Scaling Open vSwitch with a Computational Cache

Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein, *Technion*

https://www.usenix.org/conference/nsdi22/presentation/rashelbach

This paper is included in the Proceedings of the
19th USENIX Symposium on Networked Systems
Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the
19th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Scaling Open vSwitch with a Computational Cache

Alon Rashelbach, Ori Rottenstreich, Mark Silberstein
Technion

Abstract

Open vSwitch (OVS) is a widely used open-source virtual switch implementation. In this work, we seek to scale up OVS to support hundreds of thousands of OpenFlow rules by accelerating the core component of its data-path - the packet classification mechanism. To do so we use NuevoMatch, a recent algorithm that uses neural network inference to match packets, and promises significant scalability and performance benefits. We overcome the primary algorithmic challenge of the slow rule update rate in the vanilla NuevoMatch, speeding it up by over *three orders of magnitude*. This improvement enables two design options to integrate NuevoMatch with OVS: (1) using it as an extra caching layer in front of OVS's megaflow cache, and (2) using it to completely replace OVS's data-path while performing classification directly on OpenFlow rules, and obviating control-path upcalls. Our comprehensive evaluation on real-world packet traces and ClassBench rules demonstrates the geometric mean speedups of 1.9× and 12.3× for the first and second designs, respectively, for 500K rules, with the latter also supporting up to 60K OpenFlow rule updates/second, by far exceeding the original OVS.

1 Introduction

Open vSwitch (OVS) [22] is one of the most popular software switches used by cloud providers to implement software-defined networks [1, 8, 23]. As part of its main tasks, OVS classifies packets according to a set of match-action tuples, i.e., OpenFlow rules dynamically installed by the network controller. To achieve high throughput, OVS adopts the fast/slow path separation principle: the majority of the packets are classified in the fast *data-path*, which maintains a *megaflow cache* optimized for speedy matching. Upon a miss, OVS invokes the slower upcall into a *control-path*, which populates the megaflow cache with tuples called megaflows.

Unfortunately, OVS suffers from two primary scalability issues. First, the megaflow cache becomes slower as the number of megaflows in it grows. Our experiments (§3) show that with 500K megaflows, OVS is about an order of magnitude slower than with 1K megaflows. Importantly, the cache might hold a large number of megaflows even if the number of the original OpenFlow rules is small. This is because when OVS populates the cache, it transforms the relevant OpenFlow rules into a set of non-overlapping megaflows [22]. As a result, the OpenFlow rules might get fragmented; under certain common traffic patterns, this fragmentation leads to a dramatic increase in the number of megaflows in the cache [3, 4].

The second problem is the performance degradation that occurs when new rules are inserted into OVS by a network controller. We observe (§3) that the throughput might be affected significantly even when adding only a few dozens of new OpenFlow rules at a time. The main reason stems from the need to enforce the non-overlapping property of megaflows, which might cause OVS to remove existing megaflows, leading to slow path upcalls. Clearly, the problem gets worse in systems with frequent rule updates.

In this work, we seek to overcome these OVS limitations. Our key idea is to leverage the recently published algorithm for packet classification, called *NuevoMatch* [26, 27], which was shown to significantly outperform state-of-the-art alternatives when scaling to a large number of OpenFlow rules. NuevoMatch uses shallow neural networks comprising a *Range-Query Recursive Model Index* (RQ-RMI) to learn the distribution of the rules. The rule lookup is translated into neural-network inference that replaces the traditional index data structure traversal. Upon an update, new rules are first added to a slow-path *remainder* classifier, and the model is periodically retrained to incorporate them in the fast path. Thus, the RQ-RMI model serves as a *computational cache* for the remainder, while retraining the model is equivalent to filling that cache. The scalability of NuevoMatch follows from its small memory footprint and efficient use of CPU hardware, which together enable fast execution on modern CPUs [26].

However, our initial attempts to integrate OVS and NuevoMatch revealed one critical limitation of the original algorithm: its inability to accommodate fast updates. When rules are modified, NuevoMatch must *retrain the RQ-RMI model*

from scratch on the updated rule-set, in order to reach its full performance potential. Unfortunately, RQ-RMI training time is too long and cannot support the required update rate, particularly with a large number of OpenFlow rules as targeted by our work. Our analysis (§3) shows that the NuevoMatch training rate is *orders of magnitude* slower than the one necessary to achieve its promised performance benefits.

We tackle this challenge by introducing *NuevoMatchUP* which extends the original NuevoMatch training algorithm and improves the training rate by over *three orders of magnitude*. Thus, it requires only a few milliseconds to train tens of thousands of rules, and about one second for 500K rules, thereby paving the way to the practical integration of computational cache into OVS.

We consider two design options for integrating NuevoMatchUP with OVS. The first design, *OVS with computational cache (*OVS-CCACHE*)*, targets the scalability of the megaflow cache by accelerating it with NuevoMatchUP. OVS-CCACHE achieves higher throughput than the original design, but unfortunately inherits the low rule update performance. To support fast updates, we introduce *OVS with computational flows (*OVS-CFLOWS*)*, which leverages the power of NuevoMatchUP to efficiently match complex OpenFlow rules and obviates the need for the megaflow cache and fast-slow path separation of the original OVS. This change eliminates the the key bottleneck that restricts the rule update rates in the original OVS.

We comprehensively evaluate OVS-CCACHE and OVS-CFLOWS using real-world CAIDA [2] and MAWI [37] traces, and the standard ClassBench-generated rule-sets [32]. OVS-CCACHE improves the megaflow cache performance, achieving the end-to-end geometric mean speedups of 1.5×, and 1.9× for 100K, and 500K OpenFlow rules, respectively.

OVS-CFLOWS sidesteps the control-path limitations and is thus significantly faster, with the end-to-end geometric mean speedups of 2.6×, 8.5×, and 12.3× for 1K, 100K, and 500K OpenFlow rules, respectively. Moreover, OVS-CFLOWS handles more than 60K OpenFlow rule updates/second.

These results demonstrate the first practical use of RQ-RMI models in a production packet processing system, and show their ability to improve throughput and scalability.

2 Background

We explain the relevant details about the operation of Open vSwitch (OVS) [22, 23], and describe the NuevoMatch algorithm [26] for packet classification.

2.1 Open vSwitch

Open vSwitch (OVS) is a popular open-source virtual switch that supports industry standard OpenFlow protocols. OVS determines which action to apply on each packet according to the OpenFlow rules installed by the network controller.

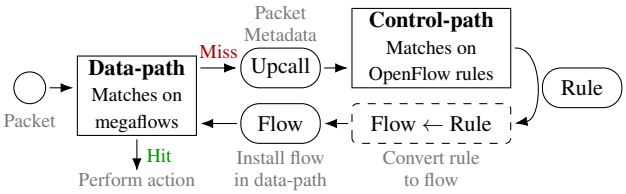


Figure 1: Fast/slow path separation in OVS.

This task is known as *packet classification*, and has been extensively studied [6, 10, 18, 19, 26, 28, 30, 35, 39].

Matching a rule. In its simplest form, a *rule* is a boolean predicate parametrized by one or more fields in the packet header (e.g., IP address, IP protocol). If a predicate is true for a given packet (the rule *matches*), an *action* associated with the rule is invoked to process the packet. An action is an operation to apply to the packet (e.g., forward to port or drop). A packet may match several *overlapping* rules, but only the one with the highest priority is selected.

Control-/data- path. OVS is split into data- (fast) and control- (slow) paths (Figure 1). All OpenFlow rules are installed and maintained in the control-path. The data-path, on the other hand, uses a *megaflow cache* to achieve high processing rates.

The megaflow cache holds non-overlapping rules called *megaflows*, generated by the control-path from the installed OpenFlow rules. Specifically, whenever the data-path encounters a packet that does not match any previously installed megaflow, it performs an *upcall* to the control-path, which in turn finds the relevant OpenFlow rule and converts it into a megaflow. Future packets with the same header fields will not require upcalls unless the megaflow is removed. OVS ensures the correctness of the matching process with the megaflow cache, terminating lookup after a hit in it. To achieve that, OVS tracks all modifications to the OpenFlow rules in the control-path. In particular, it might need to invalidate previously installed megaflows when new OpenFlow rules are added. As we show in §3, these operations might significantly affect OVS's performance.

In addition to the megaflow cache, OVS often activates a short-term exact-match cache (EMC) in front of it. The EMC can be helpful with high-locality traffic.

Megaflow cache implementation. The megaflow cache uses the *Tuple Space Search* (TSS) [30] algorithm for packet classification, as follows. Megaflows with the same mask m are stored in the same hash table H_m, with masked flow keys as entries. Given a packet header h, the megaflow cache iterates over all hash tables to find an entry that matches h (i.e., the masked header equals to the masked key). The lookup latency increases linearly with the number of hash tables traversed.

OVS's data-path can run either in the user-space using DPDK [25], or as a dedicated Kernel module. In this paper we use the DPDK version for its higher performance [34].

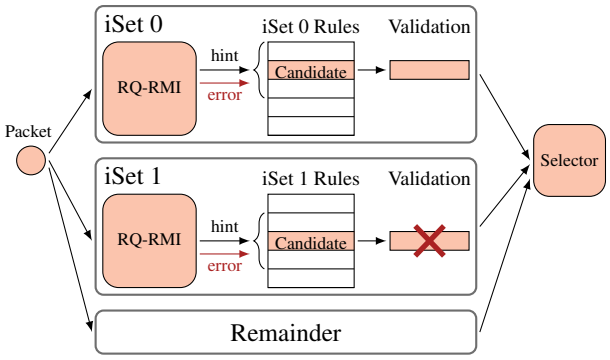


Figure 2: NuevoMatch algorithm [26], RQ-RMI inference provides hints to find the matching rule (details in §2.2).

Size Category	SC 0	SC 1	SC 2	SC 3
# Input Rules	$< 10^3$	10^3 - 10^4	10^4-10^5	$> 10^5$
# Neural Nets	5	21	133	265 or 521

Table 1: RQ-RMI model size (number of neural nets) for different number of rules to index (values taken from [26]).

2.2 NuevoMatch Classification Algorithm

NuevoMatch (NM) is a new class of packet classification algorithms that leverage neural nets to scale to many rules [26].

Figure 2 presents the main components of the algorithm. NM partitions a given set of rules into several independent subsets (iSets), such that each iSet s has a header field h_s in which its rules do not overlap. The fraction of an iSet's rules out of all rules is called the *iSet's coverage*. In practice, two iSets are often sufficient to cover more than 90% of the rules for large enough rule-sets [26]. Rules that do not fit in any of the iSets are handled by a *remainder classifier*, which can be implemented by any other packet classification technique.

For each iSet s, NM trains a hierarchical model called *Range-Query Recursive Model Index* (RQ-RMI) which consists of multiple shallow neural-nets. RQ-RMI learns the distribution of ranges represented by the rules and outputs the *estimated* index of the matching rule within an array. At the inference time, this estimation is used as a starting index to *search* for the matching rule within the array. Crucially, the RQ-RMI training algorithm guarantees a tight bound on the maximum error of the estimated index, which in turn bounds the search and ensures lookup correctness. During the search, the candidate rules are *validated* by matching over *all* fields of the incoming packet. Finally, the highest priority rule is selected out of all the matching rules from all the iSets and the remainder.

The number of neural nets (NNs) in an RQ-RMI model depends on the number of rules it indexes. The original paper suggests four RQ-RMI size categories, reported in Table 1. The larger the model, the longer it takes to train it. However smaller models would fail to achieve the target error bound guarantees and would result in a slower lookup. Thus, there is a fundamental trade-off between the lookup latency and the training time. NuevoMatchUP changes the way RQ-RMI is constructed to modify this trade-off, allowing a much faster training with negligible degradation in the lookup latency.

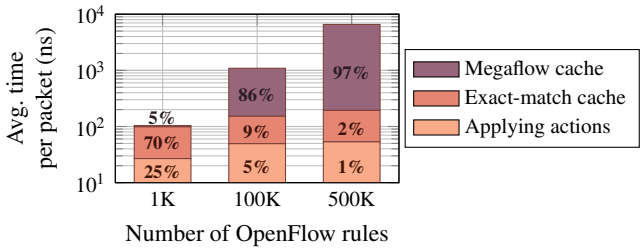


Figure 3: Breakdown of packet processing times in the data-path for different number of OpenFlow rules.

3 Motivation

We analyze OVS's scalability bottlenecks and highlight the potential benefits of using faster packet classification.

For the analysis we use the same setup and workloads as described in §7. In particular, we generate 36 ClassBench OpenFlow rule-sets (12 application types of three size categories each: 1K, 100K, 500K rules), and evaluate the throughput by replaying Caida-short packet trace (100M packets).

Does OVS get slower with more rules? We compare the throughput with 1K rules vs. the throughput with 100K and 500K rules, separately for each ClassBench application type. We observe that the geometrical mean *slowdown* for 100K and 500K rules vs. 1K rules is 5.8× and 9.1×, respectively.

Takeaway 1: OVS does not scale well to a large number of OpenFlow rules.

Where is the bottleneck in the data-path? We analyze the average processing time of a packet in the OVS data-path while varying the number of OpenFlow rules across all the rule-sets. Figure 3 shows that packets spend the majority of time in the megaflow cache, i.e., 86% and 97% of the CPU time on average, for 100K and 500K rules respectively.

Takeaway 2: OVS megaflow cache becomes the main data-path performance bottleneck as the number of OpenFlow rules increases.

Are the control-path upcalls the primary bottleneck? Misses in the megaflow cache trigger upcalls into the control-path. The frequency of the upcalls is hard to predict; it depends on the interplay between the rule-set and the traffic pattern [3, 4]. Unfortunately, frequent upcalls cause major throughput drop. For example, Figure 4 shows the throughput and the rate of deletions and upcalls, sampled every 100ms, for a 100K rules (rule-set number 2 in §7). The higher the

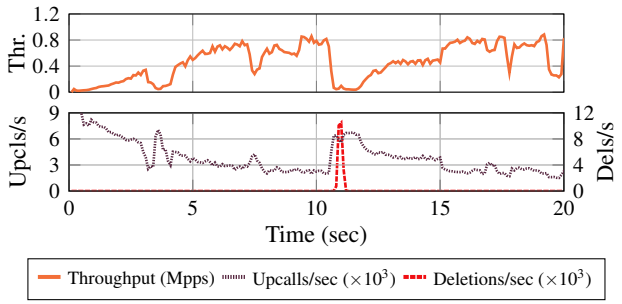


Figure 4: OVS throughput is affected by control-path upcalls.

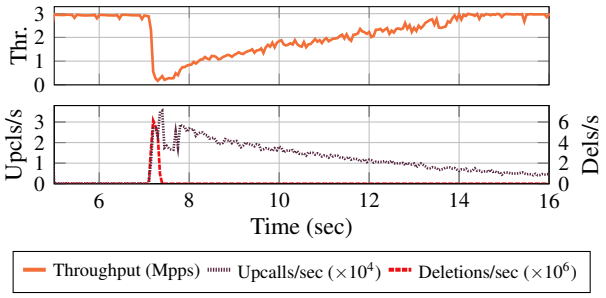


Figure 5: Insertion of new OpenFlow rules: the throughput drops at $t = 7s$ when 60 new OpenFlow rules are added to 500 existing ones. Note the coinciding peak in the deletion rate from the megaflow cache and the subsequent increase in the number of upcalls.

number of upcalls, the lower the throughput. Similarly, the performance drop is observed due to deletions, triggered by the periodic megaflow cache cleanup of idle flows. For other rule-sets the behavior is similar.

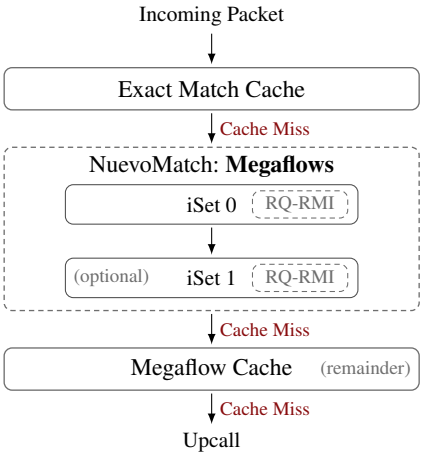
Takeaway 3: frequent upcalls are detrimental to performance.

Impact of OpenFlow rule updates. OVS might experience a sharp drop in throughput when OpenFlow rules are modified.

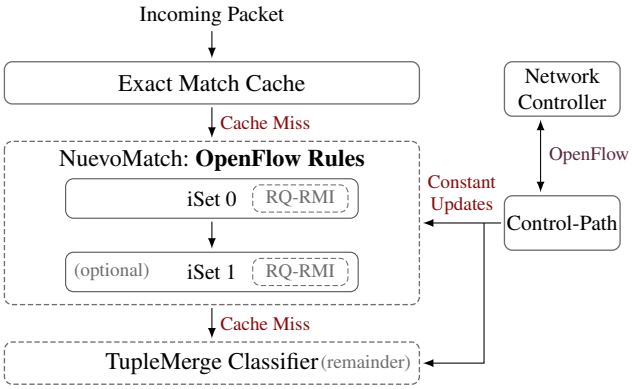
To show that, we install 500 OpenFlow rules in the beginning and update 60 rules at time $t = 7$. Figure 5 shows the results. The moment before the update occurs, there are 144K megaflows in the cache. We see that the update causes about 104K deletions from the megaflow cache, followed by tens of thousands of upcalls. As a result, the throughput drops dramatically and takes a few seconds to recover.

This graph illustrates a general problem rooted in the megaflow algorithm. When inserting new rules that overlap existing ones with lower priorities, OVS must delete all megaflows that correspond to the existing rules (§3). While the magnitude of the throughput degradation depends on the rules being updated, the issue is significant in particular with high update rates.

Takeaway 4: modifying a handful of OpenFlow rules might significantly affect the throughput because of the increase in upcalls.



(a) OVS-CCACHE with NuevoMatch accelerating the megaflow cache.



(b) OVS-CFLOWS with NuevoMatch performing OpenFlow rule classification in the data-path.

Figure 6: Design options for integrating NuevoMatch with OVS. See §8 for the discussion why to choose one over the other.

4 Design Options and Challenges

Our analysis indicates two primary reasons for the OVS performance degradation: (a) poor scalability of the megaflow cache; (b) frequent upcalls to the control-path. In the following we consider two designs to solve these issues.

4.1 OVS with Computational Cache

To tackle the first issue, the most natural solution is to replace the megaflow cache with a more scalable NuevoMatch. This approach is appealing because it fits well in the existing OVS design. Here, NuevoMatch uses the megaflow cache as a remainder, and can be seen as an additional layer of caching for megaflows. We call this approach an *OVS with a computational cache* (OVS-CCACHE).

Figure 6a shows the proposed OVS-CCACHE design, depict-

Num. of OF rules	Upcalls per sec	Num. of Megaflows in cache	Training time est.	Coverage degrad. est.
1K	128	6.5K	30s	3.8%
100K	6.7K	102K	270s	7%
500K	6.4K	90K	250s	8%

Table 2: Characterization of rule update rate requirements in the megaflow cache. NuevoMatch training should be at least 100× faster to be applicable to the megaflow cache.

ing only the data-path. The control-path is unmodified. Incoming packets are first matched against the exact-match cache. A miss is then forwarded to the computational cache provided by NuevoMatch RQ-RMI models. The original megaflow cache serves the lookups which did not match in RQ-RMI. If missed again, the packet continues with the original OVS upcall mechanism.

When new megaflows are added to the data-path, they are first inserted into the original megaflow cache. The RQ-RMI model is periodically re-trained in a separate thread by pulling the added megaflows from the megaflow cache. When the training finishes, the old RQ-RMI models are replaced with the newly trained ones that already incorporate the new megaflows, and the megaflow cache is emptied.

Unfortunately, this solution inherits the performance limitations of the upcall mechanism, and thus would not scale well in case of frequent upcalls.

4.2 OVS with Computational Flows

To solve the issue of slow upcalls, one option is to apply NuevoMatch to the control-path classifier to speed up the handling of upcalls. Unfortunately, control-path tasks go well beyond OpenFlow rule matching, and it is unclear how to use NuevoMatch in this context. Specifically, the control-path effectively implements the algorithm for tracking and generating non-overlapping megaflows. This is the core of the control-path and it is tightly coupled with the rule matching logic. Thus, NuevoMatch is not suitable for control-path acceleration.

On the other hand, the excessive number of upcalls we observed stems primarily from the design choice to generate non-overlapping megaflows for the data-path. The fact that megaflows do not overlap is an essential feature in OVS design that allows fast-path performance optimizations, but it is also the one that deteriorates the throughput dramatically in case of frequent upcalls [3, 4].

Therefore, our proposed solution, *OVS with computational flows (*OVS-CFLOWS*)*, leverages NuevoMatch to perform efficient packet classification directly on complex OpenFlow rules, without resorting to non-overlapping megaflows. As a result, we remove the megaflow cache mechanism and the associated control-path logic, and obviate the need for upcalls. This approach, while more intrusive than OVS-CCACHE, holds the promise to boost OVS performance both with and without OpenFlow rule updates. How it fairs against OVS-CCACHE is one of the questions we answer in our evaluation.

Figure 6b shows the design of OVS-CFLOWS. While it resembles OVS-CCACHE, the difference is that NuevoMatch here is used to match OpenFlow rules instead of megaflows as in OVS-CCACHE. Similarly to OVS-CCACHE, updates are first inserted into the remainder (we use TupleMerge [6] for its implementation), and RQ-RMI models are periodically retrained to accommodate them.

4.3 Challenge: Slow NuevoMatch Updates

Unfortunately, in practice, NuevoMatch cannot support either OVS-CCACHE or OVS-CFLOWS. Recall that rule modification in the classifier requires retraining all its RQ-RMI models from scratch with the new, modified set of rules (§2.1). Therefore, the rule update rate is bounded by the training time of the models, which in turn depends on the number of rules in the classifier rather than on the number of modified rules.

In the following, we analyze the update rate requirements for OVS-CCACHE and OVS-CFLOWS, and show that NuevoMatch is over two orders of magnitude slower than required.

Megaflow cache rule churn. To understand the training rate requirements for NuevoMatch in OVS-CCACHE, we analyze the rule churn rate in the megaflow cache. For each OpenFlow rule size category we measure (1) the average rate of upcalls, which is equivalent to the rate of updates in the megaflow cache (we count insertions only, as NuevoMatch supports deletions without retraining), and (2) the average number of megaflows in the cache, which dictates the NuevoMatch training time if it were used to accelerate the megaflow cache.

Table 2 shows that for larger rule-sets (100K, 500K) there are about 6.5K upcalls per second, and the megaflow cache holds about 100K megaflows. Thus, NuevoMatch would have to retrain the model with 100K rules every 150 μs. This is of course unrealistic: training a model of that size would require about 270 seconds according to the original paper.

The solution suggested by the authors of NuevoMatch is to accumulate the updates in the remainder and serve the queries from it while training. Thus, the *coverage* of the RQ-RMI model is lower during the training; hence, the performance is lower because more queries are served in the remainder. When the training is finished, the coverage improves, and a new round of training begins right away to catch up with the rules modified during the previous training round.

Unfortunately, this option is not practical either. If 6.7K rules get modified each second, the expected coverage degradation per second would be about 7% (see Table 2). If we accumulate the updates while training for 270 seconds, the coverage will become practically zero, nullifying the NuevoMatch performance benefits completely. For comparison, even to

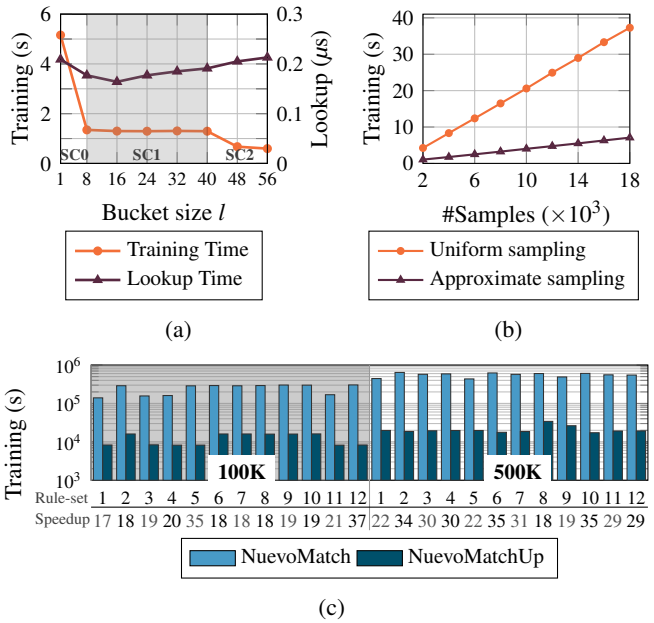


Figure 7: **(a)** The effect of the bucket size l on the RQ-RMI training and lookup times. See RQ-RMI size categories (SC) in Table 1. **(b)** Training using approximate sampling is faster. Here we train 500K rules using bucket size $l = 40$. **(c)** The training implementation of NuevoMatchUP is 20 times more efficient than that of NuevoMatch.

achieve the coverage of 25%, which is the cutoff suggested in the paper for NuevoMatch to provide minimum performance benefits, the training must complete within 4 seconds. This is almost *two orders of magnitude* faster compared to 270 seconds that NuevoMatch allows today. We use the same model as the original paper to produce these estimates: $E = R \cdot e^{-U/R}$, where R and U are the total number of rules and the number of updates respectively, and E is the expected number of rules in the model left after the updates.

OVS-CFLOWS update requirements. The update rate of OpenFlow rules varies between 400 to 338K updates per second [12, 13]. Supporting an average rate of 100K updates per second in NuevoMatch would require retraining every 500ms to achieve a coverage of 90% for a rule-set of 500K OpenFlow rules. Unfortunately, the actual NuevoMatch training time for a rule-set of that size is about 600s, which is over three orders of magnitude slower.

We conclude that *NuevoMatch training algorithm is too slow to support the update requirements of OVS in the considered designs*.

5 NuevoMatchUP: Speeding-up Updates

We introduce *NuevoMatchUP* (NMU), a series of enhancements to NuevoMatch which together significantly improve its update rate by several orders of magnitude.

NMU introduces important changes to the RQ-RMI construction and training algorithms, as well as to their implementation. First, it enables creating much smaller (thus faster-to-train) RQ-RMI models by constructing iSets with overlapping rules. Second, it enables major improvement in training speed by cutting down the number of memory accesses. We now discuss these changes in detail.

5.1 Relaxing iSet Constraints

An iSet s is a set of rules associated with a field h_s for which rules do not overlap (§2.2). We relax the no-overlap constraint, by allowing overlap between a certain number of rules. Informally, a *relaxed iSet* is an iSet with up to l overlapping rules in field h_s, grouped in *buckets* (defined next).

We now describe the algorithm for constructing a relaxed iSet s on a header field h_s.

Lemma 1. *Given an OVS classification rule r and a packet header field h, the set of values in h that match r can be represented by an integer range, denoted as $h(r)$.*

The correctness of the lemma directly follows from the usage of prefix based wildcard representation in OVS.

Definition 1. *A bucket is a set of up to l rules. We say that two buckets b_1 and b_2 do not overlap with respect to the header field h if for any rule r_1, r_2 in b_1, b_2 respectively, $h(r_1)$ does not overlap $h(r_2)$.*

To create buckets that do not overlap with respect to the header field h_s, we sort the rules by their ranges in h_s, and iterate over them allowing up to l overlapping ranges per bucket. Whenever we encounter a rule with a range that does not overlap with its predecessors, we include it in a new bucket. If buckets contain less than l rules, we merge adjacent buckets while keeping the constraint to have at most l rules per bucket. Next, we use RQ-RMI models to *learn the distribution of the buckets* rather than the distribution of the rules [26].

Since the number of buckets is smaller by up to a factor of l than the number of rules, RQ-RMI models in NuevoMatchUP are smaller and train faster than in NuevoMatch (see Table 1). Of course, the cost of this optimization is a slower lookup: all the rules in the same bucket must be validated via a linear scan. This trade-off, however, turned out to be beneficial to accelerate training with a negligible slowdown for the lookup. Figure 7a demonstrates this trade-off using a representative rule-set (12-500K, see §7). Buckets of sizes $l = 8, 48$ change the RQ-RMI size category and dramatically improve the model's training performance. Other bucket sizes ($l = 16, 24, 32, 40, 48, 56$) do not change the RQ-RMI size category and only add to the linear scan overhead.

5.2 Training via Approximate Sampling

In NuevoMatch, each neural net in RQ-RMI is trained using supervised learning on a labeled dataset S that is generated in advance. The dataset is sampled from an ordered set of ranges, R, sorted by the ranges' start values. An RQ-RMI model learns the function represented by the ordered set R: it maps an input to the index of the matching range. To learn this function, NuevoMatch samples from it uniformly [26]. This uniform sampling is expensive, as it requires to scan all the ranges and sample from them according to their relative sizes in the function input domain. This sampling must be done for each neural-network (NN) in the RQ-RMI model, sometimes multiple times to achieve the desired accuracy.

Our goal is to modify the sampling process to reduce the number of memory accesses from $O(|R|)$, which can be on the order of tens of thousands per NN, to $O(|S|)$, which is about several thousand per NN. Doing so is not trivial since the training converges faster when the samples are distributed with parameters $(\mu, \sigma) = (0, 1)$.

We make two observations. First, it is possible to analytically estimate the expectation μ and standard deviation σ of a uniform sampling of the NN input domain (see Appendix A.1), and thus enable correct normalization of the samples regardless of the way they are actually sampled. Second, given correct normalization, sampling R in a non-uniform way might only affect the model accuracy but not the lookup correctness, thanks to the search in the rule array (§2.2) that eliminates model approximation errors.

These observations allow us to accelerate the sampling process as follows. We generate a set of 32 samples per batch, each of the form (x, y). First, we uniformly select a range r with index i from R. Second, we uniformly select a value $x' \in r$. We then generate a normalized $x = \frac{x'-\mu}{\sigma}$; $y = \frac{i}{|R|}$ as in the original algorithm.

In Figure 7b we train a model over a representative rule-set (12-500K, see §7) and get 4-5.3× faster training using approximate sampling.

5.3 Optimized Training Implementation

NuevoMatch uses a hybrid training approach that mixes Python code, TensorFlow, and a custom native library. In contrast, NuevoMatchUP is implemented in C++, which reduces its memory requirements, and takes advantage of the CPU SIMD instructions. Figure 7c shows a 23.8× geometrical mean speedup of NuevoMatchUP over NuevoMatch over all rule-sets. In this experiment we disable all algorithmic optimizations, highlighting the speedup due to the implementation.

5.4 Putting It All Together

Each of the described optimizations in isolation would not suffice to achieve the target performance goals to support the necessary update rate. However, when combined, they allow between two to three orders of magnitude faster training (depending on the rule-set), making NuevoMatchUP suitable for integration with OVS.

6 Implementation

We implement OVS-CCACHE in C as an additional OVS module, and NuevoMatchUP in C++ as an external library (*libnuevomatchup*). We add support for OVS-CFLOWS by changing existing components in several OVS modules[1].

Overview. OVS uses *poll mode driver* (PMD) threads for packet processing and *revalidator* threads for integrity. The flows [2] are kept in a dedicated *flow-table*, one per PMD thread, that supports a single writer and multiple concurrent readers. A PMD thread is responsible for inserting new flows into its flow-table, while the revalidator threads remove stale ones.

We modify OVS as follows. We introduce a single *trainer* thread to train all the NuevoMatchUP models used by each PMD thread. In addition, we add *manager* threads, one per PMD thread, for tracking the PMD flows, and create training tasks to accommodate the changes.

Concurrency. We use a fine grained locking with a spinlock per flow-table entry, and limit the number of occurrences in which we modify the flow-table. This mechanism is essential mostly for OVS-CCACHE, in which valid flows frequently migrate between the megaflow cache and the RQ-RMI models.

Training RQ-RMI models. At any given time, there are two instances of RQ-RMI models per *manager* thread: the one that is used by an active classifier in the packet processing pipeline, and the one being trained, referred to as a *shadow* model. In each iteration, a *manager* thread goes over all the flows, checks which are marked for deletion and which are new. Next, it enqueues the request with the modified rule-set to the trainer thread to retrain the shadow model. The rules added during training are updated in the remainder of the active classifier. When the training completes, the active classifier replaces its model with the newly trained shadow model, and the recently learned rules are removed from the remainder. This process repeats whenever the number of flows in the remainder is higher than 10%.

Data-path modifications. The megaflow cache constructs a new hash table whenever it encounters a previously unseen mask, and destroys it when it no longer holds flows.Since in OVS-CCACHE, megaflows frequently migrate between the megaflow cache and the RQ-RMI models, we enable the exis-

[1] https://github.com/acsl-technion/ovs-nuevomatchup

[2] In this section we use the OVS terminology and refer to match-action rules of any kind, either megaflows or OpenFlow rules, as flows.

Name	Number of Packets	Unique 5-Tuples	Average Delay Between Packets (μs)
CAIDA-short	100 M	6 M	1.68 ± 69.54
Mawi	237 M	15 M	3.39 ± 9.11
CAIDA-long	401 M	23 M	1.62 ± 119.20

Table 3: Evaluated traces.

tence of empty hash tables to reduce the number of hash table constructions and deletions to bare minimum.

6.1 Updates in OVS-CFLOWS

OVS-CFLOWS offers a new design trade-off for performing OpenFlow rule updates. Specifically, it allows trading the time it takes to activate the updated rules in the data-path for higher throughput during the update. When a network controller updates the rules, it might need to ensure that the updates are installed and visible to the data-path. In OVS and OVS-CCACHE, the acknowledgement to the controller is sent when the rules are installed in the *control-path*. The data-path pulls the rules on demand via upcalls.

In OVS-CFLOWS, we can implement two policies. The *instant update* policy updates the active classifier with the new rules immediately, pushing them into the remainder and thus applying them to the data-path without any delay. The *delayed update* policy first stores the new rules in a temporary structure not visible to the classifier, retrains the shadow model and only then updates the data-path.

As we will see in the evaluation, when a large number of updates is necessary, the instant update policy results in lower throughput while the new rules are being added due to reduced model coverage, but provides lower update latency from the perspective of the network controller. Delayed updates yield higher latency for the controller, but avoid the throughput degradation during the update. On the other hand, with only a handful of updated rules, the immediate update policy achieves low latency without affecting the throughput.

7 Evaluation

We perform end-to-end experiments and provide an in-depth analysis of the system performance using microbenchmarks.

7.1 Methodology

Setup. We use two machines connected back-to-back via Intel X540-AT2 10Gb Ethernet NICs with DPDK-compatible driver. All our tests stress the OVS logic thus the workload is CPU-bound and the network is not saturated.

The system-under-test machine (SUT) runs Ubuntu 18.04, Linux 5.4, OVS 2.13 with DPDK 19.11, on Intel Xeon Sliver 4116 CPU @ 2.1GHz with 32KB L1 cache, 1024KB L2 cache, and 16.5MB LLC. The load-generating machine (LGEN) runs a native DPDK application that generates packets on-the-fly according to a predefined policy, and records the responses from the SUT.

We configure both machines to use DPDK with four 1GB huge pages for maximum performance. We disable hyper-threading and set the CPU governor to maximum performance for stable results.

Synthetic OpenFlow rules. We generate OpenFlow rules using ClassBench [33], the standard benchmark for packet classification [6, 18, 19, 26, 35, 39]. ClassBench creates 5-tuple rule-sets that correspond to the distribution of three applications: Access Control List (ACL), Firewall (FW), and IP Chain (IPC). We generate rule-sets with 1K, 100K, and 500K rules, each size category with 12 rule-sets. We only generate rules for either TCP, UDP, or ICMP IP protocols. The mapping between the generated rule-sets' names to their numbers appears in Appendix A.3.

Traffic traces. The traces are summarized in Table 3 and detailed below.

(1) **CAIDA** [2]. The real trace from the Equinix data-center in Chicago, collected in January 2019. We use CAIDA-short in all experiments except for the one that needs longer trace (Figure 14) where we use CAIDA-long.
(2) **MAWI** [37]. The real trace from a link between Japan and the USA, collected in April 2020.

Adjusting traces to rules. There are no published OpenFlow rules used for processing the packets in the recorded traces. We thus resort to the method used in prior work [26]. Specifically, we modify the packet headers in the trace to match the evaluated ClassBench rule-sets, as follows. For each unique 5-tuple we uniformly select a rule, and modify the packet header to match it. We also set all TCP packets to have a SYN flag. This method preserves the temporal locality of the original trace while consistently covering all the rules.

Packet generation policies. We use minimum-size 64-byte packets to stress the OVS classification logic. We evaluate the system with two load generation methods.
Constant TX rate. To ensure unbiased evaluation, we run the experiments with a constant-rate load generator, and report the highest rate that permits the average drop rate over the whole trace to be below 1%. The first 5% of the packets in each trace are used as a warmup and the associated drops are ignored. We do this as we observe that bootstrapping the megaflow-cache causes many packet drops. With 5% warmup packets, we achieve consistent throughput results.
Adaptive TX rate. We use the timestamps from CAIDA/MAWI packet traces but scale down the inter-packet delay to replay the packets at the highest rate that strives to maintain an average per-second packet drop rate below 1%. To achieve that, we dynamically adjust the

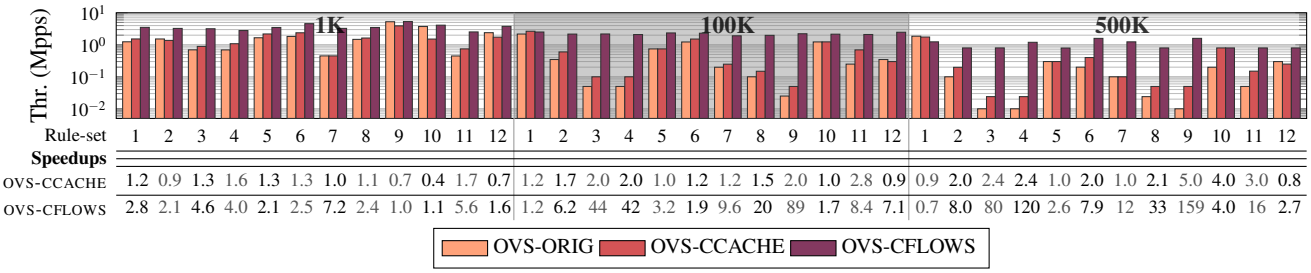


Figure 8: OVS-CFLOWS, OVS-CCACHE and OVS-ORIG on CAIDA-short using constant TX rate. Higher is better.

sending rate once per second: we cut it to half when the drop rate over the last second exceeds 1%, and increase by 50% otherwise. This methods provides a conservative estimate of expected system performance because of its simplistic congestion control which does not aggressively ramp up the throughput after drops.

Measurements. We measure end-to-end performance, i.e., receiving, processing, and sending packets back to the LGEN. We preload all OpenFlow rules into control-path.

OVS configuration. We use the default OVS configuration [22] both for the baseline and our designs: revalidator threads support up to 200K flows, flows with no traffic are removed after 10 seconds, and the *signature-match-cache* (SMC) is disabled. The EMC insertion probability is 20%. Connection tracking is not used. Unless stated otherwise, all experiments use a single NUMA node with one core dedicated to a PMD (poll mode driver) thread and another core dedicated to all other threads. Thus, the baseline OVS, OVS-CCACHE, and OVS-CFLOWS always use *the same number of CPU cores*.

NuevoMatchUP configuration. We use iSets with minimum 45% coverage, and train RQ-RMI neural nets with 4K samples. Similar to [26], we repeat the training until the RQ-RMI maximal error is lower than 128, and stop after 6 unsuccessful ones. We set $l = 40$, namely, each iSet bucket has at most 40 overlapping rules. We use the same RQ-RMI size categories as in Table 1. Due to the use of buckets, the largest size category is never used. We keep OVS's flow matching mechanism that supports an arbitrary number of fields, but limit the iSet construction mechanism to use 5-tuples.

We train RQ-RMI models based on either all megaflows (for OVS-CCACHE) or OpenFlow rules (for OVS-CFLOWS). The model size is determined by the NuevoMatchUP algorithm to allow lowest error, fast training time and low memory footprint.

7.2 End-to-end Performance

Figure 8 shows the throughput comparison of OVS-CFLOWS, OVS-CCACHE and OVS-ORIG (unmodified OVS) for *CAIDA-short* with constant TX rate and without updates to the OpenFlow rule-set. The geometric mean speedups of OVS-CCACHE

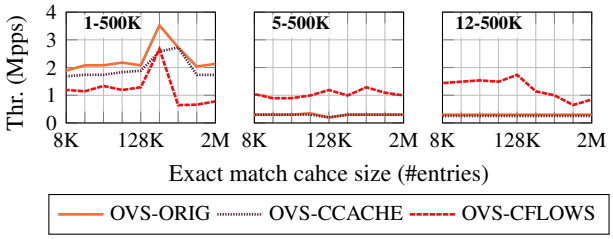


Figure 9: The effect of the exact-match cache size on throughput for the top three fastest rule-sets with 500K rules.

are 1.02×, 1.5×, and 1.9× for 1K, 100K, and 500K OpenFlow rules respectively. Note that for OVS-CCACHE the computational cache is *constantly updated* with newly installed megaflows.

The same setup with OVS-CFLOWS yields higher speedups. OVS-CFLOWS is 2.6×, 8.5×, and 12.3× faster than OVS-ORIG for 1K, 100K, and 500K OpenFlow rules, respectively. Not only is OVS-CFLOWS faster than OVS-CCACHE, but it also maintains a relatively stable absolute throughput for 100K and 500K rules. OVS-ORIG performance varies substantially across rule-sets of the same size, whereas OVS-CFLOWS shows more homogeneous behavior. OVS-ORIG has particularly low performance for larger rule-sets (e.g., 3,4 for 500K) due to a massive number of upcalls.

The performance trends with an adaptive TX rate are consistent with those obtained with the constant TX-rate (see Figure 18a in the Appendix). The speedups are still significant but more modest for two reasons: the adaptive TX fails to increase the sending rate fast enough after packet drops, which particularly affects the absolute throughput of faster OVS-CFLOWS. At the same time, it achieves higher average rate for lower-performant OVS-ORIG and OVS-CCACHE because it suffices to slowly increase the rate when the traffic pattern affords that. Rule-set 9-100K and 3-500K are the best illustrations of this effect.

Rule-set 1-500K performs differently from the rest. Here, OVS runs faster with 100K and 500K rules than with 1K rules. We find that this is due to the high temporal locality, which leads to a low upcall rate (over 3× less than in other rule-sets for 500K) and a small megaflow cache. This analysis

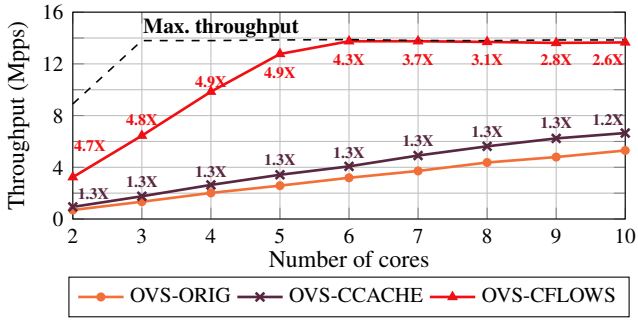


Figure 10: OVS throughput as a function of the number of cores. One core is dedicated to revalidator, manager, and trainer threads. For the rest, we allocate one PMD thread per core. The maximum throughput is measured with only EMC hits. The numbers refer to speedups vs. OVS-ORIG.

is corroborated by the experiments that vary the EMC size (Figure 9). This result motivates dynamic choice between the original and the suggested classification mechanisms as we discuss in §8.

The same experiments on the *Mawi* trace yield low throughput results for OVS-CCACHE and OVS-ORIG using constant TX rates due to the excessive number of drops. For the dynamic TX rate, the geometric mean speedups are: 2.1×, 18.2×, and 18.7× for 1K, 100K, and 500K rules for OVS-CFLOWS, and 1.02×, 1.4×, and 1.7× for 1K, 100K, and 500K rules for OVS-CCACHE.

7.3 Sensitivity to OVS parameters

The effect of the EMC size. We take the top three rule-sets with 500K rules that perform best for OVS-ORIG (rule-sets 1,5 and 12), and test their throughput with different Exact Match Cache (EMC) sizes (8K (default) to 2M), see Figure 9. The performance effect of the EMC size depends on the rule-set. The default size (8K) works reasonably well, whereas a too large EMC reduces throughput, likely because of the CPU cache contention. The relative performance of different designs, however, remains largely the same with the EMC of up to 128K entries.

Megaflow cache size. When the OVS megaflow cache reaches its maximum capacity it flushes all its contents. We validated that this never occurs in our experiments. Thus, the megaflow cache can practically grow as necessary, periodically evicting idle (for 10s) flows. This is the most favorable configuration.

Data-path scalability. We add PMD threads and pin them each to a separate core, while dedicating one more core for the revalidator, manager and trainer threads. We use the *CAIDA-short* trace with the constant TX setting, and report the results of a representative rule-set with 1K rules (3-1K) in Figure 10.

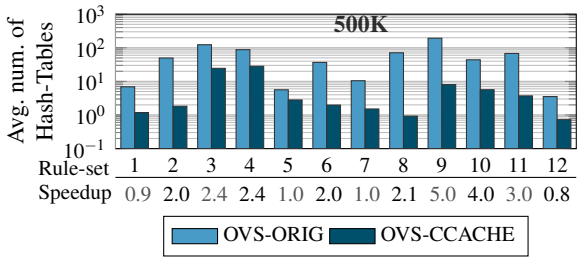


Figure 11: The average number of hash-tables in the megaflow cache on CAIDA-short trace. Lower is better. See full chart in the Appendix (Figure 18b).

This is the best-case scenario for OVS-ORIG because in larger rule-sets it is much slower. We measure the upper bound of the OVS forwarding performance by sending 100M packets that always hit the 8K flows-large EMC (black dashed line). For a 10Gb NIC, the performance saturates at 13.8Mpps, 93% of the line-rate[3].

Figure 10 shows that OVS-CCACHE maintains a constant speedup of 1.3× over OVS-ORIG, even though more PMD threads lead to higher model retraining load. This is because the single trainer thread is fast enough to retrain models from eight PMD threads (nine cores in total on the graph). The additional, ninth PMD thread saturates the trainer. Without training fast enough, the scaling is no longer linear (1.2× speedup vs. 1.3× for fewer PMD cores). Thus, more PMD threads would require allocating additional trainer cores to maintain the speedup.

OVS-CFLOWS reaches the maximum throughput with five PMD cores (six cores overall), a 4.3× speedup over OVS-ORIG using the same number of cores. OVS-ORIG would have required about 26 cores (linear extrapolation of the current trend) to reach the same performance. Note that in contrast to OVS-CCACHE, models in OVS-CFLOWS are not retrained in the steady state between OpenFlow rule updates, thus the throughput scales linearly with more PMD threads without additional trainer cores.

7.4 Analysis of OVS-CCACHE

Understanding performance variability of OVS-CCACHE. Why does OVS-CCACHE is faster than OVS-ORIG for some rule-sets and is on-par or slower for others? The answer follows from Figure 11 which shows the average number of megaflow cache hash-tables traversed for OVS-ORIG and OVS-CCACHE. Recall that the classification is slower with higher number of hash-tables [3]. The computational cache achieves higher speedups when the number of hash-tables traversed by OVS-ORIG is large enough to justify inference computations instead of memory lookup. As a result, the performance

[3] 14.88Mpps for 64B packets on a 10Gb NIC, considering bytes of Ethernet preamble and 9.6ns of inter-frame gap.

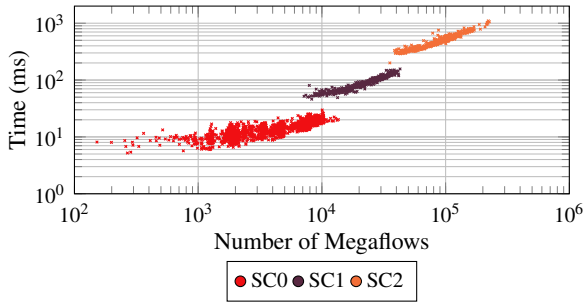


Figure 12: NuevoMatchUP training time in OVS-CCACHE as a function of number of megaflows and the model size category (Table 1). SC0=5, SC1=21, SC2=133 neural nets.

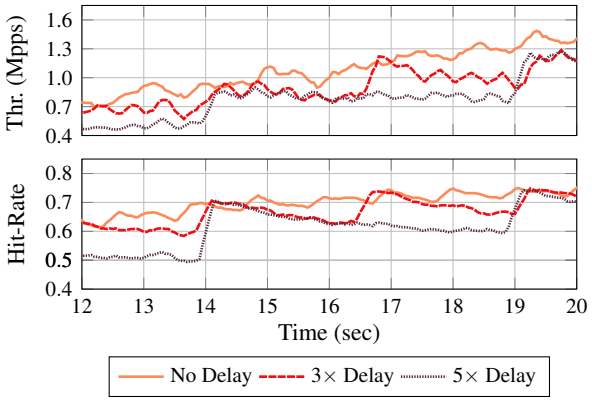


Figure 13: OVS-CCACHE throughput and computational cache hit-rate for different training rates, while adding megaflows. Rapid retraining is critical for high throughput.

savings from using NuevoMatchUP are higher in such cases.

Training in the data-path. In the following experiments, we generate packets at a constant rate of 5 Mpps. This setup saturates the OVS packet-processing pipeline and thus helps highlight the reasons why NuevoMatchUP improves the end-to-end performance.

We measure the actual training time for RQ-RMI models in the data-path during the experiment. To understand the training behavior, we measure the number of megaflows being used and the training time. We show the training time for each of the three used RQ-RMI size categories.

Figure 12 shows that the training time ranges from milliseconds for a small number of megaflows, to about one second for 200K megaflows. For comparison, NuevoMatch reported the training time of 270 seconds for a rule-set with 100K rules which NuevoMatchUP can train in 500ms - 540× faster.

Hit-rate and training time. We further analyze the dynamic throughput behavior of OVS-CCACHE when new megaflows are installed in it by the control path.We use a single rule-set with 100K OpenFlow rules (rule-set 9-100K), and vary the training rate while measuring the throughput.

Figure 13 shows that when new rules are just added the

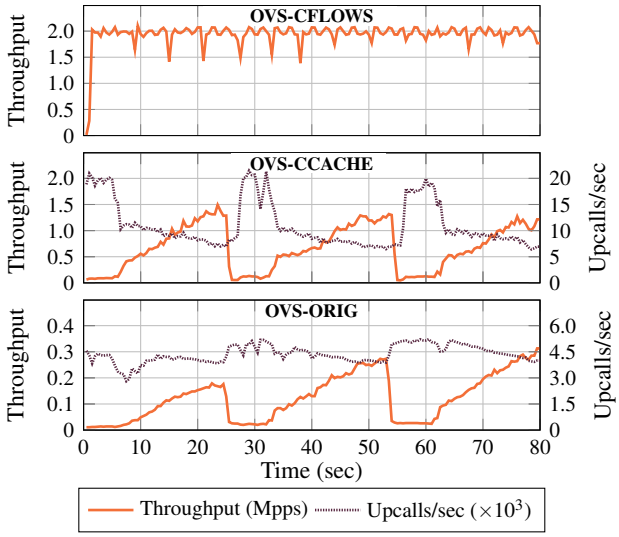


Figure 14: The throughput and number of upcalls over time.

throughput decreases initially, but then recovers. This behavior is expected. The rules are first installed in the original megaflow cache, which causes an increase in the number of hash-tables in it and the throughput drops. Also, the hit-rate in RQ-RMI models drops because the new rules are not yet part of the model. However, after the RQ-RMI model is retrained with the new rules, the hit-rate increases back, until the new rules get installed, and so on. Observe that the throughput is lower when the training is slower (i.e., 5× slower than the original rate) since in such cases the system cannot keep up with new rules. This experiment clearly demonstrates the importance of fast training provided by NuevoMatchUP.

Updates in OVS-CCACHE. We measure OVS-CCACHE average update rate for a different number of OpenFlow rules. We see 944, 11.6K and 11.2K updates per second, on average, for 1K, 100K and 500K rules, respectively.

Further inspection reveals that OVS-CCACHE sensitivity to upcalls affect its update rate, similar to the effect presented in Figures 4,5 for OVS-ORIG. Since we cannot explicitly control the upcalls, we test this by artificially delaying NuevoMatchUP updates and measuring the temporal behavior of the throughput, number of upcalls, and iSet coverage. We find that while NuevoMatchUP accelerates the megaflow cache, upcalls are still the dominating factor for its performance. See Appendix A.2 for details.

7.5 Analysis of OVS-CFLOWS

No upcalls in OVS-CFLOWS. We compare the throughput of OVS-ORIG, OVS-CCACHE and OVS-CFLOWS over time, sampled every 500ms. We use the *CAIDA-long* trace, so that each experiment is roughly 80 seconds long, and show the results of a single rule-set with 100K OpenFlow rules (rule-set 9-100K) while keeping the rules unmodified throughout the

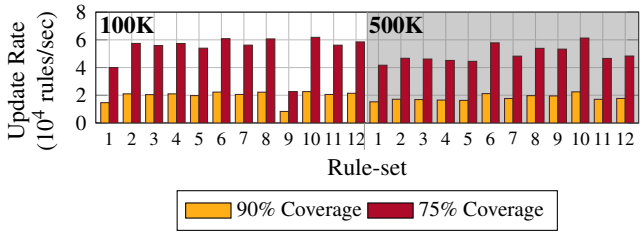


Figure 15: Max OpenFlow rule update rate of OVS-CFLOWS, for maintaining 75% and 90% NuevoMatch coverage.

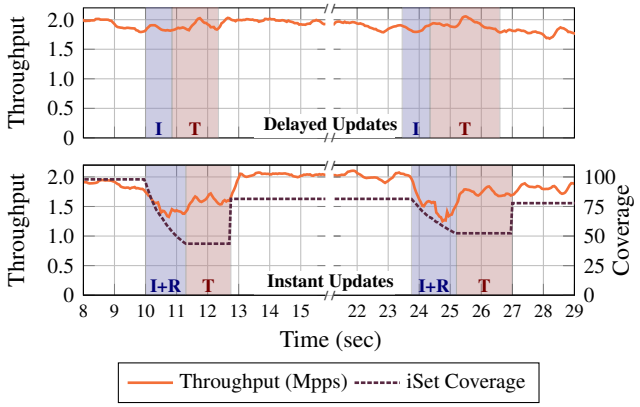


Figure 16: Update policies in OVS-CFLOWS. I: iteration time, R: remainder time, T: training time.

experiments. Other rule-sets behave similarly.

The results in Figure 14 clearly illustrate the benefits of OVS-CFLOWS design (top graph). OVS-ORIG (bottom) and OVS-CCACHE (middle) suffer from significant performance fluctuations directly correlated with the number of upcalls into the control-path. This experiment corroborates our conclusions in Section §3. OVS-CCACHE inherits these performance problems because it simply replaces the megaflow cache with a faster alternative, but uses the same fast-/slow- path split. It does, however, improve the end-to-end throughput. The higher throughput of OVS-CCACHE is the reason why its upcall rate is proportionally higher than in OVS-ORIG.

OVS-CFLOWS avoids the use of upcall mechanism altogether, achieving consistently higher throughput and good scalability for a large number of OpenFlow rules.

OpenFlow updates in OVS-CFLOWS. We estimate the maximum OpenFlow rule update rate in OVS-CFLOWS for 100K and 500K rule-sets, as they pose the main challenge. Unfortunately, we could not measure the maximum update rate experimentally because of the slow OVS control-path that did not allow us to invoke updates back-to-back.

Our estimate of the update rate indicates the number of rules that can be updated per second in order to achieve 75% and 90% coverage by NuevoMatchUP. These are conservative coverage values that were shown to result in small throughput degradation in NuevoMatch. To estimate, we measure the training time for each rule-set and compute the expected update rate according to the formula in §4.3. The results in Figure 15 show an average of 19K and 51K updates per second for 90% and 75% coverage, respectively. Both size categories achieve similar update rates since the average training time per rule is roughly the same, while the coverage deteriorates slower with more rules. These results assume the use of delayed updates which achieve higher throughput during the update.

Throughput during OpenFlow rule updates. We periodically add bundles of 125K new OpenFlow rules, so the number of rules increases throughout the experiment. We use this number of updates to make the dynamic system behavior over time more visible. We disable the EMC so that the measurements capture only NuevoMatchUP characteristics. We start the experiment with 100K OpenFlow rules, and measure the throughput and iSet coverage over time. We show the results on a representative rule-set (rule-set 9-500K), but the performance is representative of all rule-sets.

Figure 16 compares the delayed and instant update policies (§6.1). For the delayed policy, the time it takes for the data-path to receive the recent changes includes the time to process new rules (iterate over them) and to train, whereas in the instant updates setting, it includes the iteration and remainder update times. The training time depends only on the total number of rules, i.e., 225K and 350K in the first and second training sessions at 10 sec and 24 sec respectively. As expected, the instant update policy causes throughput degradation because the rules are added to the remainder, and thus the model coverage is low. Further, the accesses to the remainder data structure must be synchronized, creating contention. In this case, the use of delayed updates is beneficial as insertions do not cause measurable performance drop.

However, the instant update policy works well when the number of the inserted rules is small. An experiment using bundles of 100, 1K, and 10K new OpenFlow rules yields a 150ms-long drop in throughput with a maximum drop of 2%, 8% and 13% for 100, 1K and 10K rules, respectively. We start OVS with 500 OpenFlow rules and issue an update at $t = 10$ seconds. We use the *CAIDA-short* trace and the constant TX setting with 2.5Mpps. We use the same rule-set as in Figure 16 (rule-set 9-500K); other rule-sets behave similarly. We disable the EMC so the measurements capture only the characteristics of NuevoMatchUP. Figure 17 reports the throughput and iSet coverage within a three second time-frame surrounding the update.

8 Discussion and Future Work

Combining OVS-CCACHE and OVS-CFLOWS. Our evaluation shows that in most cases, OVS-CFLOWS is faster than both

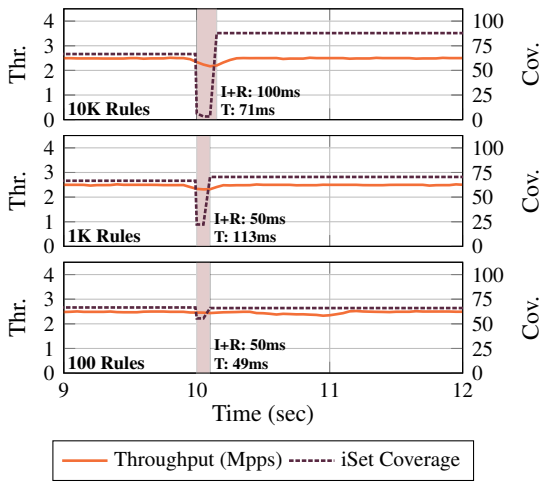


Figure 17: OVS-CFLOWS throughput and iSet coverage upon OpenFlow rule updates using the instant update policy. I: iteration time, R: remainder time, T: training time.

OVS-CCACHE and OVS-ORIG. However, there are cases where it would be desirable to switch between the classification mechanisms dynamically. The computational cache is beneficial when the number of hash-tables in the megaflow cache increases. This can be used to determine when to use it instead of the megaflow cache. Similarly, when the number of control-path upcalls increases, they become the main bottleneck, suggesting the use of OVS-CFLOWS.

NIC OVS offloads. OVS-CCACHE is compatible with the OVS ecosystem, and can be used with in-NIC OVS offloads [20]. In particular, it may accelerate the CPU handling of misses to the hardware OVS cache. Another question is how to use NIC OVS offloads with OVS-CFLOWS. It was shown that NICs become slow when the number of updates gets higher [13]. Thus, switching to OVS-CFLOWS whenever a high number of cache misses is detected may improve performance. We leave it for future work.

In-switch applications. Our work shows a practical use of NuevoMatchUP in packet classification. There are many similar tasks, e.g., longest-prefix matching in switches, which cannot scale due to small on-chip memory. We believe that NuevoMatchUP might help scaling up these tasks by compressing the indexing structure to save on-chip memory.

In-NIC NuevoMatchUP. RQ-RMI inference is a hardware-friendly task. Enabling its execution on the emerging data-parallel accelerators integrated with SmartNICs [21] may improve flexibility of the restricted packet classification offloading logic in NICs today.

P4 OVS. The possibility to use OVS with P4 in addition to OpenFlow was recently suggested [24]. Both the computational cache and computational flows are compatible with P4 as it uses the general structure of match-action tuples which is the fundamental building block for NuevoMatch.

9 Related Work

Packet classification. Software algorithms for packet classification are categorized into decision-tree approaches [9, 10, 18, 19, 28, 35, 39] and hash-table approaches [6, 22, 30]. NuevoMatch [26] is a new approach that shows superior performance for a larger number of rules, hence our choice to use it in this work.

OVS performance. Previous works have highlighted the problem of match-action fragmentation in OVS, and exploited it for mounting denial of service attacks on OVS [3, 4]. Ours is different: it analyses the causes of throughput degeneration and offers a solution.

Machine-learning in the data-path. Several works apply machine-learning models in performance-critical parts of the design, i.e., flash devices [11], RDMA key-value stores [36], programmable switches [38], and NICs [29]. To the best of our knowledge, ours is the first work that applies neural nets and integrates their training into a virtual network switch.

Trading memory accesses for computations. The pioneering work on learned indices [16] and several later works [5, 7, 14, 15, 17, 31] have shown the performance benefits of trading memory accesses for computations using machine-learning models, applying them to data-bases and key-value stores. NuevoMatch [26] extends these concepts and introduces the RQ-RMI data-structure that specializes in range-value queries. Our work improves the training technique of NuevoMatch by several orders of magnitude, making its integration with real-world systems feasible.

10 Conclusion

OVS is a leading virtual networking infrastructure used by many cloud systems. Our work demonstrates two designs which improve its throughput and scalability. We adopt a recent NuevoMatch algorithm for packet classification using neural nets, and integrate it with OVS. Our modifications to NuevoMatch make its use in OVS practical by accelerating its training by over three orders of magnitude. We show significant improvements in both steady-state throughput and update rate for large rule-sets on real-world packet traces. We believe that our work opens new opportunities to practical applications of neural-net based data structures in production networking systems.

11 Acknowledgements

We thank the anonymous reviewers of NSDI'22 and our shepherd Anuj Kalia for their helpful comments and feedback. This work was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate. We gratefully acknowledge support from Israel Science Foundation (Grant 1027/18).

References

[1] The OpenStack authors. The OpenStack project. https://docs.openstack.org/liberty/networking-guide/scenario-classic-ovs.html, 2021.

[2] CAIDA. The CAIDA UCSD anonymized internet traces. http://www.caida.org/data/passive/passive_dataset.xml, 2019.

[3] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Korösi, Balázs Sonkoly, Dávid Haja, Dimitrios P. Pezaros, Stefan Schmid, and Gábor Rétvári. Tuple space explosion: A denial-of-service attack against a software packet classifier. In *ACM CoNEXT*, 2019.

[4] Levente Csikor, Vipul Ujawane, and Dinil Mon Divakaran. On the feasibility and enhancement of the tuple space explosion attack against Open vSwitch. *arXiv preprint arXiv:2011.09107*, 2020.

[5] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From Wisckey to Bourbon: A learned index for log-structured merge trees. In *USENIX OSDI*, 2020.

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

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. ALEX: An updatable adaptive learned index. In *ACM SIGMOD*, 2020.

[8] The Linux Foundation. Kubernetes. https://kubernetes.io/docs/concepts/services-networking/network-policies/, 2021.

[9] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.

[10] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[11] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. Linnos: Predictability on unpredictable flash storage with a light neural network. In *USENIX OSDI*, 2020.

[12] Danny Yuxing Huang, Ken Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *ACM HotSDN*, 2013.

[13] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostic, and Gerald Q. Maguire Jr. What you need to know about (smart) network interface cards. In *PAM*, 2021.

[14] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A single-pass learned index. *arXiv preprint arXiv:2004.14541*, 2020.

[15] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A learned database system. In *CIDR*, 2019.

[16] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *ACM SIGMOD*, 2018.

[17] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. A scalable learned index scheme in storage systems. *arXiv preprint arXiv:1905.06256*, 2019.

[18] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. CutSplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM*, 2018.

[19] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *ACM SIGCOMM*, 2019.

[20] NVIDIA Networking (Mellanox). OVS offload using ASAP^2 direct. https://docs.mellanox.com/pages/viewpage.action?pageId=39264792, 2020.

[21] NVIDIA. NVIDIA BlueField-2x AI-Powered DPU. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2021.

[22] 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. The design and implementation of Open vSwitch. In *USENIX NSDI*, 2015.

[23] A Linux Foundation Collaborative Project. Open vSwitch. https://www.openvswitch.org/, 2020.

[24] A Linux Foundation Collaborative Project. Open vSwitch and OVN 2020 fall conference. https://www.openvswitch.org/support/ovscon2020/#D4, 2021.

[25] The DPDK Project. DPDK - data plane development kit. https://www.dpdk.org, 2020.

[26] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. In *ACM SIGCOMM*, 2020.

[27] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. *IEEE/ACM Transactions on Networking (TON)*, pages 1–15, 2021.

[28] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, 2003.

[29] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. Running neural networks on the NIC. *arXiv preprint arXiv:2009.02353*, 2020.

[30] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *ACM SIGCOMM*, 1999.

[31] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: A scalable learned index for multicore data storage. In *ACM PPoPP*, 2020.

[32] David E Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[33] David E Taylor and Jonathan S Turner. ClassBench: A packet classification benchmark. *IEEE/ACM transactions on networking (TON)*, 15(3):499–511, 2007.

[34] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch dataplane ten years later. In *ACM SIGCOMM*, 2021.

[35] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing packet classification for memory and throughput. In *ACM SIGCOMM*, 2010.

[36] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *USENIX OSDI*, 2020.

[37] WIDE MAWI WorkingGroup. Measurement and analysis on the wide internet (MAWI). http://mawi.wide.ad.jp/mawi/, 2020.

[38] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *ACM SIGCOMM HotNets Workshop*, 2019.

[39] Sorrachai Yingchareonthawornchai, James Daly, Alex X Liu, and Eric Torng. A sorted-partitioning approach to fast and scalable dynamic packet classification. *IEEE/ACM Transactions on Networking (TON)*, 26(4):1907–1920, 2018.

A Appendix

A.1 Approximate sampling

We show how to analytically calculate the expectation μ and standard deviation σ of a uniform sampling of an RQ-RMI neural-net input domain. We use the definitions and notations from [26].

RQ-RMI models contain several stages of *submodels* (neural-networks). In each stage, a single submodel is selected based on the output of the previous stage [26]. Let m be an RQ-RMI submodel.

The responsibility R_m of m is defined as the set of all values in $\mathbb{R}$ that might reach m as inputs, formally $I_1 \cup ... \cup I_n$, where $n \geq 1$ and $I_i = [a_i, b_i]$ are sorted non-overlapping intervals in $\mathbb{R}$.

For $1 \leq i \leq n$, define t_i as the sum of all weighted averages of I_j, $1 \leq j \leq i$. For ease of notation, $t_0 = 0$. Note that the intervals $[t_{i-1}, t_i) \subseteq [0,1]$ do not overlap, and their location in $[0,1]$ is relative to the weighted average of I_i.

For all $1 \leq i \leq n$, define the linear function $g_i(z) : [0,1] \rightarrow R_m$ as follows:

$$g_i(z) = \frac{b_i - a_i}{t_i - t_{i-1}} \cdot (z - t_{i-1}) + a_i$$

In particular, $g_i(z)$ maps between the weighted average of I_i in $[0,1]$ to $I_i = [a_i, b_i]$. The complete mapping between $[0,1]$ to R_m can be described as the collection of all g_i functions, or as follows:

$$g(z) = \left\{ g_i(z) \middle| z \in [t_{i-1}, t_i), 1 \leq i \leq n \right\}$$

Given a uniform random variable $z \sim U[0,1]$, the expectation μ and variance σ^2 of R_m can be described using $g(z)$:

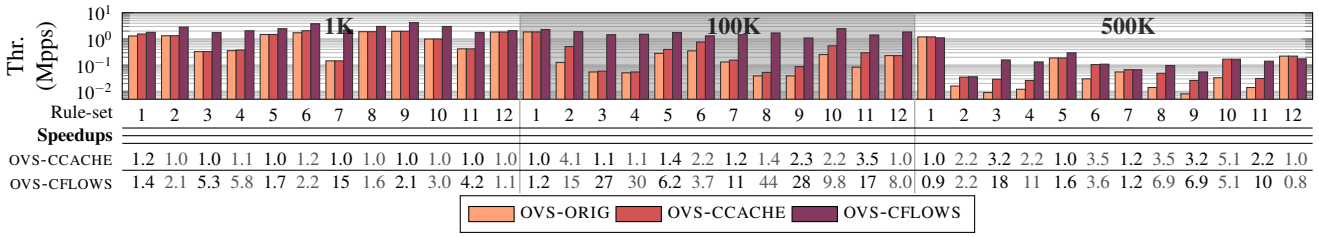
$$\mu = \mathbb{E}[g(z)] \quad \sigma^2 = \mathbb{E}[g(z)^2] - \mathbb{E}[g(z)]^2$$

The two can be manually calculated from the equations above.

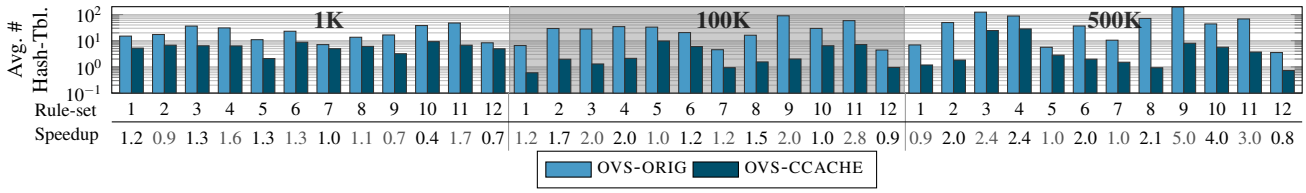
A.2 More on updates in OVS-CCACHE

We test the temporal behavior of OVS-CCACHE when facing upcalls and different update rates, similar to the analysis presented for OVS-ORIG (§3). Since we cannot control OVS-CCACHE update rate (§7), we artificially delay adjacent NuevoMatchUP training sessions. We use the same rule-set and trace as in Figure 4, and sample the system's throughput, number of upcalls, and NuevoMatchUP iSet coverage, each 100ms.

The results shown in Figure 19 emphasize the importance of fast updates in OVS-CCACHE, as frequent upcalls cause the iSet coverage to drop to zero after just a few seconds, cutting the throughput by half ($t = 5$ sec). Note that the slow throughput of the system causes it to effectively digest the input at a lower rate, which in turn causes the upcall rate to go down as a result.



(a)



(b)

Figure 18: **(a)** OVS-CFLOWS, OVS-CCACHE and OVS-ORIG on CAIDA-short using adaptive TX rate. Higher is better. **(b)** The average number of hash-tables in the megaflow cache on CAIDA-short trace. Lower is better. This is an extended version of Figure 11.

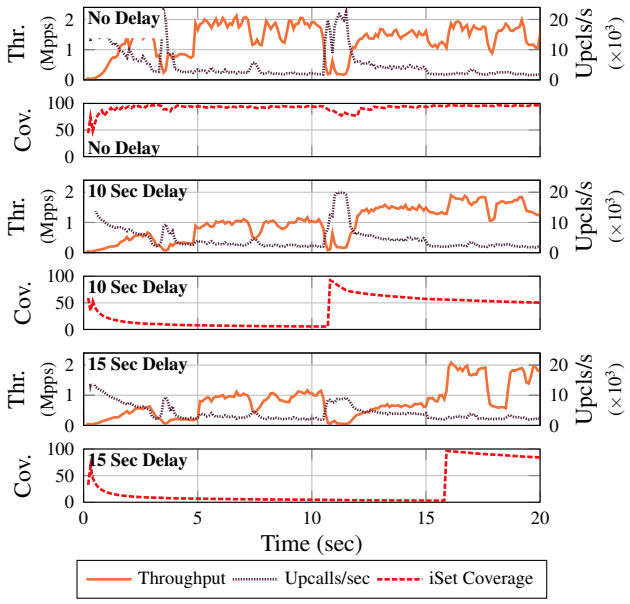


Figure 19: The effect of upcalls and NuevoMatchUP update rate on OVS-CCACHE throughput.

The results also show that upcalls still dominate the throughput as in OVS-ORIG ($t = 11$ sec), thus paving the motivation for OVS-CFLOWS.

A.3 Rule-set names

Rule-set names in Figures 7c, 8, 11, 15, 18a, and 18b by order: ACL1, ACL2, ACL3, ACL4, ACL5, FW1, FW2, FW3, FW4, FW5, IPC1, IPC2.