than for the others. Table 3 shows the number of groups generated for each scenario by the two techniques. Here, we see that **GenMatcher** and **Bitweaving** both generate only one group for scenarios 2–5. Despite having the same number of groups, **GenMatcher** still produces better performance. This is because **GenMatcher** generates a smaller number of trie nodes than **Bitweaving**, which is shown in Table 4. The different number of trie nodes is obtained by the different number of inserted rules in the trie data structure, as shown in Table 5. In our **GenMatcher**, we insert rules by its priority order. In **Bitweaving**, it sorts the rules by an ascending order of the number of wildcards in a rule [32], which breaks the rules' priority order. Thus, **Bitweaving** does not insert rules by its priority order, which results in redundant trie nodes in trie data structure.

Fig. 8. Search time speedup comparison, normalized against **Linear**.Table 4. The Number of Trie Nodes Result
on *Rule num* = 4,096

S#	2	3	4	5
Gen	43,163	42,284	42,806	42,839
Bit	48,522	47,414	47,610	48,065

Table 5. The Number of Inserted Rules
Result on *Rule num* = 4,096

S#	2	3	4	5
Gen	1,913	1,923	1,908	1,926
Bit	3,125	3,197	3,156	3,102

Fig. 9. Memory cost versus the number of groups at different rule sample sizes.

Since **Bitweaving** typically generates a large number of groups to maintain the given memory threshold, its performance is even worse than **Linear** when the number of rules is small. However, the performance of **Bitweaving** improves with increasing rules. When the number of rules is up to 4,096, as shown in Figure 8(e), **Bitweaving** surpasses **Linear**.

Out of the six different rule counts, **GenMatcher** achieves the best performance on *Rule num* = 8,192. On average, **GenMatcher** achieves an 58.9X speedup compared to the baseline. To summarize, the best search performance can be achieved by minimizing the number of groups while remaining under the memory cost threshold.

5.4.2 Memory Cost. Figure 9 shows the memory cost with respect to 10 different scenario numbers on various numbers of rules. Out of the six different rule counts, **Linear** has the least memory cost, which is determined by the number of rules. The memory cost of **Linear** is the product of the number of rules and the size of a rule. The size of a rule consists of two 64-bit words for the *mask* field and *value* field. Thus, the size of a rule is 16B. Both **Bitweaving** and **GenMatcher** are based on trie data structures, thus the cost is determined by the number of trie nodes. Their memory cost is the product of the number of the trie nodes and the node size. The trie node consists of two pointers (each pointer is 64-bit) and one integer (32-bit). The node size is $8 + 8 + 4 = 20B$.

Table 6. Grouping Result on *Rule num* = 8,192

S#	1	2	3	4	5	6	7	8	9	10
Gen	2	2	1	2	2	2	2	2	2	2
Bit	5	9	3	9	7	5	7	5	9	7

Table 7. Expansion Result on *Rule num* = 8,192

S#	1	2	3	4	5	6	7	8	9	10
Gen	51,360	55,088,256	536,887,296	55,088,256	51,360	0	12,880	51,360	55,088,256	51,360
Bit	0	0	0	0	0	0	0	0	0	0

Fig. 10. Performance comparisons at different memory threshold.

Between **Bitweaving** and **GenMatcher**, without rule expansion, the memory cost is consistent, depending on the number of groups. The fewer number of groups, the smaller the memory cost. However, the memory cost is more diverse when rule expansion occurs. In Figure 9(a)–(d), **Bitweaving** always consumes more memory than **GenMatcher**. This is because the number of groups generated by **Bitweaving** is larger than that generated by **GenMatcher**, and there is no rule expansion for both of them.

We take the *Rule num* = 4,096 and *Rule num* = 8,192 cases as examples. Table 3 shows the grouping results for *Rule num* = 4,096. The performance is consistent with the number of groups. Note that on scenarios 2, 3, 4, and 5, the cost of **Bitweaving** and **GenMatcher** are lowest because the number of the groups are the smallest and there is no rule expansion.

Tables 6 and 7 show the grouping result and expansion result for *Rule num* = 8,192, respectively. Note that, since there is a large number of expanded rules on scenarios 2, 3, 4 and 9, though **GenMatcher** generates less groups than **Bitweaving**, **GenMatcher** requires much more memory. As shown in Figure 9(f), on scenario 2, 3, 4, and 9, **GenMatcher** has a greater memory cost than **Bitweaving**. To achieve the best search performance, we trade off memory cost. This meets our objective: minimize the search time while not exceeding the memory threshold.

Figure 10 shows the performance of GenMatcher at different memory thresholds (8MB, 16MB, 32MB, 64MB). For all 10 scenarios examined, GenMatcher only produces a significantly different memory cost for the third test. In this test, GenMatcher chooses to create fewer groups when more memory is available through the more aggressive use of rule expansion. These fewer groups lead to a significant performance improvement as shown in Figure 10(b), where thresholds of 32MB and 64MB perform significantly better. We note that GenMatcher is highly conservative when configured to maintain a given memory threshold, thus the actual memory used is often much lower than the threshold. This is because GenMatcher uses a heuristic at grouping phase to estimate the upper bound of memory, which might be used during the expansion phase. This

Fig. 11. Memory cost per rule.

heuristic often leaves some memory on the table but is much faster (at configuration time) than iteratively grouping and expanding to hit a given memory target.

Figure 11 shows the memory cost per rule with respect to Bitweaving and GenMatcher for all the six different rule sizes (256, 512, 1,024, 2,048, 4,096, 8,192). For all cases except 8,192, GenMatcher has a lower memory cost per rule than Bitweaving. In the 8,192nd case, to preserve matching performance by reducing groups, GenMatcher expands more rules than Bitweaving (while remaining under the defined memory budget), thus, GenMatcher uses more memory per rule than Bitweaving. However, GenMatcher maintains a better search time in return for the extra memory per rule.

6 RELATED WORK

Prior work on arbitrary matching mainly falls into two categories: TCAM-based solutions and algorithmic solutions. The TCAMs [47] represent a hardware-based approach to matching. A TCAM is a specialized memory array with integrated comparison logic, where each entry stores a rule that encoded in a ternary format (i.e., 0, 1, and *). Thus, arbitrary rules can be stored in a TCAM directly. The TCAM can compare keys against all stored rules in parallel, and return the match result with the highest priority. TCAMs take $O(1)$ time to finish the search and generate a match result; however, TCAM cells require much higher area and power than traditional static random access memory (SRAM) arrays. As a result, TCAMs are expensive and often have limited capacity. Further, TCAM parameters, such as rule length, and number of entries are determined at the hardware design time, often making them a poor fit for the SDN paradigm of network application definition/configuration at runtime.

TCAMs are generally unavailable in general purpose complexity. Significant research has been done on TCAM compression techniques [32, 33] to reduce the number of entries stored in TCAM. One such method defined by Meiners et. al is Bitweaving [32]. The primary aim of Meiners et. al in Bitweaving [32] was the compression and reformatting of rules such that they could be efficiently implemented in a TCAM. Their proposed technique is composed of a bit swapping algorithm and a rule merging algorithm. For bit swapping, first, they sort the rules by the number of wildcards in each rule. Second, they separate rules from the rule table into groups. Within each group, a single permutation is applied to each rule's bits to produce a reordered rule. After grouping, all the rules become prefix rules. They also propose a bit merging operation. In each group, if the hamming distance between any two adjacent rules is 1, then the rules will be merged into one rule with bit * on that position. For the hamming distance to be 1, there can only be one bit position difference between rules. After finishing the bit merge operation, the new merged rules will be processed by TCAM.

Bitweaving's intent was to produce rules that could be used in a TCAM; however, there is much to be learned from this work for the application of software-only arbitrary matching. In order to

process any type of rules in our GenMatcher, we adapt Bitweaving's bit swapping algorithm to transform rules into a prefix format to the maximum extent. In the Bitweaving mechanism, all the rules are able to be transformed to a prefix format by applying a large swapping algorithm. We note that, critically, Bitweaving makes no attempt at expanding rules to minimize the number of groups. Thus, rules are broken into groups wherever a prefix match is not possible. Unlike Bitweaving, the objective of GenMatcher is to minimize the number of groups within a memory threshold. Our technique expands the non-prefix wildcards to reduce the number of groups under a given memory threshold, which improves the search performance at some memory cost.

Although the approach here is intended for generic matching, there is much to be learned from prior work, software solutions based packet classification. He et al. [10] proposed the *SmartSplit* algorithm. They split a large rule set in several subsets and use different packet classification algorithms for different subsets. Although this approach decreases memory consumption as well as increases classification speed, it still suffers from rule duplication and cut decision configuration cost. Inoue et al. [12] proposed multidimensional-cutting via selective bit-concatenation to accelerate many-field packet classification. However, the method they propose to generate lookup tables is complicated and there is no guarantee for a correct classification. Kogan et al. [14] split the rules into two parts: independent order rules and order-dependent rules. The independent rules are divided into multiple groups, and the dependent rules are processed in the TCAM. Although they avoid impacting space and time complexities, it only focuses on range matching and requires a false positive test.

Other approaches [36, 37] perform packet classification on multi-core processors. Qi et al. propose a technique that is able to support very large rule sets [37]. The approach, however, suffers from memory expansion and can only support up to 1K 5-field rules. Qu et al. [36] proposed an efficient bit-string aggregation technique to avoid excessive memory usage on multi-core processors. Lu and Sahini [27] propose a collection of hash tables to represent a multi-dimensional packet classification table. This work focuses on range matching. For a large size of rules and a many-field packet, the complexity of building the trie will become large. Several other software-based packet classification algorithms have been proposed, which only deal with some specific rules or sub-rules but not the general form of the matching problem [10, 12, 14, 27, 36, 37]. By contrast, GenMatcher is a generic arbitrary matching mechanism for any form of matching.

Several groups considered representations based on rule disjointness [16, 17, 34] and addressed efficient time-space tradeoffs for multi-field classification, where fields are represented by ranges. In these works, they assign all rules into multiple disjoint groups, where every group obeys a structural property on a subset of bit indices of a rule. Unlike their work, GenMatcher considers the rule as a whole, which results in a smaller time complexity for grouping.

7 DISCUSSION

7.1 Intelligent Matching

One important characteristic of big data is the variety of data types and data sources. For the packet classification applications, the keys from different applications can be described by many different fields with different data types. Since packet classification performance is determined by both the classification algorithm and ruleset, we could achieve better performance if we are able to choose the classification algorithm according to a given ruleset. This raises an interesting research direction, performing intelligent grouping by exploiting and learning the statistical characteristics of the keys. The grouping results indicate that it is possible to build a more efficient data structure when also considering some statistical bias in the keys.

Elmahgiubi et al. [7] propose a novel framework for efficient, automatic packet classification algorithm selection. They utilized Meta-learning and Artificial Neural Networks (ANNs) to predict

the most appropriate packet classification algorithm and achieve an average accuracy of 90% classification. Wang [48] employed stacked autoencoders (SAE) to implement feature learning, protocol classification, anomalous protocol detection, and unknown protocol identification. Lotfollahi et al. [26] propose a deep learning based framework, called Deep Packet, which can handle both application identification and traffic characterization tasks. They used two deep learning methods: stacked autoencoders (SAE) neural network (NN) and convolutional NN. Deep Packet can identify encrypted traffic and also distinguishes between virtual private network (VPN) and non-VPN network traffic. While these works represent interesting research directions, they solve very different problems than we are trying to address here in deterministic, generic matching.

7.2 Composability of GenMatcher

There are many techniques to optimize the tries. In this article, we build binary search tries. For further performance improvement, we can add the trie optimization technique to further optimize the trie data structure. For example, we can utilize the stride trie to gain additional speedup. We are able to employ compression techniques to minimize node counts and thus balance the memory usage and search time.

In this article, we use a 64-bit key. *GenMatcher* is able to process a larger key. For the larger key, we can utilize the single instruction multiple data (SIMD) techniques [21–23] to take advantage of wider instructions available in modern processors. To further improve search performance, we can extend this work to support a parallel [29, 38] arbitrary matching architecture to process multiple tries in parallel.

8 CONCLUSIONS

This article proposes *GenMatcher*, a generic, arbitrary, software-only matching mechanism for fast, efficient searches under a given memory threshold. *GenMatcher* employs our proposed grouping algorithm to assign the arbitrary rules with the greatest similarities into the same group. *GenMatcher* generates a minimal number of groups within a memory threshold and is able to build an efficient trie data structure to perform fast binary searching. For a rule table with a size 8,192, *GenMatcher* achieves a mean speedup of 58.9X over a Linear baseline.

REFERENCES

- [1] [n. d.]. The CAIDA Anonymized 2012 Internet Traces—2012. Retrieved from http://www.caida.org/data/passive/passive_2012_dataset.xml.
- [2] Carlos E. Andrade, Mauricio G. C. Resende, Howard J. Karloff, and Flávio K. Miyazawa. 2014. Evolutionary algorithms for overlapping correlation clustering. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO'14)*. ACM, New York, NY, 405–412.
- [3] Arindam Banerjee, Chase Krumpelman, Joydeep Ghosh, Sugato Basu, and Raymond J. Mooney. 2005. Model-based overlapping clustering. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD'05)*. ACM, New York, NY, 532–537. DOI: <https://doi.org/10.1145/1081870.1081932>
- [4] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda. 2016. Adaptive and dynamic design for MPI tag matching. In *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–10. DOI: <https://doi.org/10.1109/CLUSTER.2016.69>
- [5] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. *CoRR* abs/1202.3205 (2012). arxiv:1202.3205
- [6] Dana Drachler, Martin Vechev, and Eran Yahav. 2014. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, NY, 343–356. DOI: <https://doi.org/10.1145/2555243.2555269>
- [7] M. Elmahgiubi, O. Ahmed, S. Areibi, and G. Grewal. 2016. Efficient algorithm selection for packet classification using machine learning. In *Proceedings of the 2016 IEEE 21st International Workshop on Computer Aided Modelling and Design of Communication Links and Networks (CAMAD)*. 24–30. DOI: <https://doi.org/10.1109/CAMAD.2016.7790325>
- [8] Message Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard Version 3.1. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

- [9] Yi Gu and Chaoli Wang. 2010. A study of hierarchical correlation clustering for scientific volume data. In *Proceedings of the 6th International Conference on Advances in Visual Computing—Volume Part III (ISVC'10)*. Springer-Verlag, Berlin, 437–446.
- [10] P. He, G. Xie, K. Salamatian, and L. Mathy. 2014. Meta-algorithms for software-based packet classification. In *2014 IEEE 22nd International Conference on Network Protocols*. 308–319. DOI: <https://doi.org/10.1109/ICNP.2014.53>
- [11] Shane V. Howley and Jeremy Jones. 2012. A non-blocking internal binary search tree. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*. ACM, New York, NY, 161–171. DOI: <https://doi.org/10.1145/2312005.2312036>
- [12] C. L. Hsieh and N. Weng. 2016. Many-field packet classification for software-defined networking switches. In *Proceedings of the 2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 13–24. DOI: <https://doi.org/10.1145/2881025.2881036>
- [13] T. Inoue, T. Mano, K. Mizutani, S. I. Minato, and O. Akashi. 2014. Rethinking packet classification for global network view of software-defined networking. In *Proceedings of the 2014 IEEE 22nd International Conference on Network Protocols*. 296–307. DOI: <https://doi.org/10.1109/ICNP.2014.52>
- [14] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. 2014. SAX-PAC (scalable and eXpressive PAcet classification). In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, New York, NY, 15–26. DOI: <https://doi.org/10.1145/2619239.2626294>
- [15] K. Kogan, S. I. Nikolenko, P. Eugster, A. Shalimov, and O. Rottenstreich. 2016. FIB efficiency in distributed platforms. In *Proceedings of the 2016 IEEE 24th International Conference on Network Protocols (ICNP)*. 1–10. DOI: <https://doi.org/10.1109/ICNP.2016.7784452>
- [16] K. Kogan, S. I. Nikolenko, P. Eugster, A. Shalimov, and O. Rottenstreich. 2017. Efficient FIB representations on distributed platforms. *IEEE/ACM Transactions on Networking* 99 (2017), 1–14. DOI: <https://doi.org/10.1109/TNET.2017.2728642>
- [17] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. 2016. Exploiting order independence for scalable and expressive packet classification. *IEEE/ACM Transactions on Networking* 24, 2 (April 2016), 1251–1264. DOI: <https://doi.org/10.1109/TNET.2015.2407831>
- [18] Jungwon Lee, Hayoung Byun, Ju Hyoung Mun, and Hyesook Lim. 2017. Utilizing 2-D leaf-pushing for packet classification. *Computer Communications* 103 (2017), 116–129.
- [19] B. Leibe, K. Mikolajczyk, and B. Schiele. 2006. Efficient clustering and matching for object class recognition. In *Proc. BMVC*. 81.1–81.10. doi:10.5244/C.20.81.
- [20] Pierre Lemarini, Khalid Hasanov, Srikanth Venugopal, and Kostas Katrinis. 2016. Architecting malleable MPI applications for priority-driven adaptive scheduling. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI'16)*. ACM, New York, NY, 74–81. DOI: <https://doi.org/10.1145/2966884.2966907>
- [21] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, New York, NY, 289–300. DOI: <https://doi.org/10.1145/2463676.2465322>
- [22] Yinan Li and Jignesh M. Patel. 2014. WideTable: An accelerator for analytical data processing. *Proc. VLDB Endow.* 7, 10 (June 2014), 907–918. DOI: <https://doi.org/10.14778/2732951.2732965>
- [23] Hyesook Lim and Ha Young Byun. 2015. Packet classification using a bloom filter in a leaf-pushing area-based quadtree. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'15)*. IEEE Computer Society, Washington, D.C., 183–184.
- [24] A. X. Liu, C. R. Meiners, and E. Torng. 2010. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking* 18, 2 (April 2010), 490–500. DOI: <https://doi.org/10.1109/TNET.2009.2030188>
- [25] A. X. Liu, C. R. Meiners, and Y. Zhou. 2008. All-match based complete redundancy removal for packet classifiers in TCAMs. In *IEEE INFOCOM 2008—The 27th Conference on Computer Communications*. DOI: <https://doi.org/10.1109/INFOCOM.2008.31>
- [26] Mohammad Lotfollahi, Ramin Shirali Hossein Zade, Mahdi Jafari Siavoshani, and Mohammadsadegh Saberian. 2017. Deep packet: A novel approach for encrypted traffic classification using deep learning. *CoRR* abs/1709.02656 (2017). arxiv:1709.02656 DOI: <http://arxiv.org/abs/1709.02656>
- [27] H. Lu and S. Sahni. 2007. $O(\log W)$ multidimensional packet classification. *IEEE/ACM Transactions on Networking* 15, 2 (April 2007), 462–472. DOI: <https://doi.org/10.1109/TNET.2007.892845>
- [28] Chen Luo, Wei Pang, and Zhe Wang. 2014. *Semi-supervised Clustering on Heterogeneous Information Networks*. Springer International Publishing, Cham, 548–559. DOI: https://doi.org/10.1007/978-3-319-06605-9_45
- [29] Yadi Ma, Suman Banerjee, Shan Lu, and Cristian E. Estan. 2010. Leveraging parallelism for multi-dimensional packet classification on software routers. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'10)*. ACM, New York, NY, 227–238. DOI: <https://doi.org/10.1145/1811039.1811065>

- [30] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00)*. ACM, New York, NY, 169–178. DOI : <https://doi.org/10.1145/347090.347123>
- [31] Luke Mchale, Jasson Case, Paul V. Gratz, and Alex Sprintson. 2014. Stochastic pre-classification for SDN data plane matching. In *Proceedings of the 2014 IEEE 22nd International Conference on Network Protocols (ICNP'14)*. IEEE Computer Society, Washington, D.C., 596–602. DOI : <https://doi.org/10.1109/ICNP.2014.95>
- [32] C. R. Meiners, A. X. Liu, and E. Torng. 2012. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking* 20, 2 (April 2012), 488–500. DOI : <https://doi.org/10.1109/TNET.2011.2165323>
- [33] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel. 2011. Split: Optimizing space, power, and throughput for TCAM-based classification. In *Proceedings of the 2011 ACM/IEEE 7th Symposium on Architectures for Networking and Communications Systems*. 200–210. DOI : <https://doi.org/10.1109/ANCS.2011.36>
- [34] S. I. Nikolenko, K. Kogan, G. Rétvári, E. R. Bérczi-Kovács, and A. Shalimov. 2016. How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes. In *Proceedings of the 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 521–526. DOI : <https://doi.org/10.1109/INFCOMW.2016.7562132>
- [35] Xinghao Pan, Dimitris S. Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. 2015. Parallel correlation clustering on big graphs. *CoRR* abs/1507.05086 (2015). arxiv:1507.05086
- [36] Y. Qi, B. Xu, F. He, X. Zhou, J. Yu, and J. Li. 2007. Towards optimized packet classification algorithms for multi-core network processors. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP'07)*. DOI : <https://doi.org/10.1109/ICPP.2007.82>
- [37] Y. Qu, S. Zhou, and V. K. Prasanna. 2013. Scalable many-field packet classification on multi-core processors. In *Proceedings of the 2013 25th International Symposium on Computer Architecture and High Performance Computing*. 33–40. DOI : <https://doi.org/10.1109/SBAC-PAD.2013.29>
- [38] Akhtar Rasool and Nilay Khare. 2013. Generalized parallelization of string matching algorithms on SIMD architecture. *International Journal of Computer Science and Information Security* 11 (2013), 6–16.
- [39] Gábor Rétvári, János Tapolcai, Attila Körösi, András Majdán, and Zsolt Heszberger. 2013. Compressing IP forwarding tables: Towards entropy bounds and beyond. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)*. 111–122.
- [40] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous. 2001. Survey and taxonomy of IP address lookup algorithms. *IEEE Network* 15, 2 (Mar 2001), 8–23.
- [41] Y. K. Sia, H. G. Goh, S. Y. Liew, and M. L. Gan. 2015. Spanning multi-tree algorithm for node and traffic balancing in multi-sink wireless sensor networks. In *Proceedings of the 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*. 2190–2195. DOI : <https://doi.org/10.1109/FSKD.2015.7382292>
- [42] David Sidler, Zsolt István, Muhsen Owaid, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. ACM, New York, NY, 403–415. DOI : <https://doi.org/10.1145/3035918.3035954>
- [43] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman. 2010. Building scalable virtual routers with trie braiding. In *2010 Proceedings IEEE INFOCOM*. 1–9. DOI : <https://doi.org/10.1109/INFCOM.2010.5461960>
- [44] Kiri Wagstaff and Claire Cardie. 2000. Clustering with instance-level constraints. In *Proceedings of the 17th International Conference on Machine Learning (ICML'00)*. Morgan Kaufmann, San Francisco, CA, 1103–1110.
- [45] Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schrödl. 2001. Constrained K-means clustering with background knowledge. In *Proceedings of the 18th International Conference on Machine Learning (ICML'01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 577–584.
- [46] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X. Liu, Qi Li, and Laurent Mathy. 2014. Guarantee IP lookup performance with FIB explosion. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 39–50. DOI : <https://doi.org/10.1145/2740070.2626297>
- [47] Fang Yu, R. H. Katz, and T. V. Lakshman. 2004. Gigabit rate packet pattern-matching using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP'04)*. 174–183. DOI : <https://doi.org/10.1109/ICNP.2004.1348108>
- [48] Wang Z. 2015. The application of deep learning on traffic identification. *BlackHat* (2015). <https://www.blackhat.com/docs/us-15/materials/us-15-Wang-The-Applications-Of-Deep-Learning-On-Traffic-Identification-wp.pdf>.

Received May 2018; revised August 2018; accepted September 2018