Flow Caching for High Entropy Packet Fields

Nick Shelly
Stanford University
nshelly@cs.stanford.edu

Ethan J. Jackson
VMware, Inc.
jacksone@vmware.com

Teemu Koponen
VMware, Inc.
tkoponen@vmware.com

Nick McKeown
Stanford University
nickm@cs.stanford.edu

Jarno Rajahalme
VMware, Inc.
jrajahalme@vmware.com

ABSTRACT

Packet classification on general purpose CPUs remains expensive regardless of advances in classification algorithms. Unless the packet forwarding pipeline is both simple and static in function, fine-tuning the system for optimal forwarding is a time-consuming and brittle process. Network virtualization and network function virtualization value general purpose CPUs exactly for their flexibility: in such systems, a single x86 forwarding element does not implement a single, static classification step but a sequence of dynamically reconfigurable and potentially complex forwarding operations. This leaves a software developer looking for maximal packet forwarding throughput with few options besides flow caching. In this paper, we consider the problem of flow caching and more specifically, how to cache forwarding decisions that depend on packet fields with high entropy (and therefore, change often); to this end, we arrive at algorithms that allow us to efficiently compute near optimal flow cache entries spanning several transport connections, even if forwarding decisions depend on transport protocol headers.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Networking

Keywords

Packet classification, caching

1 Introduction

Network virtualization and network function virtualization (NFV) have been hot topics in recent networking trade press and indeed, the coverage is well deserved: the majority of service providers are already at least planning network virtualization and NFV deployments and a significant number have already either, or both, in testing or production. While the exact definitions of network virtualization and NFV are not universally agreed upon, most definitions emphasize the role of the network edge in providing the complex packet processing functions: forwarding elements in the network core are left with high speed, low latency, and power efficient transportation of packets from a network edge to another. The edge provides the complex packet operations implementing network services and policies relevant for end hosts.

Packet processing built on generic CPUs allows a single server to execute an arbitrary sequence of complex packet processing operations for a packet. Without leaving the server, a packet may traverse a simple *chain* of services (in the case of NFV) or a more complex logical *topology* of switches, routers, and services (in the case of network virtualization). This flexibility combined with the steadily improving efficiency of x86 has made software forwarding at the network edge not only practical but attractive with network virtualization and NFV.[1]

Despite the impressive advances in software packet forwarding, any classification beyond destination address matching remains challenging for x86 hardware; the lack of TCAMs necessitates algorithmic classification which remains difficult to optimize for cache hierarchies of general CPUs. This difficulty combined with network virtualization and NFV, translating to multiple classification steps for a single packet on an x86 server, has forced developers to look for non-algorithmic ways to limit the cost of packet classification.

Unsurprisingly, developers have focused on an old trick, flow caching. For example, since its first version, Open vSwitch (OVS) has implemented an exact match-based connection cache. For the first packet of the transport connection, the system executes sequential, flow classification steps in a *slow path*, and caches the exact match result. Then, subsequent packets are handled in the *fast path* through a hash lookup, based on all header fields. In the slow path there is a worst case lookup time of $O(n)$ where n is the number of rules, whereas in the fast path run time is of complexity $O(1)$. Similar mechanisms are commonplace on x86 based appliances. For example, modern middleboxes often use caching to accelerate processing of payload packets after seeing the HTTP request. While a significant improvement, connection cache still requires the slow path to be involved in each new transport connection, even if the resulting forwarding decisions were similar across multiple transport connections. For instance, multiple client connections towards the same HTTP server IP address and port from a single client IP (but with varying source ports) require separate cache entries when using an exact match-based connection cache.

[1]Today L2 switching and L3 routing of minimum sized packets at 10G consumes only a fraction of a modern x86 CPU core [6, 11].

HotSDN '14 August 18, 2014, Chicago, Illinois USA

http://dx.doi.org/10.1145/2620728.2620755.

As a further improvement, one can limit flow cache entries not to match beyond L2 and L3 header fields. As one example, OVS implements "megaflows" for these exact use cases and reverts to exact match-based connection caching only if forwarding decisions on the slow path depend on L4 headers. As long as the matching is based on the L2 and L3 headers only, the computed cache entries do not require per transport connection updates and fast path can process new connections without punting packets to the slow path.

However, practical forwarding decisions increasingly depend on L4 headers. Almost all network services operate over L4 headers: a firewall service might allow all traffic except SMTP. Similarly, HTTP traffic might require redirection to a caching proxy while the other traffic bypasses the proxy. This leaves exact match-based connection caching as the only practical flow caching option.

In this paper, we consider the problem of computing flow cache entries for forwarding decisions that depend on high entropy packet fields. We loosely define high entropy packet fields as those which are likely to have differing values from packet to packet flowing through a switch. For example, all traffic originating from a particular host will likely have the same source and destination MAC fields, but the source and destination L4 port fields are likely to change from connection to connection. Thus, we say the L4 port fields are higher entropy than the L2 address fields.

Our primary focus here is on TCP/IP and hence on the relatively high entropy transport port fields, but the proposed algorithms are more general and applicable to future protocols. Our goal is to compute flow cache entries that represent larger traffic aggregates than individual transport connections. In the firewall example above, cached entries for SMTP traffic would still match the exact port number (25) while the rest of entries would wildcard the port number bits to the extent possible without changing the overall forwarding semantics. Since the computation is done for potentially smaller traffic aggregates than achievable at L2 and L3, the computation has to be cheap enough to remain practical.

We consider three approaches to compute the flow cache entries. First, we consider the ideal case and compute the cache contents *proactively* by taking the cross-product of flow tables that packets traverse within the slow path. For small forwarding tables, computing the cross-producted flow table is manageable. However, for a large number of flows, the cross product can grow non-polynomially.

The second case we consider involves on-demand, *reactive* algorithms, where we compute an entry in the cache based on the packet received. To compute a flow cache entry, for each flow table traversed in the slow path we must subtract the header space matching higher priority flows from lower priority flows. Using header space analysis, the complement of the group of higher priority flows can be resolved to a union, which we wish to intersect with the packet to determine the packet header field bits that can be wildcarded in the cached entry.

As a third approach, we present several heuristic algorithms for computing the flow cache entries, including a method that finds common matches amongst a union of rules and that differ from the packet, and a method that uses a longest prefix match. We compare the algorithms using customer production rule sets matching over transport protocol ports, both with a custom simulator as well as with an implementation in OVS. We examine *cache expansion* without making any assumptions about traffic properties and thus real-world cache hit rates. In other words, we measure resulting cache sizes in the worst case, when there is network scan on a high entropy field (*e.g.,* 2^{16} entries for a scan on L4 destination ports). Handling low entropy traffic would be trivial, as the cache could easily handle all of the traffic with a small number of entries.

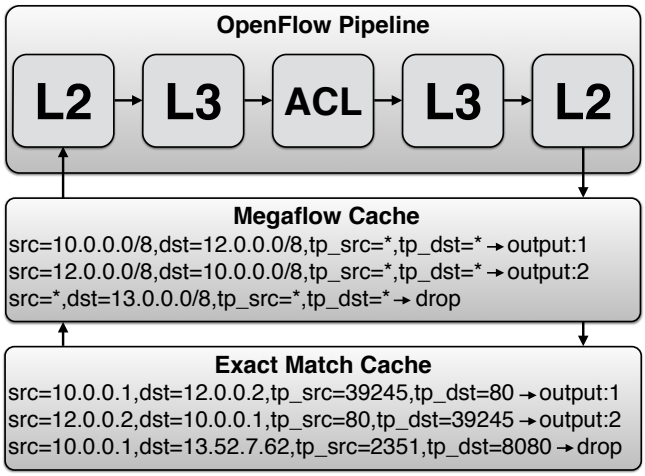

Figure 1: Cache hierarchy in OVS. Packets first check the exact match cache, then the megaflow cache, before finally traversing the OpenFlow pipeline.

This paper is structured as follows. In the next section, we briefly review the related work. In Section 3, we consider the complexity of computing the ideal flow cache entries. In Section 4 we lay out several practically feasible heuristic algorithms for computing the flow cache. Finally, in Sections 5 and 6 we present micro-benchmarks for our algorithms and conclude the paper.

2 Background

Packet classification. The literature on algorithmic packet classification is vast and we only cover the state of the art here.

Longest prefix matching algorithms for a single field can provide fast lookup for searches based on a contiguous prefix and in a runtime proportional to the number of bits [10]. In packet classification over multiple packet fields, decision tree classifiers are the current state of the art. Early decision tree classifiers focused on minimizing memory accesses required per classification, and thus, suffered from high memory consumption due to rule duplication [7]. EffiCuts [9] uses equi-sized cuts and introduced the idea of "separable" trees to address the variation in the rule-space density to limit the duplication and hence the memory consumption.

Decision tree classifiers come with complex tree update logic and in practice developers may favor simplicity over optimality. For instance, Open vSwitch uses a modified version of Tuple Space Search [8] exactly because of its ability to support simple constant time updates, in contrast to a decision tree classifier. With tuple space search, a classifier update is an update to a hash table.

Flow caching. Flow caching and separation of the data plane into a "fast path" and a "slow path" have a long history in networking [4] and over the years advocates have argued for network traffic having sufficient localities to provide high cache hit rates with relatively small cache sizes [1, 3].

Many modern software forwarding products rely on flow caching. For instance, Open vSwitch recognized early the need for a slow path for complex forwarding logic (in userspace) and a fast path for simple forwarding logic merely replicating decisions of the slow path (in kernel) [5]. Originally, the Open vSwitch flow cache was exact match. Upon entering the kernel, a packet's headers would be looked up in a simple hash table using all the packet fields as a key. If present, the cache returned instructions for forwarding the packet. If not present, the packet would be sent to a slow path in userspace

which executed the full OpenFlow pipeline, possibly consisting of several packet classifications (in the form of multiple OpenFlow tables traversed). The slow path would cache the decision in the kernel's fast path for subsequent packets of the transport connection.

The current Open vSwitch architecture is depicted in Figure 1. Just as before, packets entering the system are first checked against an exact match cache. Packets which miss the exact match cache are punted up to a megaflow cache. The megaflow cache provides the same simplified semantics as the exact match cache, with the additional ability to support arbitrary bitwise matching on any packet header field. Packets which miss the megaflow cache are sent up once more to the OpenFlow pipeline provided by an SDN controller. The OpenFlow pipeline supports a wide range of complicated operations defined by the OpenFlow standard and various OVS extensions. These features, such as arbitrary packet metadata, multiple packet classification tables, and arbitrary recursion, are significantly more complicated and computationally expensive than the basic packet classification provided by the megaflow and exact match caches.

Given the packet forwarding performance difference between the megaflow cache and OpenFlow pipeline, population of the megaflow cache with "good" entries (that cover large traffic aggregates) can have a significant impact on the overall performance of the switch. Initially Open vSwitch implemented a naïve algorithm to populate the megaflow cache. Any bit not matched by the OpenFlow pipeline would be marked as wildcarded in the final megaflow cache entry. This algorithm works fine if the slow path exclusively matches low-entropy packet fields, such as L2/L3 addresses. Not wildcarding these fields has a low probability of creating excessive cache misses (since a single IP address probably originates several transport connections). However, if the slow path matches high entropy fields like port numbers, the megaflow cache may reduce to an exact match connection cache. In this paper, we consider tackling exactly this problem: how to compute megaflow cache entries such that we don't converge to exact match connection caching even if the slow path operates over L4 headers.

Theoretical underpinnings. Throughout this paper, we use header space analysis [2] to provide the arithmetic for our reasoning about flows, including determining the intersection, union and complements of packets and rules. The work addresses the difficulty in minimizing header space unions to their minimal representations, which require Karnaugh Maps or Quine-McCluskey algorithms.

3 Ideal Flow Cache

3.1 Cache Population

First, we consider the complexity of computing the ideal fast path, a pre-populated cache table which never results in a miss. Here, we assume the slow path is a pipeline of flow tables (supporting wildcards and priorities) whereas the fast path implements the flow cache with a single flow table (supporting wildcards but not priorities, as in OVS). To completely avoid cache misses, the slow path must translate the slow path table pipeline into a single flow table with equivalent forwarding semantics and push that into cache.[2]

[2] We do not distinguish between packet field types but consider the general problem. Neither do we consider representing the fast path flow cache as a pipeline of multiple smaller tables: while it could reduce the total number of flow cache entries required, it would introduce additional computational load. We leave exploring this trade-off between increased computational load and reduced memory footprint as a future work.

Rule	Priority	Match	Actions
1	1000	tp_dst == 80	drop
2	1000	tp_dst == 443	drop
3	500	dl_src == 01:xx:xx	reg1 <= reg5, reg4 <= 1, goto next table

Table 1: Simple ACL table with priorities.

Rule	Priority	Match	Actions
1	1000	reg4 == 1	output:2

Table 2: A flow table sending packets out after ACL processing.

To arrive at this single classification table, we *proactively* cross-product the tables in the slow path packet processing pipeline. This expanded table can be derived through the following algorithm:

1. Create an all-wildcards header space and send the header space to the initial forwarding table.
2. Intersect the current header space for each rule that matches, by creating a new header space and applying the rule's action. Subtract any higher priority rules from this header space.
3. If there is an output action, add the input header space and actions to the combined table. Otherwise, send the intersected header space to the next table.

As an example, we consider the two simple flow tables provided in Table 1 and 2. The first table has a few ACLs while the second table holds nothing but a single output rule after the ACL processing. So the header space input to Rule 3 is all-wildcards intersected by its rule, minus the header space of transport destination ports 80 and 443:

$$\{\text{all-wildcards}\} \cap \{\text{dl\_src} == 01\text{:xx:xx}\} - \{\text{tp\_dst} == 80 \cup \text{tp\_dst} == 443\}$$

We apply the actions for Flow 3 to this header space. At each step, we have both an *input header space*, which is the set of all packets that can arrive at a given rule, and an *output header space* which is the header space after the rule's actions are applied to the input header space. The output header space must take into account wildcards that could be shifted through registers, such as REG5 into REG1 in Table 1. Essentially, the header space of lower priority rules becomes the union of higher priority flows subtracted from the initial all-wildcards header space.

Thus, after computing the cross product of these forwarding tables, we have one full forwarding table strictly defined by header spaces and actions, shown in Table 3. To use the resulting forwarding table with a fast path classifier, the resulting header spaces have to be further translated to unions of header spaces (each corresponding to a single rule) through header space arithmetic.

While the proactive, cross-producting algorithm is useful in understanding the ideal cache table, it is impractical due to flow table expansion. The table size can grow in polynomial time with the number of rules, as shown in Figure 2.

Match	Actions
tp_dst == 80 ∪ 443	drop
dl_src = 01:xx:xx −{ tp_dst == 80 ∪ 443 }	reg1 <= reg5 reg4 <= 1, output(2)

Table 3: The resulting cross-producted flow table.

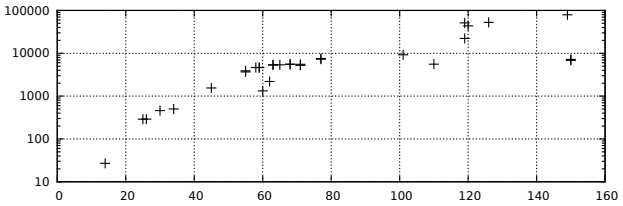

Figure 2: The # of flow entries in the cross-producted flow table as a function # of input flow entries. Input flows are from NVP and implement a single logical datapath of varying complexity. Pipeline is 10-15 stages long.

3.2 Incremental Population

Given the cost of the proactive approach, we now consider how to incrementally compute the cross-product table, based on packets received. The intuition is similar to the general algorithm, but on a per-flow basis when a packet traverses the pipeline of tables:

1. When the packet arrives in the first table, initialize a flow header space to be all-wildcards.
2. Subtract all higher priority rules that fail to match the packet, and intersect the flow header space with the rule that does. Apply the actions of the matched rule to the packet and the flow header space.
3. If forwarding the packet, submit it and its flow header space to the next table. Repeat Step 2, by further subtracting higher priority flows and applying matched rules, until we either drop the packet or have a final output action.
4. Add a rule to the cache that matches the entire flow header space. Logically, the processed packet is an element of this flow header space.

For example, suppose we have a forwarding table with ACLs, $A = 0101$ and $B = 0110$ (without specifying which fields bits correspond), that drop all packets, and a lower priority general rule $C =$ xxxx that matches all packets and forwards on Port 2:

$A = 0101$	drop
$B = 0110$	drop
$C =$ xxxx	output:2

Assuming the incoming packet matches C, we now wish to compute the general rule to install. As discussed above, this corresponds to the header space $h_c = C - A - B$. We wish to determine a general rule for a given packet that is the most general subset of h_c. This can be derived through header space algebra by evaluating the intersection of the complement of higher priority flows, B and C, and distributing the union over the intersection:

$$\begin{aligned}
h_c &= C - A - B \\
&= C \cap A' \cap B' \\
A' \cap B' &= (0101)' \cap (0110)' \\
&= (1\text{xxx} \cup \text{x0xx} \cup \text{xx1x} \cup \text{xxx0}) \\
&\quad \cap (1\text{xxx} \cup \text{x0xx} \cup \text{xx0x} \cup \text{xxx1}) \\
&= 1\text{xxx} \cup \text{x0xx} \\
&\quad [(\text{xx1x} \cup \text{xxx0}) \cap (\text{xx0x} \cup \text{xxx1})] \\
&= 1\text{xxx} \cup \text{x0xx} \\
&\quad \cup [\text{xx1x} \cap (\text{xx0x} \cup \text{xxx1})] \\
&\quad \cup [\text{xxx0} \cap (\text{xx0x} \cup \text{xxx1})] \\
&= 1\text{xxx} \cup \text{x0xx} \cup \text{xx11} \cup \text{xx00}
\end{aligned}$$

For a packet of $p = 1011$ to match h_c above, we intersect the packet, P_s, with the above sets for $A' - B'$, which results in 1xxx, x0xx, or xx11:

$$\begin{aligned}
P_s(p) - A - B &= P_s(p) \cap A' \cap' B' \\
&= P_s(1011) \cap (1\text{xxx} \cup \text{x0xx} \cup \text{xx11} \cup \text{xx00}) \\
&= 1\text{xxx} \cup \text{x0xx} \cup \text{xx11}
\end{aligned}$$

While it is easy to express this header space with logic, minimizing the set of a non-polynomial number unions is an NP-hard problem. Furthermore, we only wish to install one rule per packet for simplicity: the one with the fewest number of un-wildcarded bits. In the general case, for each packet of size L, there are $2^L - 1$ possible wildcard expressions that match the packet, based on which k bits are un-wildcarded. For packet $p = 1011$ we have $\binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{1}$ possible cache entries we could install, depending on the subtracted higher priority flows:

$$\begin{aligned}
P_s(1011) = \{&1\text{xxx}, \text{x0xx}, \text{xxx1}, \text{xxx0}, \\
&10\text{xx}, \text{x01x}, \text{x01x}, \text{xx11}...\}
\end{aligned}$$

The total number of possible flows that include the packet are:

$$|P_s(p)| = \sum_{k=1}^{L} \binom{L}{k} = 2^L - 1$$

Thus, we turn to heuristics in the following section to find the most general rule to install.

4 Heuristic Algorithms

We aim to cache a rule that matches the packet and a broad range of other similar packets, but does not match any of the higher priority rules. Caching an exact match on high entropy packet fields is not useful, as the next packet will likely differ slightly and need to be re-processed. Due to the difficulty of minimizing the union of a non-polynomial number of sets, our goal is to achieve near optimal performance by just un-wildcarding a few bits that match the flow.

Common match. The first heuristic we employed was finding a "common match" of higher priority rules, and intersecting its complement with the packet. This algorithm is of constant time if there exist common bits for the higher priority flows, but fails to find a subset if there are no common matches that differ from the packet. Here, we un-wildcard the most significant bit (e.g. often a packet will have a TCP destination port > 10,000 while the ACLs block certain ports less than 10,000).

The un-wildcarding of 1 bit can be performed in $O(1)$ time for each new packet, after an initial processing of rules in $O(N)$ time. In fact, if there are bits common to each of the higher priority rules, we can install the most general rule possible that matches the flow.

PROOF. Suppose there is no common bit shared by all of the higher priority rules; that is, all higher priority rules cover all possible bits at each location in the match. However, there exists an optimal general rule, P^*, that has only 1 bit un-wildcarded for a packet that does not match any of the higher priority flows (*e.g.,* a rule matching x1xx or xxx0). This means, for all higher priority rules, $\{r_0, r_1, ...r_n\}$, $P^* \cap r_i = \emptyset$. Thus, the inverse of the bit is shared by all of the higher priority rules (*e.g.,* x0xx or xxx1, respectively), which is a contradiction. □

Multi-bit common match. In the second heuristic algorithm, we expand the common match on a per packet basis.

1. Begin with an array of arrays, `common_match_array`, where each element is a bit array representing the common match of a set of rules. Initialize `common_match_array` with a single common match bit array representing an arbitrary higher priority rule.

2. For each higher priority rule, successively iterate over each common match in `common_match_array`.

   (a) If common bits exist between the rule and a common match, update the common match and continue to the next rule.

   (b) If the rule shares no common bits with any of the common matches, append a new common match containing only this rule to `common_match_array`. Continue to the next rule.

3. By the end of examining the higher priority rules, each element of `common_match_array` now matches a section of the rules and differ from the packet. Un-wildcard one bit from each of these common match bit arrays to produce a flow that is unique to the packet but differs from all higher priority flows.

This process is linear time, $O(kn)$ where k is the number of bits and n is the number of rules. While this solution works well for most use cases, it requires more processing time and does not always provide the optimal rule (*i.e.,* it could return a mask with 5-bits un-wildcarded instead of a more optimal one with 4).

Longest prefix match. As a third algorithm, we build a decision tree that contains all of the higher priority rules as leaves, segmenting children based on '1' or '0'. When we classify a new packet, we traverse the tree, un-wildcarding bits along the way, starting with the root until we get to a branch with no leaves. This algorithm is looking for a subset of the union, by narrowing the search space by only looking at prefixes and runs in constant time, $O(k)$, where k is the number of bits. However, it does not always identify the broadest header space, as it only wildcards contiguous bits, excluding valid masks such as x1x1.

5 Evaluation

5.1 Datasets

In order to evaluate our heuristics we construct ACL classification tables from the firewalls of two large private cloud providers. We extract from these ACL tables a list of L4 destination ports (and port ranges) which have policies attached to them. Ranges are converted into an equivalent series of masks. After translating the configuration into flows, Provider 1 matches on a total of 1038 ports, while Provider 2 matches on a total of 32 ports. In addition, we supplement this data with random sets of 50 ports generated at each test run.

5.2 Heuristic comparison

To compare our heuristic algorithms, we built a simulator in Python which operates exclusively over the L4 destination ports, as they represented a common classification on high entropy fields. The classifier receives a packet, which is looked up in a megaflow cache, representing the fast path. If no entry exists, it is processed by a provided algorithm that determines which rule to install in the cache for future ports. By systematically sending all possible port numbers through the system, representing "maximum entropy," we can determine the number of megaflows needed to cache a particular port set in its entirety. The results in Figure 3 and Table 4 show that the common match algorithm results in the fewest number of cache entries.

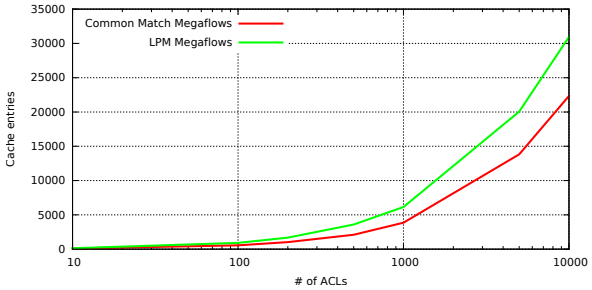

Figure 3: Resulting cache entries for Common Match and Decision Tree (LPM) algorithms.

	Common Match			Decision Tree		
ACLs	Flows	Masks	Approx.	Flows	Masks	Approx.
10	93	76	0.940	126	14	0.159
100	552	435	0.796	913	13	0.134
1000	3860	2669	0.573	6141	9	0.074
5000	13822	504	-	20060	7	-
10000	22333	3563	-	30947	6	-

Table 4: Cached flows, unique masks and the approximation of the optimal flow by two heuristic algorithms, for randomly generated ACL tables.

The longest prefix match also fares quite well, and has an advantage in the number of unique masks.[3] This makes longest prefix match quite attractive for customer data sets in Open vSwitch, where "high entropy" rules are concentrated on one end of the field range, and the hashing for the fast path can be performed on a small number of masks. The results are shown in Table 5.

We also compare the result of each heuristic algorithm against a brute force algorithm, to find the most general rule possible:

1. Un-wildcard b-bits of the packet, where b is initially 1.

2. Check if the flow matches any of the higher priority rules. If it does, we break and try the next un-wildcard sequence, otherwise un-wildcard any $b + 1$ bits.

3. Continue until we find a flow that matches the packet and no higher priority rule.

This brute force algorithm has a worst case performance of $O(n(2^L - 1))$ possible wildcards. However, we can be certain that we will find the best possible rule this way, so it is useful for calculating the approximation of the optimal solution by the heuristic algorithms. The common match algorithm performs well with approximating the optimal solution – the fewest number of bits un-wildcarded – with 80% of the optimal solution for 100 ACLs, as shown in Table 4. The decision tree algorithm, which limits itself to flows defined by contiguous prefixes, caches flows quickly and

[3]In tuple space based flow classification, as in Open vSwitch, the number of unique masks directly determines the number of hash lookups required.

ACLs	C Flows	C Masks	D Flows	D Masks
Provider 1	194	153	178	15
Provider 2	9124	846	3009	12

Table 5: Comparison of cache performance for two private cloud providers.

ACLs	Baseline	Decision Tree	Common Match
Provider 1	16.845	26.085	285.326
Provider 2	8.041	9.404	11.749
Random 50	1.938	2.877	10.902

Table 6: Time (in seconds) required to do 10 million packet classifications.

in a small number of masks, but produces only 13% of the optimal solution's ACLs.

The experiment does not examine hit rate, because this would be dependent entirely on the traffic and the size of the cache. If packets were concentrated on one end of the spectrum (*e.g.,* web traffic), there would be a greater number of cache hits. The worst case would be a network scan across all high entropy fields and a significant number of low entropy fields. For example, this would occur if a malicious user launched a distributed denial of service (DDoS) attack by sending packets from all TCP ports to all ports, from and to multiple IP addresses. In this case, if the slow path used high entropy fields in its forwarding decisions, the system would converge towards installing a flow cache entry per received packet. The solution would be to lower the timeout period for clearing flow cache entries or to replace Least Recently Used (LRU) entries.

5.3 Microbenchmarks

While the heuristic algorithms provide a significant performance benefit from a whole system perspective, they do impose a cost on the raw slow path packet classification performance. To quantify the additional overhead to slow path flow classification, we loaded the reference ACL tables into the Open vSwitch slow path packet classifier enhanced with our various heuristic algorithms. Datasets loaded, we recorded the time it takes to perform 10 million packet classifications on randomly generated packets. This represents the worst case overhead. The results shown in Table 6 suggest that for a modest number of ACLs, both the decision tree and common match algorithms perform reasonably. For Provider 1's dataset, the common match performance is quite a bit slower due to its $O(n)$ time complexity. However, this still may be acceptable given the caching benefits it enables.

6 Conclusion

In this paper, we considered the problem of computing flow cache entries for a slow path operating over high entropy packet fields, such as transport protocol port numbers. After ruling out the ideal, proactive header space based algorithms as too expensive, we developed heuristic, reactive algorithms that provide near optimal results with limited overhead in the slow path. The decision tree algorithm produces a flow cache with a small number of unique flow masks, despite being suboptimal compared to the common match algorithm in identifying the most general flow cache entries. However, in Open vSwitch, the decision tree algorithm is preferred because the implemented tuple space search packet classification algorithm is $O(n)$ in the number of masks, and software can handle the resulting larger cache size. In a memory-constrained environment, or with a limited number of rules on high entropy fields, the common match algorithm would be preferred. We have contributed our results to mainstream Open vSwitch and its megaflow implementation now supports flow caching of forwarding decisions including L4 headers, without requiring every new forwarded transport connection to be handled by the slow path.

7 References

[1] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In *Proc. of HotNets*, October 2008.

[2] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. of NSDI*, April 2012.

[3] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting Route Caching: The World Should Be Flat. In *Proc. of PAM*, April 2009.

[4] P. Newman, G. Minshall, and T. L. Lyon. IP Switching - ATM under IP. *IEEE/ACM Transactions on Networking*, 6(2):117–129, 1998.

[5] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of HotNets*, October 2009.

[6] L. Rizzo. Netmap: a Novel Framework for Fast Packet I/O. In *Proc. of USENIX ATC*, June 2012.

[7] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *Proc. of SIGCOMM*, August 2003.

[8] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification using Tuple Space Search. In *Proc. of SIGCOMM*, August 1999.

[9] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *Proc. of SIGCOMM*, August 2010.

[10] M. Waldvogel, G. Varghese, J. S. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proc. of SIGCOMM*, September 1997.

[11] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. of CoNEXT*, December 2013.