A Computational Approach to Packet Classification

Alon Rashelbach
Technion
alonrs@campus.technion.ac.il

Ori Rottenstreich
Technion
or@technion.ac.il

Mark Silberstein
Technion
mark@ee.technion.ac.il

ABSTRACT

Multi-field packet classification is a crucial component in modern software-defined data center networks. To achieve high throughput and low latency, state-of-the-art algorithms strive to fit the rule lookup data structures into on-die caches; however, they do not scale well with the number of rules.

We present a novel approach, *NuevoMatch*, which improves the memory scaling of existing methods. A new data structure, *Range Query Recursive Model Index* (RQ-RMI), is the key component that enables NuevoMatch to replace most of the accesses to main memory with model inference computations. We describe an efficient training algorithm that guarantees the correctness of the RQ-RMI-based classification. The use of RQ-RMI allows the rules to be compressed into model weights that fit into the hardware cache. Further, it takes advantage of the growing support for fast neural network processing in modern CPUs, such as wide vector instructions, achieving a rate of tens of nanoseconds per lookup.

Our evaluation using 500K multi-field rules from the standard ClassBench benchmark shows a geometric mean compression factor of 4.9×, 8×, and 82×, and average performance improvement of 2.4×, 2.6×, and 1.6× in throughput compared to CutSplit, NeuroCuts, and TupleMerge, all state-of-the-art algorithms[1].

1 INTRODUCTION

Packet classification is a cornerstone of packet-switched networks. Network functions such as switches use a set of *rules* that determine which action they should take for each incoming packet. The rules originate in higher-level domains, such as routing, Quality of Service, or security policies. They match the packets' metadata, e.g., the destination IP-address and/or the transport protocol. If multiple rules match, the rule with the highest *priority* is used.

Packet classification algorithms have been studied extensively. There are two main classes: those that rely on Ternary Content Addressable Memory (TCAM) hardware [13, 20, 23, 28, 37], and those that are implemented in software [3, 8, 21, 22, 34, 36, 41, 44]. In this work, we focus on software-only algorithms that can be deployed in virtual network functions, such as forwarders or ACL firewalls, running on commodity X86 servers.

Software algorithms fall into two major categories: decision-tree based [8, 21, 22, 34, 41, 44] and hash-based [3, 36]. The former use decision trees for indexing and matching the rules, whereas the latter perform lookup via hash-tables by hashing the rule's prefixes. Other methods for packet classification [7, 38] are less common as they either require too much memory or are too slow.

A key to achieving high classification performance in modern CPUs is to ensure that the classifier fits into the CPU on-die cache. When the classifier is too large, the lookup involves high-latency memory accesses, which stall the CPU, as the data-dependent access pattern during the lookup impedes hardware prefetching. Unfortunately, as the number of rules grows, it becomes difficult to maintain the classifier in the cache. In particular, in decision-tree methods, rules are often replicated among multiple leaves of the decision tree, inflating its memory footprint and affecting scalability. Consequently, recent approaches, notably CutSplit [21] and NeuroCuts [22], seek to reduce rule replication to achieve better scaling. However, they still *fail to scale* to large rule-sets, which in modern data centers may reach hundreds of thousands of rules [6]. Hash-based techniques also suffer from poor scaling, as adding rules increases the number of hash-tables and their size.

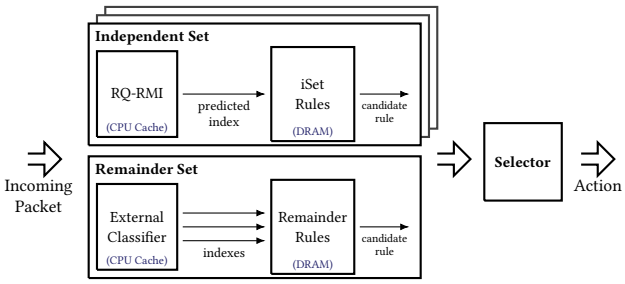


Figure 1: NuevoMatch overview. The rules are divided into Independent Sets indexed by RQ-RMIs and the Remainder Set indexed by any classifier. One RQ-RMI predicts the storage index of the matching rule. The Selector chooses the highest-priority matching rule.

We propose a novel approach to packet classification, *NuevoMatch*, which *compresses* the rule-set index dramatically to fit it entirely into the upper levels of the CPU cache (L1/L2) even for large 500K rule-sets. We introduce a novel *Range Query Recursive Model Index* (RQ-RMI) model, and train it to *learn* the rules' matching sets, turning rule matching into *neural network inference*. We show that RQ-RMI achieves out-of-L1-cache execution by reducing the memory footprint on average by 4.9×, 8×, and 82× compared to recent CutSplit [21], NeuroCuts [22], and TupleMerge [3] on the standard ClassBench [39] benchmarks, and up to 29× for real forwarding rule-sets.

To the best of our knowledge, NuevoMatch is the first to perform packet classification using trained neural network models. NeuroCuts also uses neural nets, but it applies them for optimizing the decision tree parameters during the *offline* tree construction phase; their rule matching still uses traditional (optimized) decision trees. In contrast, NuevoMatch performs classification via RQ-RMIs, which are more space-efficient than decision trees or hash-tables, improving scalability by an order of magnitude.

[1] This work does not raise any ethical issues.

NuevoMatch transforms the packet classification task from memory- to compute-bound. This design is appealing because it is likely to scale well in the future, with rapid advances in hardware acceleration of neural network inference [11, 19, 29]. On the other hand, the performance of both decision trees and hash-tables is inherently limited because of the poor scaling of DRAM access latency and CPU on-die cache sizes (e.g., 1.5× over five years for L1 in Intel's CPUs).

NuevoMatch builds on the recent work on *learned indexes* [18], which applies a Recursive Model Index (RMI) model to indexing key-value pairs. The values are stored in an array, and the RMI is *trained to learn* the mapping function between the keys and the indexes of their values in the array. The model is used to *predict* the index given the key. When applied to databases [18], RMI boosts performance by compressing the indexes to fit in CPU caches.

Unfortunately, RMI is not directly applicable for packet classification. First, a key (packet field) may not have an exact matching value, but match a *rule range*, whereas RMI can learn only exact key-index pairs. This is a fundamental property of RMI: it guarantees correctness only for the keys used during training, but provides no such guarantees for non-existing keys ([18], Section 3.4). Thus, for range matching it requires enumeration of all possible keys in the range, making it too slow. Second, the match is evaluated over multiple packet fields, requiring lookup in a multi-dimensional space. Unfortunately, multi-dimensional RMI [17] requires that the input be flattened into one dimension, which in the presence of wildcards results in an exponential blowup of the input domain, making it too large to learn for compact models. Finally, a key may match multiple rules, with the highest priority one used as output, whereas RMI retrieves only a single index for each key.

NuevoMatch successfully solves these challenges.

RQ-RMI. We design a novel model which can match keys to ranges, with an efficient training algorithm that does not require exhaustive key enumeration to learn the ranges. The training strives to minimize the prediction error of the index, while maintaining a small model size. We show that the models can store indices of 500K ClassBench rules in 35 KB (§5.2.1). We prove that our algorithm *guarantees range lookup correctness* (§3.3).

Multi-field packet classification. To enable multi-field matching with overlapping ranges, the rule-set is split into independent sets with non-overlapping ranges, called *iSets*, each associated with a single field and indexed with its own RQ-RMI model. The iSet partitioning (§3.6) strives to cover the rule-set with as few iSets as possible, discarding those that are too small. The *remainder set* of the rules not covered by large iSets is indexed via existing classification techniques. In practice, the rules in the remainder constitute a small fraction in representative rule-sets, so the remainder index fits into a fast cache together with the RQ-RMIs.

Figure 1 summarizes the complete classification flow. The query of the RQ-RMI models produces the hints for the secondary search that selects one matching rule per iSet. The validation stage selects the candidates with a positive match across all the fields, and a selector chooses the highest priority matching rule.

Conceptually, NuevoMatch can be seen as an *accelerator* for existing packet classification techniques and thus complements them. In particular, the RQ-RMI model is best used for indexing rules with high value diversity that can be partitioned into fewer iSets. We show that the iSet construction algorithm is effective for selecting the rules that can be indexed via RQ-RMI, leaving the rest in the remainder (§5.3.1). The performance benefits of NuevoMatch become evident when it indexes more than 25% of the rules. Since the remainder is only a fraction of the original rule-set, it can be indexed efficiently with smaller decision-trees/hash-tables or will fit smaller TCAMs.

Our experiments[2] show that NuevoMatch outperforms all the state-of-the-art algorithms on synthetic and real-life rule-sets. For example, it is faster than CutSplit, NeuroCuts, and TupleMerge, by 2.7×, 4.4× and 2.6× in latency and 2.4×, 2.6×, and 1.6× in throughput respectively, averaged over 12 rule-sets of 500K ClassBench-generated rules, and by 7.5× in latency and 3.5× in throughput vs. TupleMerge for the real-world Stanford backbone forwarding rule-set.

NuevoMatch supports rule updates by removing the updated rules from the RQ-RMI and adding them to the remainder set indexed by another algorithm that supports fast updates, e.g., TupleMerge. This approach requires periodic retraining to maintain a small remainder set; hence it does not yet support more than a few thousands of updates (§3.9). The algorithmic solutions to directly update RQ-RMI are deferred for future work.

In summary, our contributions are as follows.

- We present an novel RQ-RMI model and a training technique for learning packet classification rules.
- We demonstrate the application of RQ-RMI to multi-field packet classification.
- NuevoMatch outperforms existing techniques in terms of memory footprint, latency, and throughput on challenging rule-sets with up to 500K rules, compressing them to fit into small caches of modern processors.

2 BACKGROUND

This section describes the packet classification problem and surveys existing solutions.

2.1 Classification algorithms

Packet classification is the process of locating a single rule that is satisfied by an input packet among a set of rules. A rule contains a few fields in the packet's metadata. Wildcards define *ranges*, i.e., they match multiple values. Ranges may overlap with each other, i.e., a packet may match several rules, but only the one having the highest priority is selected. Figure 2 illustrates a classifier with two fields and five overlapping matching rules. An incoming packet matches two rules (R^3, R^4), but R^3 is used as it has a higher priority.

Packet classification performance becomes difficult to scale as the number of rules and the number of matching fields grow. Therefore, it has received renewed interest with increased complexity of software-defined data center networks, featuring hundreds of thousands of rules per virtual network function [5] and tens of matching fields (up to 41 in OpenFlow 1.4 [27]).

Decision Tree Algorithms. The rules are viewed as hyper-cubes and packets as points in a multi-dimensional space. The axes of the *rule space* represent different fields and hold non-negative integers.

[2]The source code of NuevoMatch is available in [31].

	IPv4 Address	Port	Priority	Action
R^0	10.10.*.*	10-18	1 (highest)	a_1
R^1	10.10.1.*	15-25	2	a_2
R^2	10.*.*.*	5-8	3	a_3
R^3	10.10.3.*	7-20	4	a_4
R^4	10.10.3.100	19	5 (lowest)	a_5

Incoming packet: 10.10.3.100:19 → Action to take: a_4

Figure 2: Packet classification with two fields: IP address and port.

A recursive partitioning technique divides the rule space into subsets with at most *binth* rules. Thus, to match a rule, a tree traversal finds the smallest subset for a given packet, while a secondary search scans over the subset's rules to select the best match.

Unfortunately, a *rule replication* problem may hinder performance in larger rule-sets by dramatically increasing the tree's memory footprint when a rule spans several subspaces. Early works, such as HiCuts [8] and HyperCuts [34] both suffer from this issue. More recent EffiCuts [41] and CutSplit [21], suggest that the rule set should be split into groups of rules that share similar properties and generate a separate decision-tree for each. NeuroCuts [22], the most recent work in this domain, uses reinforcement learning for optimizing decision tree parameters to reduce its memory footprint, or the number of memory accesses during traversal, by efficiently exploring a large tree configuration space.

Hash-Based Algorithms. Tuple Space Search [36] and recent TupleMerge [3] partition the rule-set into subsets according to the number of prefix bits in each field. As all rules of a subset have the same number of prefix bits, they can act as keys in a hash table. The classification is performed by extracting the prefix bits, in all fields, of an incoming packet, and checking all hash-tables for matching candidates. A secondary search eliminates false-positive results and selects the rule with the highest priority.

Hash-based techniques are effective in an *online classification* problem with frequent rule updates, whereas decision trees are not. However, decision trees have been traditionally considered faster in classification. Nevertheless, the recent TupleMerge hash-based algorithm closes the gap and achieves high classification throughput while supporting high performance updates.

2.2 Poor performance with large rule-sets

The packet classification performance of all the existing techniques does not scale well with the number of rules. This happens because their indexing structures spill out of the fast L1/L2 CPU caches into L3 or DRAM. Indeed, as we show in our experiments (§5), TupleMerge and NeuroCuts exceed the 1MB L2 cache with 100K rules and CutSplit with 500K rules. However, keeping the *entire* indexing structure in fast caches is critical for performance. The inherent lack of access locality in hash and tree data structures, combined with the data-dependent nature of the accesses, make hardware prefetchers ineffective for hiding memory access latency. Thus, the performance of all lookups drops dramatically.

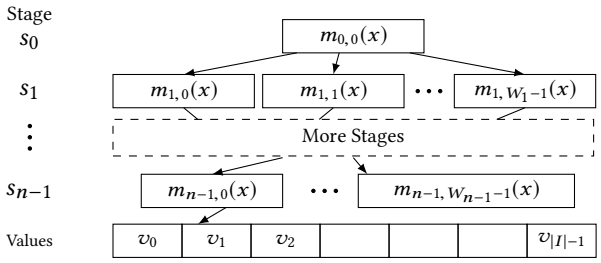


Figure 3: RMI model structure and inference [18].

The performance drop is significant even when the data structures fit in the L3 cache. This cache is shared among all the cores, whereas L1 and L2 caches are per-core. Thus, L3 is not only slower (up to 90 cycles in recent X86 CPUs), but also suffers from cache contention, e.g., when another core runs a cache-demanding workload and causes cache trashing. We observe the effects of L3 contention in §5.2.1.

NuevoMatch aims to provide more space efficient representation of the rule index to scale to large rule-sets.

3 NUEVOMATCH CONSTRUCTION

We first explain the RMI model for learned indexes which we use as the basis, explain its limitations, and then show our solution that overcomes them.

3.1 Recursive Model Index

Kraska et al. [18] suggest using machine-learning models for storing key-value pairs instead of conventional data structures such as B-trees or hash tables. The values are stored in a *value array*, and a *Recursive Model Index* (RMI) is used to retrieve the value given a key. Specifically, RMI *predicts the index* of the corresponding value in the value array using a model that *learned* the underlying key-index mapping function.

The main insight is that any index structure can be expressed as a continuous monotonically increasing function $y = h(x) : [0, 1] \mapsto [0, 1]$, where x is a key scaled down uniformly into $[0, 1]$, and y is the index of the respective value in the value array scaled down uniformly into $[0, 1]$. RMI is trained to learn $h(x)$. The resulting *learned index model* $\widehat{h}(x)$ performs lookups in two phases: first it computes the *predicted* index $\widehat{y} = \widehat{h}(key)$, and then performs a *secondary search* in the array, in the vicinity ϵ of the predicted index, where ϵ is the *maximum index prediction error* of the model, namely $|\widehat{h}(key) - h(key)| \leq \epsilon$.

Model structure. RMI is a hierarchical model made of several (n) stages (Figure 3). Each stage i includes W_i *submodels* $m_{i,j}$, $j < W_i$, where W_i is the *stage width*. The first stage has a single submodel. Each successive stage has a larger width. The submodels in each stage are trained on a progressively smaller subset of the input keys, *refining* the index prediction toward the submodels in the leaves. Thus, each key-index pair is learned by one submodel at each stage, with the leaf submodel producing the index prediction.

RMI is a generic structure; a variety of machine learning models or data structures can be used as submodels, such as regression

models or B-trees. The type of the submodels, the number of stages and the width of each stage are configured prior to training.

Training. Training is performed stage by stage.

First stage. The submodel in stage $m_{0,0}$ is trained on the *whole* data set. Then, the input key-index pairs are split into W_1 disjoint subsets. The input partitioning is performed as follows. For each input key-index pair $\{key : idx\}$ we compute the submodel prediction $\widehat{j} = m_{0,0}(key)$, satisfying $\widehat{j} \in [0, 1)$. The output $\widehat{j}$ is used to obtain $j = \lfloor \widehat{j} \cdot W_1 \rfloor$ which is the index of the submodel in stage 1, $m_{1,j}$, to be used for learning $\{key : idx\}$. We call the subset of the input to be learned by model $m_{i,j}$ as model *input responsibility domain* $R_{i,j}$, or responsibility for short. $R_{0,0}$ is the whole input.

Internal stages. The submodels in stage i, $m_{i,j}$, are trained on the keys in $R_{i,j}$ ($j < W_i$). After training, the responsibilities of the submodels in stage $i + 1$ are computed, and the process continues until the last stage.

Last stage. The submodels of the last stage must predict the actual index of the matching value in the value array. However, a submodel may have a prediction error. Therefore, RMI uses the model prediction as a *hint*. The matching value is found by searching in the value array in the vicinity of the predicted index, as defined by the maximum error bound ϵ of the model. Note that ϵ should be valid for *all* input key-index pairs. To compute ϵ, RMI exhaustively computes the submodel prediction for each input key in its responsibility. Submodels with a high error bound are retrained or converted to B-trees.

Inference. Given a key, we iteratively evaluate each submodel stage after stage, from $m_{0,0}$. We use the prediction in stage $i - 1$ to select a submodel in stage i, until we reach the last stage. The last selected submodel predicts the index in the value array. This index $\widehat{i}$ determines the range for the secondary search in the value array that spans $[\widehat{i} - \epsilon, \widehat{i} + \epsilon]$.

3.2 RMI limitations

Direct application of RMI to indexing packet classification rules is not possible for the following reasons:

No support for range matching[3]**.** RMI allows only an exact match for a given key, whereas packet classification requires retrieving rules with matching *ranges* as defined by wildcards. This problem is fundamental: RMI exhaustively enumerates *all* the keys in all the ranges to calculate the submodel responsibility and the maximum model prediction error (see the underlined parts of the training algorithm). In other words, all the values in the range must be materialized into key-index pairs for RMI to learn them, since RMI *does not guarantee correct lookup for keys not used in training* [18]. The original paper sketches a few possible solutions, however, they either rely on model monotonicity (while we do not) or use smarter yet still expensive enumeration techniques.

Slow multi-dimensional indexing. RMI is ineffective for multi-dimensional indexes because the proposed solution [17] leads to generating an exponential number of rules in the presence of wildcards. For example, a single rule with wildcards in destination IP (0.0.0.*), port (10-100), and protocol (TCP/UDP) results in 46,592 distinct key-index pairs. Since the input domain becomes too large, it requires a large model that exceeds the CPU cache.

In the following we outline the solutions to these challenges We first discuss Range-Query RMI (RQ-RMI), which extends RMI to perform *range-value queries* in a one-dimensional index where ranges do not overlap (§3.3-§3.5). We then show how to apply RQ-RMI in multi-dimensional index space with overlaps (§3.6-§3.7).

[3]The RMI paper also uses the term *range index* while applying RMI to range index data structures (i.e., B-trees) that can quickly retrieve all stored keys in a requested numerical range. Our work is fundamentally different: given a key it retrieves *the index of its matching range.*

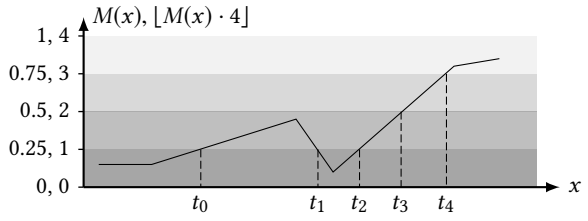


Figure 4: Transition inputs ($t_0, ..., t_4$) for a piece-wise linear function with the output domain=4.

3.3 One-dimensional RQ-RMI

We first seek to find a way to perform range matching over a set of non-overlapping ranges in one dimension.

There are two basic ideas:

Sampling. Each submodel $m_{i,j}$ is trained by generating a uniform sample of key-index pairs from input ranges in its responsibility. The samples are generated on-the-fly for each submodel (§3.5.4).

Analytical error bound estimation for ranges. We eliminate the RMI's requirement for exhaustive key-value enumeration during training by making the following observation: *if a submodel is a piece-wise linear function, the worst-case error bound ϵ can be computed analytically*, thereby enabling efficient learning of ranges.

The intuition behind this observation is illustrated in Figure 4. It shows the graph of some piece-wise linear function which represents a submodel M whose outputs are quantized into integers in $[0, 4)$, i.e., M predicts the index in an array of size 4. We call the inputs for which this function changes its quantized output *transition inputs* $t_i \in T$. In turn, transition inputs determine the region of inputs with the *same* quantized output. Therefore, given an input range in the model's responsibility, to compute the model's maximum prediction error for any key in that range, it suffices to evaluate the prediction error in the transition inputs that fall in the range. We describe the training algorithm that relies on these observations in Section 3.4. We now provide a more formal description, but leave most of the proofs in the Appendix.

3.4 Using a neural network as a submodel

We choose to use a 3-layer fully-connected neural network (NN) with a single hidden layer and ReLU activation A. Such NNs have been suggested in the original RMI paper [18]; however, they did not leverage their properties for accelerating error bound computations.

We denote a submodel as $m_{i,j}$, and define it as follows.

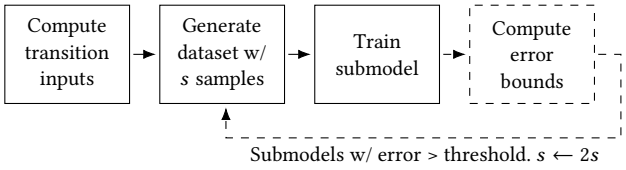


Figure 5: The submodel training process. The additional phase for training submodels in the leaves is depicted with dashed lines.

Definition 3.1 (RQ-RMI submodel). Denote the output of a 3-layer fully-connected neural network as:

$$N_{i,j}(x) = A\big(x \cdot \boldsymbol{w_1} + \boldsymbol{b_1}\big) \times \boldsymbol{w_2} + b_2$$

where x is a scalar input, $\boldsymbol{w_1}, \boldsymbol{b_1}$ are the weight and bias row-vectors for layer 1 (hidden layer), and $\boldsymbol{w_2}, b_2$ are the weight column-vector and bias scalar for layer 2. Note that $N_{i,j}(x)$ is a scalar. The ReLU function A applies a function a on each element of an input vector:

$$a(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0. \end{cases}$$

The submodel output, denoted $M_{i,j}(x)$, is defined as follows:

$$M_{i,j}(x) = H\big(N_{i,j}(x)\big)$$

where $H(x)$ trims the output domain to be in $[0, 1)$.

Corollary 3.2. *$M_{i,j}(x)$ is a piece-wise linear function.*

3.5 RQ-RMI training

We use Corollary 3.2 to compute the transition inputs and the responsibility of the submodels. We provide a simplified description; see Appendix for the precise explanation.

3.5.1 Overview. RQ-RMI training is similar to RMI's. It is performed stage by stage. Figure 5 illustrates the training process for one stage. We start by training the single submodel in the first stage using the entire input domain. Next, we calculate its transition inputs (§3.5.2) and use them to find the responsibilities of the submodels in the following stage (§3.5.3). We proceed by training submodels in the subsequent stage using designated datasets we generate based on the submodels' responsibilities (§3.5.4). We repeat this process until all submodels in all internal stages are trained. For the submodels in the leaves (last stage), there is an additional phase (dashed lines in Figure 5). After training, we calculate their error bounds and retrain the submodels that do not satisfy a predefined error threshold (§3.5.6).

3.5.2 Computing transition inputs. Given a trained submodel m we can analytically find all its linear regions, and respectively the inputs delimiting them, which we call *trigger inputs* g_l. For all inputs in the region $[g_l, g_{l+1}]$, the model function, denoted as $M(x)$, is linear by construction. On the other hand, the uniform output quantization defines a step-wise function $Q = \lfloor M(x) \cdot W \rfloor / W$, where W is the size of the quantized output domain (Figure 4). Thus, for each input region $[g_l, g_{l+1}]$, the set of transition inputs $t_l \in T$ are those where $M(x)$ and Q intersect.

3.5.3 Computing the responsibilities of submodels in the following stage. Given a trained submodel $m_{i,j}$ in an internal stage i, we say that it *maps* a *key* to a submodel $m_{i+1,k}$, $k < W_{i+1}$, if $\lfloor M_{i,j}(key) \cdot W_{i+1} \rfloor = k$. As discussed informally earlier, the responsibility $R_{i+1,k}$ of $m_{i+1,k}$ is defined as all the inputs which are mapped by submodels in stage i to $m_{i+1,k}$. In other words, the trained submodels at stage i define the responsibility of untrained submodels at stage $i+1$.

Knowing the responsibility of a submodel is crucial, as it determines the subset of the inputs used to train the submodel. RMI exhaustively evaluates all the inputs, which is inefficient. Instead, we compute $R_{i+1,k}$ using the transition inputs of $m_{i,j}$. In the following, we assume for clarity that $R_{i,j}$ is contiguous, and $m_{i,j}$ is the only submodel at stage i.

We compute $R_{i+1,k}$ by observing that it is composed of all the inputs in the regions (t_l, t_l+1) that map to submodel $m_{i+1,k}$, where $t_l \in T_{i,j}$ are transition inputs of $m_{i,j}$. By construction, the inputs in the region between two adjacent transition points map to the same output. Then, it suffices to compute the output of $m_{i,j}$ for its transition points, and choose the respective input ranges that are mapped to $m_{i+1,k}$.

3.5.4 Training a submodel with ranges using sampling. Up to this point, we used only key-index pairs as model inputs. Now we focus on training on input *ranges*. A range can be represented as all the keys that fall into the range, all associated with the *same* index of the respective rule. For example, 10.1.1.0-10.1.1.255 includes 256 keys. Our goal is to train a model such that given a key in the range, the model predicts the correct index. Enumerating all the keys in the ranges is inefficient. Instead, we use sampling as follows.

We generate the training key-index pairs by uniformly sampling the submodel's responsibility. We start with a low sampling frequency. A sample is included in the training set if there is an input rule range that matches the sampled key. Thus, the number of samples per input range is proportional to its relative size in the submodel's responsibility. Note that some input ranges (or individual keys) might not be sampled at all. Nevertheless, they will be matched correctly as we explain further.

3.5.5 Submodel training. We train submodels on the generated datasets using supervised learning and Adam optimizer [14] with a mean squared error loss function.

3.5.6 Computing error bounds. Given a trained submodel in the last stage, we compute the prediction error bound for *all inputs in its responsibility* by evaluating the submodel on its transition inputs. The prediction error is computed also for the inputs that were not necessarily sampled, guaranteeing match correctness. If the error is too large, we double the number of samples, regenerate the key-index pairs, and retrain the submodel. Training continues until the target error bound is attained or after a predefined number of attempts. If training does not converge, the target error bound may be increased by the operator. The error bound determines the *search distance of the secondary search*; hence a larger bound causes lower system performance. We evaluate this tradeoff later (§5.3.4).

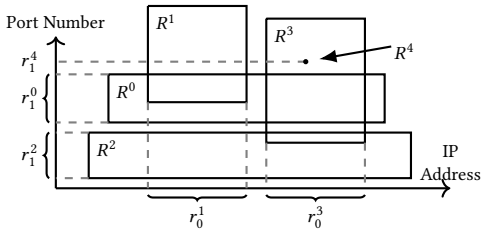


Figure 6: Rules from Figure 2 are split into two iSets: $\{R^0, R^2, R^4\}$ (by port), and $\{R^1, R^3\}$ (by IP).

3.6 Handling multi-dimensional queries with range overlaps

NuevoMatch supports overlapped ranges and matching over multiple dimensions, i.e., packet fields, by combining two simple ideas: partitioning the rule-set into disjoint independent sets (*iSets*), and performing multi-field validation of each rule. In the following, we use the terms dimension and field interchangeably.

Partitioning. Each iSet contains rules that do not overlap in *one specific dimension*. We refer to the *coverage* of an iSet as the fraction of the rules it holds out of those in the input. One iSet may cover all the rules if they do not overlap in at least one dimension, whereas the same dimension with many overlapping ranges may require multiple iSets. Figure 6 shows the iSets for the rules from Figure 2.

Each iSet is indexed by one RQ-RMI. Thus, to find the match to a query with multiple fields, we query all RQ-RMIs (in parallel), each over the field on which it was trained. Then, the highest priority result is selected as the output.

Each iSet adds to the total memory consumption and computational requirements of NuevoMatch. Therefore, we introduce a heuristic that strives to find the smallest number of iSets that cover the largest part of the rule-set (§3.6.1).

Multi-field validation. Since an RQ-RMI builds an index of the rules over a single field, it might retrieve a rule which does not match against other fields. Hence, each rule returned by an RQ-RMI is validated across all fields. This enables NuevoMatch to avoid indexing all dimensions, yet obtain correct results.

3.6.1 iSet partitioning. We introduce a greedy heuristic that repetitively constructs the largest iSet from the input rules, producing a group of iSets. To find the largest iSet over one dimension, we use a classical *interval scheduling maximization* algorithm [15]. The algorithm sorts the ranges by their upper bounds, and repetitively picks the range with the smallest upper bound that does not overlap previously selected ranges.

We apply the algorithm to find the largest iSet in each field. Then we greedily choose the largest iSet among all the fields and remove its rules from the input set. We continue until exhausting the input. This heuristic is sub-optimal but quite efficient. We plan to improve it in future work.

Having a larger number of fields in a rule-set might help improve coverage. For example, if the rules that overlap in one field do not overlap in another and vice versa, two iSets cover the whole rule-set, requiring more iSets for each field in isolation.

3.7 Remainder set and external classifiers

Real-world rule-sets may require many iSets for full coverage, with a single rule per iSet in the extreme cases. Using separate RQ-RMIs for such iSets will hinder performance. Therefore, we merge small iSets into a single *remainder set*. The rules in the remainder set are indexed using an *external* classifier. Each query is performed on both the RQ-RMI and the external classifier.

In essence, NuevoMatch serves as an *accelerator* for the external classifier. Indeed, if rule-sets are covered using a few large iSets, the external classifier needs to index a small remainder set that often fits into faster memory, so it can be very fast.

Two primary factors determine the end-to-end performance: (1) the number of iSets required for high coverage (depends on the rule-set), and; (2) the number of iSets for achieving high performance (set by an operator).

Our evaluation (§5.3.1) shows that most of the evaluated rule-sets can be covered with high coverage above 90% with only 2-3 iSets. This is enough to accelerate the external classifier, as is evident from the performance results. On the other hand, the choice of the number of iSets depends on the external classifier properties, in particular, its sensitivity to memory footprint. We analyze this tradeoff in §5.3.

Worst-case inputs. Some rule-sets cannot achieve good coverage with only a few iSets. For example, a rule-set with a single field whose ranges overlap requires too many iSets to be covered.

To obtain a better intuition about the origins of worst-case inputs, we consider the notion of *rule-set diversity* for rule-sets with exact matches. Rule-set diversity in a field is the number of unique values in it across the rule-set, divided by the total number of rules. *The rule-set diversity is an upper bound on the fraction of rules in the largest iSet of that field.* In other words, low diversity implies that using the field for iSet partitioning would result in poor coverage.

We can also identify challenging rule-sets with ranges. We define *rule-set centrality* as the maximal number of rules that each pair of them overlap (they all share a point in a multi-dimensional space). *The rule-set centrality is a lower bound on the number of iSets required for full coverage.*

The diversity and centrality metrics can indicate the potential of NuevoMatch to accelerate the classification of a rule-set. On the positive side, our iSet partitioning algorithm is effective at segregating the rules that cannot be covered well from the rules that can, thereby accelerating the remainder classifier as much as possible for a given rule-set. We analyze this property in §5.3.3.

3.8 Putting it all together

We briefly summarize all the steps of NuevoMatch.

Training

(1) Partition the input into iSets and a remainder set
(2) Train one RQ-RMI on each iSet
(3) Construct an external classifier for the remainder set

Lookup

(1) Query all the RQ-RMIs
(2) Query the external classifier
(3) Collect all the outputs, return the highest-priority rule

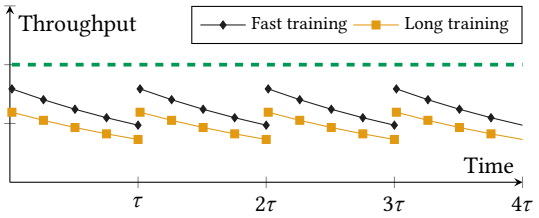


Figure 7: Updates impact on Throughput over time. An upper bound (in green) is for zero training time.

3.9 Rule Updates

We explain how NuevoMatch can support updates with a limited performance degradation.

Firstly, an external classifier used for the remainder must support updates. Among the evaluated external classifiers only TupleMerge is designed for fast updates.

Secondly, we distinguish four types of updates: *(i)* a change in the *rule action*; *(ii)* rule *deletion (iii)* rule *matching set change*; *(iv)* rule *addition*.

The first two types of updates are supported without performance degradation, and require a lookup followed by an update in the value array. However, if an update modifies a rule's matching set or adds a new rule, it might require modifications to the RQ-RMI model. We currently do not know an algorithmic way to update RQ-RMI without retraining; therefore, an updated rule is always added to the remainder set.

Unfortunately, this design leads to gradual performance degradation, as updates are likely to increase the remainder set. Accordingly, the model is retrained on the updated rule-set, either periodically or when a large performance degradation is detected. Updates occurring while retraining are accommodated in the following batch of updates.

Estimating sustained update rate. Let r and u be the total number of rules and the number of updates that move a rule to the remainder, respectively; u can be smaller than the real rate of rule updates. We assume that the updates are independent and uniformly distributed among the r rules. For each rule update, a rule is modified w.p. (with probability) $\frac{1}{r}$. Thus a rule is not modified in any of the updates w.p. $(1-\frac{1}{r})^u \approx e^{-u/r}$. The expected number of unmodified rules is $r \cdot (1-\frac{1}{r})^u \approx r \cdot e^{-u/r}$. Throughput behaves as a weighted average between that of NuevoMatch and the remainder implementation, based on the number of rules in each.

Figure 7 illustrates the throughput over time for different retraining rates given a certain update rate. If retraining is invoked every τ time units, the slower the training process, the worse the performance degradation.

With these update estimates, using the measured speedup as a function of the fraction of the remainder (§5.3.3), NuevoMatch can sustain up to 4k updates per second for 500K rule-sets, yielding about half the speedup of the update-free case, assuming a minute-long training. These results indicate the need for speeding up training, but we conjecture there might be a more efficient way to perform updates directly in RQ-RMI without complete re-training of all submodels. Accelerating updates is left for future work.

4 IMPLEMENTATION DETAILS

RQ-RMI structure. The number of stages and the width of each stage depend on the number of rules to index. We increase the width of the last stage from 16 for 10K rules to as much as 512 for 500K. See Table 4 in the Appendix.

Submodel structure. Each submodel is a fully connected 3-layer neural net with 1 input, 1 output, and 8 neurons in the hidden layer with ReLU activation. This structure affords an efficient vectorized implementation (see below).

Training. We use TensorFlow [1] to train each submodel on a CPU. Training a submodel requires a few seconds, but the whole RQ-RMI may take up to a few minutes (see §5.3.4). We believe, however, that a much faster training time could be achieved with more optimizations, i.e., replacing TensorFlow (known for its poor performance on small models). We leave it for future work.

iSet partitioning. We implement the iSet partitioning algorithm using Python. The partitioning takes at most a few seconds and is negligible compared to RQ-RMI training time.

Inference and secondary search. We implement RQ-RMI inference in C++. For each iSet we sort the rules by the value of the respective field to optimize the secondary search. To reduce the number of memory accesses, we pack multiple field values from different rules in the same cache line.

Handling long fields. Both iSet partitioning algorithms and RQ-RMI models map the inputs into single-precision floating-point numbers. This allows the packing of more scalars in vector operations, resulting in faster inference. While enough for 32-bit fields, doing so might cause poor performance for fields of 64-bits and 128-bits.

We compared two solutions: (1) splitting the fields into 32-bit parts and treating each as a distinct field, and (2) using a single-precision floating-point to express long fields. The two showed similar results for iSet partitioning with MAC addresses, while with IPv6, splitting into multiple fields worked better. Note that both the secondary search and the validation phases are not affected because the rules are stored with the original fields.

Vectorization. We accelerate the inference by using wide CPU vector instructions. Specifically, with 8 neurons in the hidden layer of each submodel, computing the prediction involves a handful of vector instructions. Validation is also vectorized.

Table 1 shows the effectiveness of vectorization. The use of wider units speeds up inference, highlighting the potential for scaling NuevoMatch in future CPUs.

Parallelization. NuevoMatch lends itself to parallel execution where iSets and the remainder classifier run in parallel on different CPU cores. The system receives the packets and enqueues each for

Table 1: Submodel acceleration via vectorization. Methods are annotated with the number of floats per single instruction.

Instruction set (width)	Serial(1)	SSE(4)	AVX(8)
Inference Time (ns)	126	62	49

execution into the worker threads. The threads are statically allocated to run RQ-RMI or the external classifier with a balanced load between the cores.

Note that since RQ-RMI are small and fit in L1, running them on a separate core enables L1-cache-resident executions even if the remainder classifier is large. Such an efficient cache utilization could not have been achieved with other classifiers running on two cores.

Early termination. One drawback of the parallel implementation is that the slowest thread determines the execution time. Our experiments show that the remainder classifier is the slowest one. It holds only a small fraction of the rules, so it returns an empty set for most of the queries, which in turn leads to the worst-case lookup time. In TupleMerge, for example, a query which does not find any matching rules results in a search over all tables, whereas in the average case some tables are skipped.

Instead, we query the remainder *after* obtaining the results from the iSets, and terminate the search when we determine that the target rule is not in the remainder.

To achieve that, we make minor changes to existing classification techniques. Specifically, in decision-tree algorithms, we store in each node the maximum priority of all the sub-tree rules. Whenever we encounter a maximum priority that is lower than that found in the iSets, we terminate the tree-walk. The changes to the hash-based algorithms are similar.

We call this optimization *early termination*. With this optimization, both the iSets and the remainder are queried on the same core. While a parallel implementation is possible, it incurs higher synchronization overheads among threads.

5 EVALUATION

In the evaluation, we pursued the following goals.

(1) Comparison of NuevoMatch with the state-of-the-art algorithms TupleMerge [3], CutSplit [21], and NeuroCuts [22];
(2) Systematic analysis of the performance characteristics, including coverage in challenging data sets, the effect of RQ-RMI error bound, and training time.

5.1 Methodology

We ran the experiments on Intel Xeon Silver 4116 @ 2.1 GHz with 12 cores, 32KB L1, 1024KB L2, and 16MB L3 caches, running Ubuntu 16.04 (Linux kernel 4.4.0). We disable power management for stable measurements.

Evaluated configurations. CutSplit (CS) is set with *binth* = 8, as suggested in [21].

For NeuroCuts (NC), we performed a hyperparameter sweep and selected the best classifier per rule-set. As recommended in [22], we focused on both top-mode partitioning and reward scaling. We ran the search on three 12-core Intel machines, allocating six hours per configuration to converge. In total, we ran NC training for up to 36 hours per rule-set. In addition, we developed a C++ implementation of NC for faster evaluation of the generated classifiers, much faster than the authors' Python-based prototype.

TupleMerge (TM) is used with the version that supports updates with *collision-limit* = 40, as suggested in [3].

NuevoMatch (NM) was trained with a maximum error threshold of 64. We present the analysis of the sensitivity to the chosen parameters and training times in §5.3.2.

Multi-core implementation. We run a parallel implementation on two cores. NuevoMatch allocates one core for the remainder computations and the second for the RQ-RMIs. For CS, NC, and TM, we ran two instances of the algorithm in parallel on two cores using two threads (i.e., no duplication of the rules), splitting the input equally between the cores. We discarded iSets with coverage below 25% for comparisons against CS and NC, and below 5% for comparisons against TM. We used batches of 128 packets to amortize the synchronization overheads. Thus, these algorithms achieve almost linear scaling and the highest possible throughput with perfect load-balancing between the cores.

Single-core implementation. We used a single core to measure the performance of NuevoMatch with the early termination optimization. For NM, we discarded iSets with coverage below 25%.

5.1.1 Packet traces and rule-sets. For evaluating each classifier, we generated traces with 700K packets. We processed each trace 6 times, using the first five as warmup and measuring the last. We report the average of 15 measurements.

Uniform traffic. We generate traces that access all matching rules uniformly to evaluate the worst-case memory access pattern.

Skewed traffic. For each rule-set we generate traces that follow Zipf distribution with four different skew parameters, according to the amount of traffic that accounts for the 3% most frequent flows (e.g., 80% of the traffic accounts for the 3% most frequent flows). This is representative of real traffic, as has been shown in previous works [13, 33].

Additionally, we use a real CAIDA trace from the Equinix datacenter in Chicago [2]. As CAIDA does not publish the rules used to process the packets, we modify the packet headers in the trace to match each evaluated rule-set as follows. For each rule, we generate one matching five-tuple. Then, for each packet in CAIDA, we replace the original five-tuple with a random five-tuple generated from the rule-set, while maintaining a consistent mapping between the original and the generated one. Note that the rule-set access locality of the generated trace is the same or as high as the original trace.

ClassBench rules. ClassBench [39] is a standard benchmark broadly used for evaluating packet classification algorithms [3, 16, 21, 22, 28, 41, 44]. It creates rule-sets that correspond to the rule distribution of three different applications: Access Control List (ACL), Firewall (FW), and IP Chain (IPC). We created rule-sets of sizes 500K, 100K, 10K, and 1K, each with 12 distinct applications, all with 5-field rules: source and destination IP, source and destination port, and protocol.

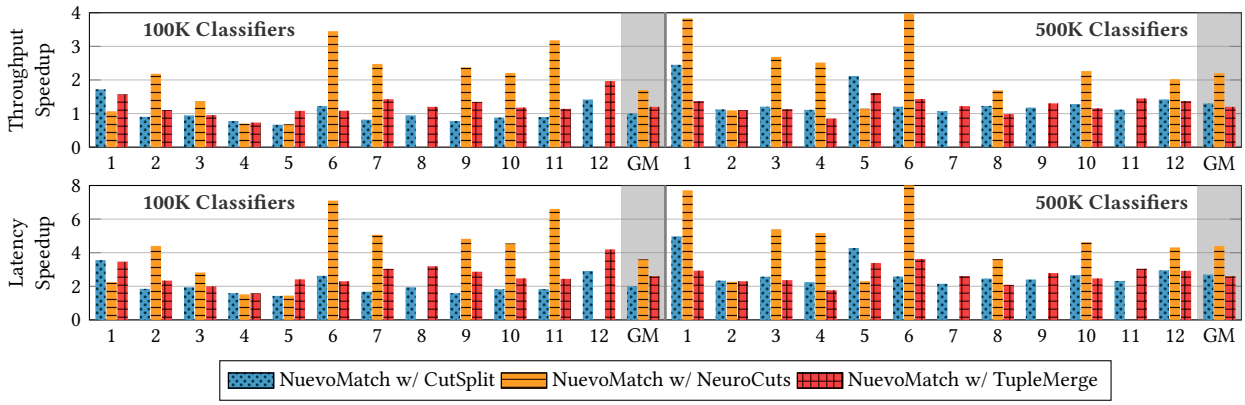


Figure 8: ClassBench: NuevoMatch vs. CutSplit, NeuroCuts, and TupleMerge, using two CPU cores. (See rule-set in the Appendix.)

Real-world rules. We used the Stanford Backbone dataset which contains a large enterprise network configuration [46]. There are four IP forwarding rule-sets with roughly 180K single-field rules each (i.e., destination IP address).

5.2 End-to-end performance

For fair comparison, NuevoMatch used *the same* algorithm for both the remainder classifier and the baseline. For example, we evaluated the speedup produced by NuevoMatch over CS while also using CS to index the remainder set.

We present the results for random packet traces, followed by skewed and CAIDA traces.

Large rule-sets: ClassBench: multi-core. Figure 8 shows that, in the largest rule-sets (500K), the parallel implementation of NuevoMatch achieves a geometric mean factor of 2.7×, 4.4×, and 2.6× lower latency and 1.3×, 2.2×, and 1.2× higher throughput over CS, NC, and TM, respectively. For the classifiers with 100K rules, the gains are lower but still significant: 2.0×, 3.6×, and 2.6× lower latency and 1.0×, 1.7×, and 1.2× higher throughput over CS, NC, and TM, respectively. The performance varies among rule-sets, i.e., some classifiers are up to 1.8× faster than CS for 100K inputs.

Large rule-sets: ClassBench: single core. Figure 9 shows the throughput speedup of NM compared to CS, NC, and TM. For 500K rule-sets, NuevoMatch achieves a geometric mean improvement of 2.4×, 2.6×, and 1.6× in throughput compared to CS, NC, and TM, respectively. For the single core execution the latency and the throughput speedups are the same.

Large rule-sets: Stanford backbone: multi-core. Figure 10 shows the speedup of NM over TM for the real-world Stanford backbone dataset with 4 rule-sets. NM achieves 3.5× higher throughput and 7.5× lower latency over TM on all four rule-sets.

Small rule-sets: multi-core. For rule-sets with 1K and 10K rules, NuevoMatch results in the same or lower throughput, and 2.2× and 1.9× on average better latency compared to CS and TM. The lower speedup is expected, as both CS and TM fit into L1 (§5.2.1), so NM does not benefit from reduced memory footprint, while adding computational overheads. See Appendix for the detailed chart.

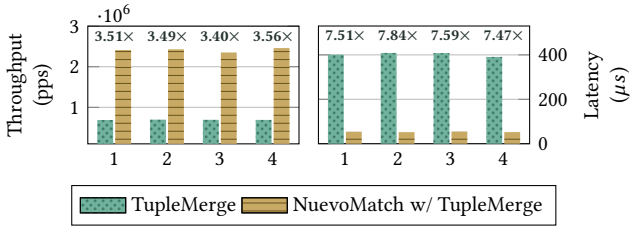


Figure 10: End-to-end performance on real Stanford backbone data sets.

The CS results are averaged over three rule-sets of 1K and six rule-sets for 10K. In the remaining rule-sets, NuevoMatch did not produce large-enough iSets to accelerate the remainder. Note, however, that it promptly identifies the rule-sets expected to be slow and falls back to the original classifier.

The source of speedups. The ability to compress the rule-set to fit into faster memory while retaining fast lookup is the key factor underlying the performance benefits of NuevoMatch. To illustrate it, we take a closer look at the performance. We evaluate TM with and without NM acceleration as a function of its memory footprint on ClassBench-generated 1K,10K,100K and 500K rule-sets for one application (ACL).

Figure 11 shows that the performance of TM degrades as the number of rules grows, causing the hash tables to spill out of L1 and L2 caches. NM compresses a large part of the rule-set (see coverage annotations), thereby making the remainder index small enough to fit in the L1 cache, and gaining back the throughput equivalent to TM's on small rule-sets.

ClassBench: Skewed traffic. Figure 12 shows the evaluation of the early termination implementation on skewed packet traces. We report the throughput speedup of NM compared to CS and TM; the results for NC are similar to those of CS.

We perform 6000 experiments using 25 traces per rule-set: five traces per Zipf distribution plus five modified CAIDA traces. We evaluate over twelve 500K rule-sets and report the geometric mean. Additionally, we evaluate CAIDA traces in two settings. First, the

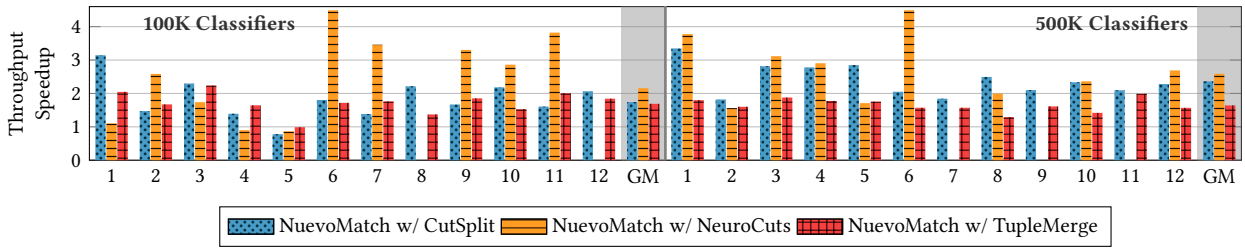


Figure 9: ClassBench: NuevoMatch vs. CutSplit, NeuroCuts, and TupleMerge, using a single CPU core.

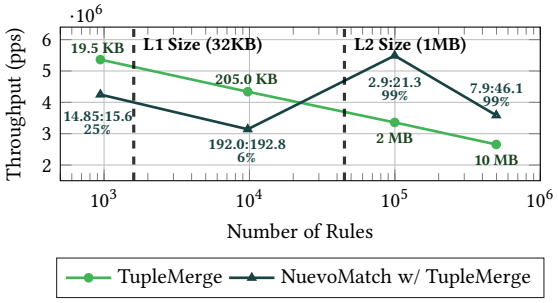


Figure 11: Throughput vs. number of rules for TupleMerge and NuevoMatch. Annotations are coverage (%) and index memory size in KB (remainder : total).

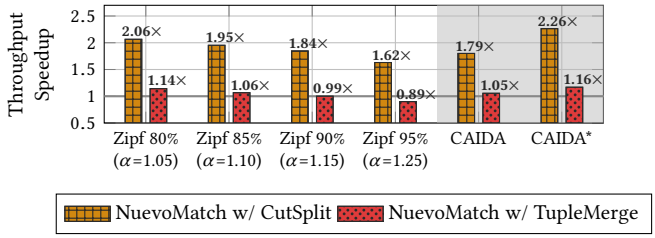


Figure 12: ClassBench: NuevoMatch vs. CutSplit and TupleMerge with skewed traffic.

classifier runs with access to the entire 16MB of the L3 cache (denoted as CAIDA). Second, the classifier use of L3 is restricted to 1.5MB via Intel's Cache Allocation Technology, emulating multi-tenant setting (denoted as CAIDA*).

NuevoMatch is significantly faster than CS, but its benefits over TM diminish for workloads with higher skews. Yet, the speedups are more pronounced under smaller L3 allocation.

Overall, we observe lower speedups for the skewed traffic than for the random trace. This is not surprising, as skewed traces induce a higher cache hit rate for all the methods, which in turn reduces the performance gains of NM over both CS and TM, similar to the case of small rule-sets. Nevertheless, it is worth noting that classification algorithms are usually applied alongside caching mechanisms that catch the packets' temporal locality. For instance, Open vSwitch applies caching for most frequently used rules. It invokes Tuple Space Search upon cache misses [30]. Therefore, if NuevoMatch is applied at this stage, we expect it to yield the performance gains equivalent to those reported for unskewed workloads. Open vSwitch integration is the goal of our ongoing work.

5.2.1 Memory footprint comparison. Figure 13 compares the memory footprint of the classifiers without and with NuevoMatch (the two right-most bars in each bar cluster). We use the same number of iSets as in the end-to-end experiments. Note that a smaller footprint alone does not necessarily lead to higher performance if more iSets are used. Therefore, the results should be considered in conjunction with the end-to-end performance.

The memory footprint includes only the index data structures but not the rules themselves. In particular, the memory footprint for NuevoMatch includes both the RQ-RMI models and the remainder classifier. Each bar is the average of all the 12 application rule-sets of the same size.

For NM we show both the remainder index size (middle bar) and the total RQ-RMI size (right-most bar). Note that due to the logarithmic scale of the Y axis, the actual ratio bewteen the two is much higher than it might seem. For example, the remainder for 10K TM is almost 100× the size of the RQ-RMI. Note also that since we run NM on two cores, both RQ-RMI and the remainder classifier use their own CPU caches.

Overall, NuevoMatch enables dramatic compression of the memory footprint, in particular for 500K rule-sets, with 4.9×, 8×, and 82× on average over CS, NC and TM respectively.

The graph explains well the end-to-end performance results. For 1K rule-sets, the original classifiers fit into the L1 cache, so NM is not effective. For 10K sets, even though the remainder index fits in L1, the ratio between L1 and L2 performance is insufficient to cover the RQ-RMI overheads. For 100K, the situation is similar for CS; however, for NC, the remainder fits in L1, whereas the original NC spills to L3. For TM, the remainder is already in L2, yielding a lower overall speedup compared to NC. Last, for 500K rule-sets, all the original classifiers spill to L3, whereas the remainder fits well in L2, yielding clear performance improvements.

Performance under L3 cache contention. The small memory footprint of NM plays an important role even when the rule-index fits in the L3 cache (16MB in our machine). L3 is shared among all the CPU cores; therefore, cache contention is not rare, in particular in data centers. NM reduces the effects of L3 cache contention on packet classification performance. In the experiment we use the 500K rule-set (1) and compare the performance of CS and NM (with CS) while limiting the L3 to 1.5MB. CS loses half of its performance, whereas NM slows down by 30%, increasing the original speedup.

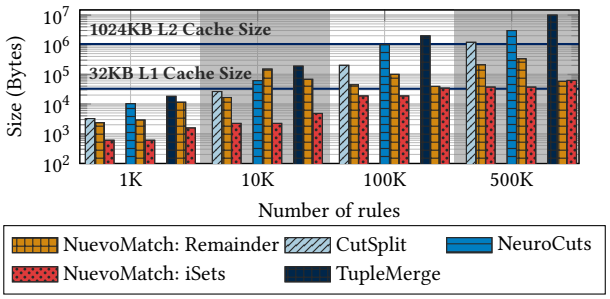


Figure 13: Memory size for CutSplit, NeuroCuts, TupleMerge vs. NuevoMatch with them indexing the remainder. Each bar is a geometric mean of 12 applications.

Table 2: iSet coverage.

	1 iSet	2 iSets	3 iSets	4 iSets
1K	20.2 ± 18.6	28.9 ± 22.3	34.6 ± 25.6	38.7 ± 27.2
10K	45.1 ± 31.6	59.6 ± 38.9	62.6 ± 37.1	65.1 ± 35.7
100K	80.0 ± 14.5	96.5 ± 8.3	98.1 ± 4.8	98.8 ± 2.7
500K	84.2 ± 10.5	98.8 ± 1.5	99.4 ± 0.6	99.7 ± 0.2
183,376	57.8	91.6	96.5	98.2

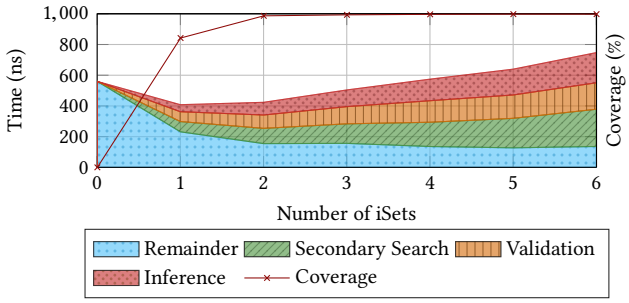


Figure 14: Coverage and execution time breakdown of NuevoMatch vs. varying number of iSets.

5.3 Performance analysis

5.3.1 iSet coverage. Table 2 shows the cumulative coverage achieved with up to 4 iSets averaged over 12 rule-sets (ClassBench) of the same size. The coverage of smaller rule-sets is worse on average, but improves with the size of the rule-set.

The last row shows a representative result for the Stanford backbone rule-set (the other three differ within 1%). Two iSets are enough to achieve 90% coverage and three are needed for 95%. This data set differs from ClassBench in that it contains only one field, providing fewer opportunities for iSet partitioning.

5.3.2 Impact of the number of iSets. We seek to understand the tradeoff between the iSet coverage of the rule-set and the computational overheads of adding more RQ-RMI. All computations were performed on a single core to obtain the latency breakdown. We use CS for indexing the remainder.

Table 3: Throughput and a single iSet coverage vs. the fraction of low-diversity rules in a 500K rule-set.

% Low diversity rules	% Coverage	Speedup (throughput)
70%	25%	1.07×
50%	50%	1.14×
30%	70%	1.60×

Figure 14 shows the geometric mean of the coverage and the runtime breakdown over 12 rule-sets of 500K. The breakdown includes the runtime of the remainder classifier, validation, secondary search, and RQ-RMI inference. Zero iSets implies that CS was run alone. Adding more iSets shows diminishing returns because of their compute overhead, which is not compensated by the remainder runtime improvements because the coverage is already saturated to almost 100%. Using one or two iSets shows the best trade-off. NC shows similar results.

TM behaved differently (not shown). TM occupies much more memory than CS; therefore, using more iSets to achieve higher coverage allowed us to further speed up the remainder by fitting it into an upper level cache. Thus, 4 iSets showed the best configuration.

We note that the runtime is split nearly equally between model inference and validation (which are compute-bound parts), and the secondary search and the remainder computations (which are memory-bound). We expect the compute performance of future processors to scale better than their cache capacity and memory access latency. Therefore, we believe NM will provide better scaling than memory-bound state-of-the-art classifiers.

5.3.3 Partitioning effectiveness. We seek to understand how low diversity rule-sets affect NuevoMatch. To analyze that, we synthetically generated a large rule-set as a Cartesian product of a small number of values per field (no ranges). We blended them into a 500K ClassBench rule-set, replacing randomly selected rules with those from the Cartesian product, while keeping the total number of rules the same.

Table 3 shows the coverage and the speedup over TM on the resulting mixed rule-sets for different fractions of low-diversity rules. The partitioning algorithm successfully segregates the low-diversity rules the best, achieving the coverage inversely proportional to their fraction in the rule-set. Note that NuevoMatch becomes effective when it offloads the processing of about 25% of the rules.

5.3.4 Training time and secondary search range. RQ-RMIs are trained to minimize the prediction error bound to achieve a small secondary search distance. Recall that a secondary search involves a binary search within the error bound, where each rule is validated to match all the fields.

The tradeoff between training time and secondary search performance is not trivial. A larger search distance enables faster training but slows down the secondary search. A smaller search distance results in a faster search but slows down the training. In extreme cases, the training does not converge, since a higher precision might require larger submodels. However, increasing the size of the submodels leads to a larger memory footprint and longer computations.

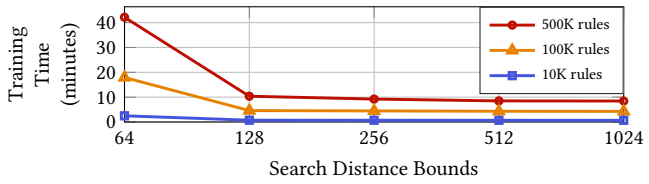


Figure 15: RQ-RMI training time in minutes vs. maximum search range bound.

Figure 15 shows the average end-to-end training time in minutes of 500 models as a function of the secondary search distance and the rule-set size. The measurements include all training iterations as described in §3.5. As mentioned (§4), our training implementation can be dramatically accelerated, so the results here indicate the general trend.

Training with the bound of 64 is expensive, but is it really necessary? To answer, we evaluate the performance impact of the search distance on the secondary search time. We measure 40ns for retrieving a rule with a precise prediction (no search). For 64, 128 and 256 distances the search time varies between 75 to 80ns thanks to the binary search. Last, it turns out that the *actual* search distance from the predicted index is often much smaller than the worst-case one enforced in training. Our analysis shows that in practice, training with a relatively large bound of 128 leads to 80% of the lookups with a search distance of 64, and 60% with 32.

We conclude that training with larger bounds is likely to have a minor effect on the end-to-end performance, but significantly accelerate training. This property is important to support more frequent retraining and faster updates (§3.9).

5.3.5 Performance with more fields. Adding fields to an existing classifier will not harm its coverage, so it will not affect the RQ-RMI performance. Nonetheless, more fields will increase validation time.

Unfortunately, we did not find public rule-sets that have a large number of fields. Thus, we ran a microbenchmark by increasing the number of fields and measuring the validation stage performance. As expected, we observed almost linear growth in the validation time, from 25ns for one field to 180ns for 40 fields.

6 RELATED WORK

Hardware-based classifiers. Hardware-based solutions for classification such as TCAMs and FPGAs achieve a very high throughput [6, 35]. Consequently, many software algorithms take advantage of them, further improving classification performance [13, 20, 23, 24, 28, 32, 37]. Our work is complementary, but can be used to improve scaling of these solutions. For example, if the original classifier required large TCAMs, the remainder set would fit a much smaller TCAM.

GPUs for classification. Accelerating classification on GPUs was suggested by numerous works. PacketShader [10] uses GPU for packet forwarding and provides integration with Open vSwitch. However, packet forwarding is a single-dimensional problem, so it is easier than multi-field classification [9]. Varvello et al. [42] implemented various packet classification algorithms in GPUs, including linear search, Tuple Space Search, and bloom search. Nonetheless, these techniques suffer from poor scalability for large classifiers with wildcard rules, which NuevoMatch aims to alleviate.

ML techniques for networking. Recent works suggest using ML techniques for solving networking problems, such as TCP congestion control [4, 12, 45], resource management [25], quality of experience in video streaming [26, 43], routing [40], and decision tree optimization for packet classification [22]. NuevoMatch is different in that it uses an ML technique for building space-efficient representations of the rules that fit in the CPU cache.

7 CONCLUSIONS

We have presented NuevoMatch, the first packet classification technique that uses *Range-Query RMI* machine learning model for *accelerating* packet classification. We have shown an efficient way of training RQ-RMI models, making them learn the matching ranges of large rule-sets, via sampling and analytical error bound computations. We demonstrated the application of RQ-RMI to multi-field packet classification using rule-set partitioning. We evaluated NuevoMatch on synthetic and real-world rule-sets and confirmed its benefits for large rule-sets over state-of-the-art techniques.

NuevoMatch introduces a new point in the design space of packet classification algorithms and opens up new ways to scale it on commodity processors. We believe that its compute-bound nature and the use of neural networks will enable further scaling with future CPU generations, which will feature powerful compute capabilities targeting faster execution of neural network-related computations.

8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers of SIGCOMM'20 and our shepherd Minlan Yu for their helpful comments and feedback. We would also like to thank Isaac Keslassy and Leonid Ryzhyk for their feedback on the early draft of the paper.

This work was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate, by the Alon fellowship and by the Taub Family Foundation. We gratefully acknowledge support from Israel Science Foundation (Grant 1027/18) and Israeli Innovation Authority.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX OSDI*.

[2] CAIDA. [n.d.]. *The CAIDA UCSD Anonymized Internet Traces 2019.* Retrieved June 15, 2020 from http://www.caida.org/data/passive/passive_dataset.xml

[3] James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourtchenko. 2019. TupleMerge: Fast Software Packet Processing for online Packet Classification. *IEEE/ACM Transactions on Networking (TON)* 27, 4 (2019), 1417–1431.

[4] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *USENIX NSDI*.

[5] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *USENIX NSDI*.

[6] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI*.

[7] Pankaj Gupta and Nick McKeown. 1999. Packet Classification on Multiple Fields. In *ACM SIGCOMM*.

[8] Pankaj Gupta and Nick McKeown. 2000. Classifying Packets with Hierarchical Intelligent Cuttings. *IEEE Micro* 20, 1 (2000), 34–41.

[9] Pankaj Gupta and Nick McKeown. 2001. Algorithms for Packet Classification. *IEEE Network* 15, 2 (2001), 24–32.

[10] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated software router. In *ACM SIGCOMM*.

[11] Intel. 2019. *Intel Nervana Neural Network Processors.* Retrieved September 25, 2019 from https://www.intel.ai/nervana-nnp/

[12] Nathan Jay, Noga H. Rotman, Philip Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2018. Internet Congestion Control via Deep Reinforcement Learning. *arXiv preprint arXiv:1810.03259* (2018).

[13] Naga Praveen Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. 2016. CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks. In *ACM SOSR*.

[14] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980* (2014).

[15] Jon M. Kleinberg and Éva Tardos. 2006. *Algorithm Design.* Addison-Wesley, 116–125.

[16] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. 2014. SAX-PAC (Scalable and expressive packet classification). In *ACM SIGCOMM*.

[17] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System.

[18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *ACM SIGMOD*.

[19] Habana Labs. 2019. *Habana AI Processors.* Retrieved September 25, 2019 from https://habana.ai/product

[20] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. 2005. Algorithms for Advanced Packet Classification with Ternary CAMs. In *ACM SIGCOMM*.

[21] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. 2018. CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification. In *IEEE INFOCOM*.

[22] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural Packet Classification. In *ACM SIGCOMM*.

[23] Alex X Liu, Chad R Meiners, and Yun Zhou. 2008. All-Match Based Complete Redundancy Removal for Packet Classifiers in TCAMs. In *IEEE INFOCOM*.

[24] Yadi Ma and Suman Banerjee. 2012. A Smart Pre-classifier to Reduce Power Consumption of TCAMs for Multi-dimensional Packet Classification. In *ACM SIGCOMM*.

[25] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *ACM SIGCOMM HotNets Workshop*.

[26] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *ACM SIGCOMM*.

[27] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR* 38, 2 (2008), 69–74.

[28] Nina Narodytska, Leonid Ryzhyk, Igor Ganichev, and Soner Sevinc. 2019. BDD-Based Algorithms for Packet Classification. In *Formal Methods in Computer Aided Design FMCAD*.

[29] Nvidia. 2019. *Nvidia Deep Learning Inference Platform.* Retrieved September 25, 2019 from https://www.nvidia.com/en-us/deep-learning-ai/solutions/inference-platform/

[30] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *USENIX NSDI*.

[31] Alon Rashelbach. 2020. *NeuvoMatch source code.* Retrieved June 21, 2020 from https://github.com/acsl-technion/nuevomatch

[32] Ori Rottenstreich and János Tapolcai. 2015. Lossy Compression of Packet Classifiers. In *ACM/IEEE ANCS*.

[33] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. 2012. Leveraging Zipf's law for traffic offloading. *Computer Communication Review* 42, 1 (2012), 16–22.

[34] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet Classification Using Multidimensional Cutting. In *ACM SIGCOMM*.

[35] Ed Spitznagel, David E Taylor, and Jonathan S Turner. 2003. Packet classification using extended TCAMs. In *IEEE ICNP*.

[36] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet Classification Using Tuple Space Search. In *ACM SIGCOMM*.

[37] David E Taylor. 2005. Survey and Taxonomy of Packet Classification Techniques. *ACM Computing Surveys (CSUR)* 37, 3 (2005), 238–275.

[38] David E. Taylor and Jonathan S. Turner. 2005. Scalable packet classification using distributed crossproducing of field labels. In *IEEE INFOCOM*.

[39] David E Taylor and Jonathan S Turner. 2007. Classbench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking (TON)* 15, 3 (2007), 499–511.

[40] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. 2017. Learning to Route with Deep RL. In *NIPS Deep Reinforcement Learning Symposium*.

[41] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2010. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *ACM SIGCOMM*.

[42] Matteo Varvello, Rafael Laufer, Feixiong Zhang, and T. V. Lakshman. 2016. Multilayer Packet Classification with Graphics Processing Units. *IEEE/ACM Transactions on Networking (TON)* 24, 5 (2016), 2728–2741.

[43] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. 2018. Neural Adaptive Content-aware Internet Video Delivery. In *USENIX OSDI*.

[44] Sorrachai Yingchareonthawornchai, James Daly, Alex X. Liu, and Eric Torng. 2018. A Sorted-Partitioning Approach to Fast and Scalable Dynamic Packet Classification. *IEEE/ACM Transactions on Networking (TON)* 26, 4 (2018), 1907–1920.

[45] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. 2015. Adaptive Congestion Control for Unpredictable Cellular Networks. In *ACM SIGCOMM*.

[46] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic Test Packet Generation. In *ACM CoNEXT*.

Appendices are supporting material that has not been peer-reviewed.

A RQ-RMI CORRECTNESS

A.1 Responsibility of a submodel

Denote the input domain of an RQ-RMI model as $D \subset \mathbb{R}$ and its number of stages as n.

Theorem A.1 (Responsibility Theorem). *Let s_i be a trained stage such that $i < n-1$. The responsibilities of submodels in s_{i+1} can be calculated by evaluating a finite set of inputs over the stage s_i.*

The intuition behind Theorem A.1 is based on Corollary 3.2, namely that submodels output piecewise linear functions. Proving it requires some additional definitions.

Definition A.2 (Stage Output). The output of stage s_i is defined for $x \in D$ as $S_i(x) = M_{i,f_i(x)}(x)$ where $f_i(x)$ is the index of the submodel in s_i that is responsible for input x, and defined as

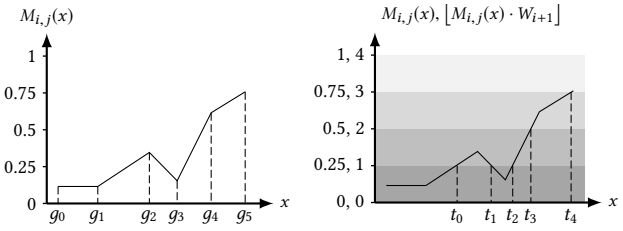
$$f_i(x) = \begin{cases} 0 & i = 0 \\ \lfloor S_{i-1}(x) \cdot W_i \rfloor & i = \{1, 2, ..., n-1\} \end{cases}$$

Definition A.3 (Submodel Responsibility). The responsibility of a submodel $m_{i,j}$ is defined as

$$R_{i,j} = \begin{cases} D & i = 0 \\ \{x \mid f_i(x) = j\} & i = \{1, 2, ..., n-1\} \end{cases}$$

Note that the responsibilities of every two submodels in the same stage are disjoint.

Definition A.4 (Left and Right Slopes). For a range R, if points $\min_{x \in R} x$ or $\max_{x \in R} x$ are defined, we refer to them as the boundaries of the range. For all other points, we refer to as internal points of the range. For a piecewise linear function defined over some



(a) Trigger Inputs $G_{i,j}$ **(b) Transition Inputs $T_{i,j}$**

Figure 16: Illustration of the trigger inputs $(g_0, ..., g_5)$ and transition inputs $(t_0, ..., t_4)$ for graph $M_{i,j}(x)$ of submodel $m_{i,j}$. Note that W_{i+1}, namely the number of submodels in stage $i+1$, affects the transition inputs of $m_{i,j}$ and equals 4.

range R, for every internal point $x \in R$, there exists $\delta > 0$ such that the function is linear in each of $(x-\delta, x), (x, x+\delta)$. Accordingly, we can refer to the left slope and the right slope of a point, defined as those of the two linear functions.

Definition A.5 (Trigger Inputs). We say that an input $g \in D$ is a *trigger input* of a submodel $m_{i,j}$ if one of the following holds: *(i)* g is a boundary point of D (namely, $g = \min_{y \in D} y$ or $g = \max_{y \in D} y$). *(ii)* g is an internal point of D and the left and right slopes of $M_{i,j}(g)$ differ.

Definition A.6 (Transition Inputs). We say that an input $t \in D$ is a *transition input* of a submodel $m_{i,j}$ if it changes submodel selection in the following stage. Formally, there exists $\epsilon > 0$ such that for all $0 < \delta < \epsilon$:

$$\left\lfloor M_{i,j}(t-\delta) \cdot W_{i+1} \right\rfloor \neq \left\lfloor M_{i,j}(t+\delta) \cdot W_{i+1} \right\rfloor$$

Definition A.7 (The function $B_i(x)$). We define the function B_i for $i \in \{0, 1, ..., n-1\}$. B_i is a staircase function of values $[0, W_{i+1} - 1]$, and defined as $B_i(x) = \lfloor x \cdot W_{i+1} \rfloor$ for $x \in [0, 1)$.

For a submodel $m_{i,j}$, we term the set of its trigger inputs as $G_{i,j}$ and the set of its transition inputs as $T_{i,j}$. See Figure 16 for illustration. From submodel definition and Corollary 3.2, we can tell that a submodel's ReLU operations determine its trigger inputs. Consequently, any set of trigger inputs is finite and can be calculated using a few linear equations. Nonetheless, calculating the transition inputs of a submodel is not straightforward. We show a fast and efficient way for doing so in the following lemma:

Lemma A.8. *Let $m_{i,j}$ be an RQ-RMI submodel, and $a < b \in G_{i,j}$ two adjacent trigger inputs of $m_{i,j}$. Then the set $S = [a, b] \cap T_{i,j}$ is finite and can be calculated using the inputs a and b alone.*

Proof. We divide the construction of S to two subsets $S = S_0 \cup S_1$. First we handle S_0. For each $x \in \{a, b\}$, $x \in S_0$ if and only if there exists $\epsilon > 0$ such that for all $0 < \delta < \epsilon$:

$$B_i\big(M_{i,j}(x-\delta)\big) \neq B_i\big(M_{i,j}(x+\delta)\big)$$

Now to S_1. Without loss of generality, $M_{i,j}(a) \leq M_{i,j}(b)$. From Corollary 3.2 and Definition A.5, $M_{i,j}$ is linear in $[a, b]$. If $B_i(M_{i,j}(a)) = B_i(M_{i,j}(b))$, then $S_1 = \emptyset$. Otherwise, $M_{i,j}(a) \neq M_{i,j}(b)$. $B_i(x)$ outputs discrete values between $B_i(M_{i,j}(a))$ and $B_i(M_{i,j}(b))$ for all $x \in (a, b)$. Denote this finite set of discrete values as M. For any $y \in M$ there exists a value $d \in (a, b]$ such that $M_{i,j}(d) \cdot W_{i+1} = y$. By the linearity of $M_{i,j}$ in $[a, b]$:

$$d = \left(\frac{y}{W_{i+1}} - M_{i,j}(a)\right) \cdot \frac{b-a}{M_{i,j}(b) - M_{i,j}(a)} + a$$

We construct S_1 as follows:

$$S_1 = \left\{ \left(\frac{y}{W_{i+1}} - M_{i,j}(a)\right) \cdot \frac{b-a}{M_{i,j}(b) - M_{i,j}(a)} + a \;\middle|\; \forall y \in M \right\}$$

□

Corollary A.9. *The set of transition inputs $T_{i,j}$ can be calculated using $G_{i,j}$ and its size is bounded such that $|T_{i,j}| \leq W_{i+1} \cdot |G_{i,j}|$.*

Not all transition inputs of all submodels are reachable, as some exist outside of their corresponding submodel's responsibility. Therefore, we define the set of reachable transition inputs of a stage s_i as the *transition set* of a stage:

Definition A.10 (Transition Set). The *transition set* U_i of a stage s_i is an ordered set, defined as:

$$U_i = \{\min(D)\} \cup \{ \bigcup_{j=0}^{W_i - 1} T_{i,j} \cap R_{i,j} \} \cup \{\max(D)\}$$

The proof of Theorem A.1 directly follows from the next two lemmas:

Lemma A.11. *Let s_i, s_{i+1} be two adjacent stages. For any two adjacent values $u_0 < u_1 \in U_i$ there exists a submodel $m_{i+1,j}$ such that $S_{i+1}(x)$ is piecewise linear and equal to $M_{i+1,j}(x)$ for all $x \in (u_0, u_1)$.*

Proof. We show that there exists a submodel $m_{i+1,j}$ such that any $x \in (u_0, u_1)$ satisfies $x \in R_{i+1,j}$, which implies $f_{i+1}(x) = j$ and so $S_{i+1}(x) = M_{i+1,j}(x)$. By Corollary 3.2, S_{i+1} is piecewise linear for all $x \in (u_0, u_1)$.

Let $x < y \in (u_0, u_1)$. Assume by contradiction there exist two submodels m_{i+1,j_0} and m_{i+1,j_1} such that $x \in R_{i+1,j_0}$ and $y \in R_{i+1,j_1}$. From Definition A.3, $f_{i+1}(x) \neq f_{i+1}(y)$ implies $B_i(S_i(x)) \neq B_i(S_i(y))$. Thus, there exists an input $z \in (x, y]$ and $\epsilon > 0$ such that for all $0 < \delta < \epsilon$:

$$B_i\big(S_i(z-\delta)\big) \neq B_i\big(S_i(z+\delta)\big)$$

Since S_i consists of the outputs of submodels in s_i, there exists a submodel $m_{i,k}$ such that $S_i(z) = M_{i,k}(z)$. Therefore, $z \in T_{i,k}$ and $z \in R_{i,k}$, which means $z \in U_i$, in contradiction to definition of u_0 and u_1. □

Lemma A.12. *Let s_i be an RQ-RMI stage such that $i \in \{0, 1, ..., n-2\}$. The function f_{i+1} defined over the space D can be calculated using the inputs U_i over S_i.*

Proof. Let $u_0 < u_1 \in U_i$ be two adjacent values. By Lemma A.11 there exists a submodel $m_{i+1,j}$ such that $S_{i+1}(x) = M_{i,j}(x)$ for all $x \in (u_0, u_1)$. From Definition A.2, $f_{i+1}(x) = j$ for all $x \in (u_0, u_1)$. By calculating $B_i(S_i(u_0))$ and $B_i(S_i(u_1))$, $f_{i+1}(x)$ is known for all $x \in [u_0, u_1]$. Since $\min\{D\} \in U_i$ and $max\{D\} \in U_i$, $f_{i+1}(x)$ is known for all $x \in D$. □

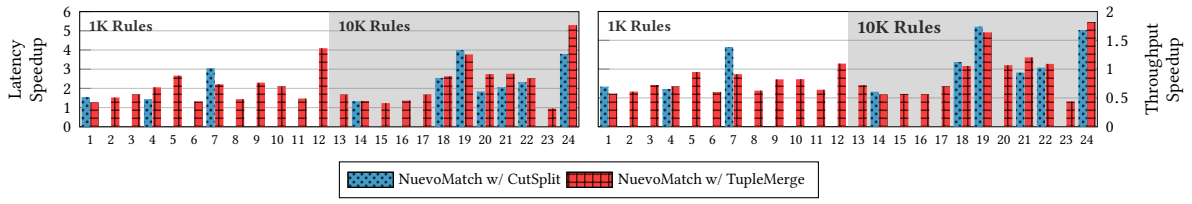


Figure 17: A detailed version of end-to-end performance for *small* rule-sets. Speedup in throughput and latency of NuevoMatch against stand-alone versions of CutSplit and TupleMerge. Classifiers with no valid iSets are not displayed.

A.2 Submodel prediction error

Theorem A.13 (Submodel Prediction Error). *Let s_{n-1} be the last stage of an RQ-RMI model. The maximum prediction error of any submodel in s_{n-1} can be calculated using a finite set of inputs over the stage s_{n-1}.*

The intuition behind Theorem A.13 is to address the set of range-value pairs as an additional, virtual, stage in the model.

Definition A.14 (Range-Value Pair). A range-value pair $\langle r, v\rangle$ is defined such that r is an interval in D and $v \in \{0, 1, 2, \ldots\}$ is unique to that pair.

We term W_n the number of range-value pairs an RQ-RMI model should index. Similar to the definitions for submodels, we extend f_i such that $f_n(x) = \lfloor S_{n-1}(x) \cdot W_n \rfloor$, and say that the responsibility R_p of a pair $p = \langle r, v\rangle$ is the set of inputs $\{x | f_n(x) = v\}$. Consequently, we make the following two observations. First, all inputs in the range $r \setminus R_p$ should have reached p but did not. Second, all inputs in the range $R_p \setminus r$ did reach p but should not.

Definition A.15 (Misclassified Pair Set). Let m be a submodel in s_{n-1} with a responsibility R_m. Denote P_m as the set of all pairs such that a pair $p = \langle r, v\rangle \in P_m$ satisfies $(r \setminus R_p) \cup (R_p \setminus r) \cap R_m \neq \emptyset$. In other words, P_m holds all pairs that were misclassified by m, and termed the *misclassified pair set* of m.

Definition A.16 (Maximum Prediction Error). Let m be a submodel in s_{n-1} with a responsibility R_m and a misclassified pair set P_m. The maximum prediction error of m is defined as:

$$\max \left\{ |f_n(x) - v| \,\middle|\, \langle r, v\rangle \in P_m, x \in R_m \right\}$$

Lemma A.17. *The misclassified pair sets of all submodels in s_{n-1} can be calculated using U_{n-1} over S_{n-1}.*

Proof. Let $q_0 < q_1$ be two adjacent values in U_{n-1}. From Lemma A.11 there exists a single submodel $m_{n-1,j}$, $j \in W_{n-1}$ s.t $S_{n-1}(x) = M_{n-1,j}(x)$ for all $x \in (q_0, q_1)$. Hence, using Corollary 3.2, S_{n-1} is linear in (q_0, q_1). Therefore, the values of S_{n-1} in $[q_0, q_1]$ can be calculated using q_0 and q_1 alone. Consequently, according to the definitions of f_n and the responsibility of a pair, the set of pairs P_j with responsibilities in $[q_0, q_1]$ can also be calculated using q_0 and q_1. Calculating the responsibilities of all pairs is performed by repeating the process for any two adjacent points in U_{n-1}.

At this point, as we know R_p for all $p = \langle r, v\rangle$, calculating the set $(r \setminus R_p) \cup (R_p \setminus r)$ is trivial. Acquiring the responsibility of any submodel in s_{n-1} using Theorem A.1 enables us to calculate its misclassified pair set immediately. □

Proof of Theorem A.13

Proof. Let m be a submodel in s_{n-1} with a responsibility R_m. For simplicity, we address the case where R_m is a continuous range. Extension to the general case is possible by repeating the proof for any continuous range in R_m.

Denote the submodel's finite set of trigger inputs as G_m. Define the set Q as follows:

$$Q = \min R_m \cup (G_m \cap R_m) \cup \max R_m$$

Let $q_0 < q_1$ be two adjacent values in Q. From the definition of trigger inputs, m outputs a linear function in $[q_0, q_1]$. Hence, the set of values $S_0 = \{f_n(x) | x \in [q_0, q_1]\}$ can be calculated using only q_0 and q_1 over S_{n-1}. From Lemma A.17, the misclassified pair set P_m can be calculated using the finite set U_{n-1}. Denote the set

$$\hat{P}_0 = \{\langle r, v\rangle \mid \langle r, v\rangle \in P_m,\ r \cap [q_0, q_1] \neq \emptyset\}$$

Calculating $\max\{s - v | s \in S_0, \langle r, v\rangle \in \hat{P}_0\}$ yields the maximum error of m in $[q_0, q_1]$. Repeating the process for any two adjacent points in Q yields the maximum error of m for all R_m. □

Rule-set names in Figures 8 and 17, by order: ACL1, ACL2, ACL3, ACL4, ACL5, FW1, FW2, FW3, FW4, FW5, IPC1, IPC2.

Table 4: RQ-RMI configurations for different input rule-set sizes.

#Rules	#Stages	Stage Widths
Less than 10^3	2	[1, 4]
10^3 to 10^4	3	[1, 4, 16]
10^4 to 10^5	3	[1, 4, 128]
More than 10^5	3	[1, 8, 256] or [1, 8, 512]