Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

CATE: AN OPEN AND HIGHLY CONFIGURABLE FRAMEWORK FOR PERFORMANCE EVALUATION OF PACKET CLASSIFICATION ALGORITHMS

Wladislaw Gusew
Sven Hager
Björn Scheuermann

Computer Engineering Group
Humboldt University of Berlin
Rudower Chaussee 25
Berlin 12489, GERMANY

ABSTRACT

Network packet classification is the central building block for important services such as QoS routing and firewalling. Accordingly, a wide range of classification schemes has been proposed, each with its own specific set of characteristics. But while novel algorithms keep being developed at a high pace, there barely exists tool support for proper benchmarking, which makes it hard for researchers and engineers to evaluate and compare those algorithms in changing scenarios. In this paper, we present the *Classification Algorithm Testing Environment (CATE)*. CATE consistently and reproducibly extracts the key performance characteristics, such as memory footprint and matching speed, for a predefined set of classification algorithms from a highly customizable set of benchmarks. In addition, we demonstrate that CATE can be used to gain new insights on both the input parameter sensitivity and the scalability of even well-studied algorithms.

1 INTRODUCTION

Packet classification is an essential part of various services in today's packet-switched networks, such as firewalling, policy routing, intrusion detection, and traffic rate limiting (Taylor 2005). In order to fulfill its task, each of these services relies on a classification sub-system, which maps certain header fields of each incoming network packet to a previously defined rule. Once the matching rule for a current packet has been detected, the system knows how to further process the packet. As a simple example, consider a firewall which is used to regulate the traffic entering or leaving a protected subnet. According to the rule set specified by Table 1, the firewall can decide for each inspected packet whether it should be forwarded or discarded.

Although the task of packet classification sounds simple at a first glance, it is generally difficult to classify packets at high rates (Gupta and McKeown 2001). The reason for this is twofold: on the one hand, there may be only a small time frame to process incoming packets in the worst case, e.g., 12.8 ns

Table 1: A simple example for a firewall rule set with four rules.

Rule	Source IPv4-Address	Destination IPv4-Addr.	Protocol	Source Port	Dest. Port	Action
1	10.42.0.1/16	10.42.0.17/32	TCP	*	80	Allow
2	10.42.0.17/32	10.42.0.1/16	TCP	80	*	Allow
3	167.1.0.0/24	10.0.0.0/8	UDP	[1025:65535]	*	Allow
4	*	*	*	*	*	Drop

for a 40 Gbps link. On the other hand, the amount of work per packet increases as the rule set size grows, which, for instance, happens due to an increasing number of hosts in a network (Vamanan et al. 2010).

According to the importance and difficulty of the packet classification problem, over the past years a considerable amount of research on the development and optimization of classification algorithms was conducted (Lakshman and Stiliadis 1998, Srinivasan et al. 1999, Gupta and McKeown 2000, Woo 2000, Gupta and McKeown 2001, Taylor 2005, Kirsch et al. 2010). Each of the proposed algorithms has its own unique set of advantages and drawbacks, and may or may not be applicable for a specific use case which requires packet classification. Therefore, researchers or engineers, who are active in this domain, are faced with the following questions:

- Which concrete algorithm should be selected for a specific target application?
- Do the algorithms scale with the input parameters dictated by the target application, such as rule set sizes, rule set structures, or the number of regarded header fields?
- What happens in case of a major change of a certain input parameter?

In order to answer these questions, we propose the *Classification Algorithm Testing Environment (CATE)*. CATE is a benchmarking framework with the goal to extract the relevant key properties, such as classification performance, memory footprint, and preprocessing time, from a set of regarded classification algorithms on top of general purpose CPUs. This is done by running each regarded algorithm against a customizable set of benchmarks, which exposes those properties and allows for detailed subsequent inspection. Moreover, CATE defines a standard interface for common tasks in packet classification, such as setting rule sets and classifying packet headers, that each regarded algorithm has to provide. Finally, CATE provides a fully scriptable front-end by employing the Lua language as its configuration language. The main contributions of this paper are:

- the introduction of CATE, an extensible and general-purpose benchmarking framework for packet classification algorithms,
- the evaluation of four of the most well-known classification schemes, namely *linear search*, *bit vector search*, *HiCuts*, and *tuple space search* within CATE, and
- new insights into the sensibility of the evaluated algorithms to substantial changes in the input parameters.

CATE is publicly available as open source code on the project's website (CATE Project Website 2015).

The remainder of this paper is organized as follows: Section 2 introduces the packet classification problem and motivates the need for benchmarking tool support. Section 3 describes the architecture of the CATE framework. In Section 4, we briefly outline four of the most well-studied classification algorithms are briefly outlined, which are evaluated in Section 5 with a particular focus on their input parameter sensitivity. Next, related work is reviewed in Section 6, and finally, Section 7 concludes this paper.

2 PROBLEM STATEMENT

In this section we review the packet classification problem, which is defined as follows: let H_j be the domain of possible header values for the j-th header field of a network packet, $j \in \{1,\ldots,d\}$. Given a tuple of *header values* $P = (h_1,\ldots,h_d)$ with $h_j \in H_j$ and an ordered list of rules (which is called *rule set* or *classifier*) $R = (R_1,\ldots,R_n)$, the goal is to determine the smallest index $i \in \{1,\ldots,n\}$ such that rule R_i *matches* on H. Here, a rule R_i consists of d *checks* c^i_j such that $R_i = (c^i_1,\ldots,c^i_d)$. Each check c^i_j is a function that maps elements from H_i to the Boolean space $\{true, false\}$. Rule R_i matches the header values P iff the predicate $c^i_1(h_1) \wedge \ldots \wedge c^i_d(h_d)$ holds. In practice, the checks c^i_j are often simple equality, range, or prefix checks on unsigned integers (Lakshman and Stiliadis 1998, Gupta and McKeown 2001). For example, rule no. 3 in Table 1 is defined by two prefix checks for the IPv4 source and destination fields, one equality check for the transport layer protocol, and two range checks for the port fields (the *wildcard* symbol ”*” means the entire domain for the corresponding field).

The total number of fields per packet d as well as the numerical limits of each field H_i are defined by the application (and, of course, by the used protocols). For the example in Table 1, we have $d = 5$, $H_1 = H_2 = \{0,\dots,2^{32}-1\}, H_3 = \{0,\dots,255\}$, and $H_4 = H_5 = \{0,\dots,2^{16}-1\}$. This is also known as the common *five-tuple* (Waldvogel 2000, Taylor 2005, Ramaswamy and Wolf 2006, Taylor and Turner 2007, Kirsch et al. 2010). In the following, we refer to the packet header structure by the term *field structure*.

The field structure dictated by an application which employs packet classification has a major impact on classification performance, preprocessing time, and storage requirements of classification algorithms (Gupta and McKeown 2001). However, many well-known algorithms were originally only evaluated against the five-tuple structure (Srinivasan et al. 1999, Gupta and McKeown 2000, Singh et al. 2003, Vamanan et al. 2010), which brings up the question how the performance characteristics adapt to changes in the field structure. In fact, it has been observed that some algorithms suffer from severe performance penalties and require significant changes when being applied in the context of the OpenFlow standard, which uses at least twelve different header fields (Stimpfling et al. 2013). The CATE framework proposed in this work can be used to detect such cases, because it provides built-in support for almost arbitrary field structures, as we will demonstrate in Section 5.

3 CLASSIFICATION ALGORITHMS TESTING ENVIRONMENT

The CATE framework is a C++ application with the goal to measure the performance characteristics and the scalability of packet classification algorithms. As such, a packet classification engine is simulated in software, without transmitting data over a virtual or physical network interface. Thus, we intentionally exclude any I/O components from the performed measurements in order to concentrate on only the algorithmic properties of the benchmarked classification schemes. In order to perform a simulation, a user must supply two inputs to CATE: a C++ implementation of the algorithms to be analyzed, and a Lua script that defines (1) which algorithms should be analyzed, (2) which algorithm-specific parameters should be used for each algorithm, and (3) which field structures, rule sets, and header traces should be used for the simulated packet classification process. We intentionally decided for a white-box approach where the algorithm must be provided as source code. We base this decision on the following two facts: first, an unknown, but potentially high fraction of the work performed by a black-box algorithm could be comprised of I/O operations. Second, there would be a communication overhead between the CATE framework and the inspected black box. Thus, it would be necessary to subtract this overhead from the obtained total execution time of such a black box in order to achieve precise classification times. This, however, is impractical, as the overhead can hardly be quantified. Of course, there is also I/O and management overhead in the white-box approach, but in contrast to the black-box approach it can be identified and can almost completely be excluded from the measurement results, as we demonstrate in our evaluation. Furthermore, extracting the exact sizes of the individual data structures used for classification is a very cumbersome task in the black-box scenario because even a rough description of these data structures may not be given.

Figure 1 provides an overview of CATE's core components. For each benchmark execution, CATE's *configuration interpreter* processes the provided Lua script and builds a data structure which is used by the *benchmark executor* in order to set up and run all simulations. In order to measure an algorithm's key characteristics, a *chronograph manager* collects and organizes measured time spans and a *memory trace manager* logs memory allocations and accesses of traced variables during a simulation run. Both is realized by manual code instrumentation, so that the user can decide about granularity and scope of the measurements. While an automatic code instrumentation could also be supported by defining (1) a set of concrete objects for which the memory usage is traced, and (2) certain scopes for which the time spans are measured, this practice would impose restrictions on programmers which might outweigh its benefits. For time duration measuring, dedicated function calls (*start* and *stop*) are inserted in an algorithm's source code. If a variable is enclosed in a code instrumentation construct, memory metering functionality for this variable is provided by utilizing class inheritance and C++ template techniques. Type implementation of the traced variable is therefore enriched at compile time so that the allocation and de-allocation, as well as all reading and writing accesses to the variable can then be logged at runtime. Of course, if time spans and memory usage are measured concurrently, the obtained time durations can turn out to be higher than without activated memory metering. We evaluate and discuss this effect in Section 5.3.

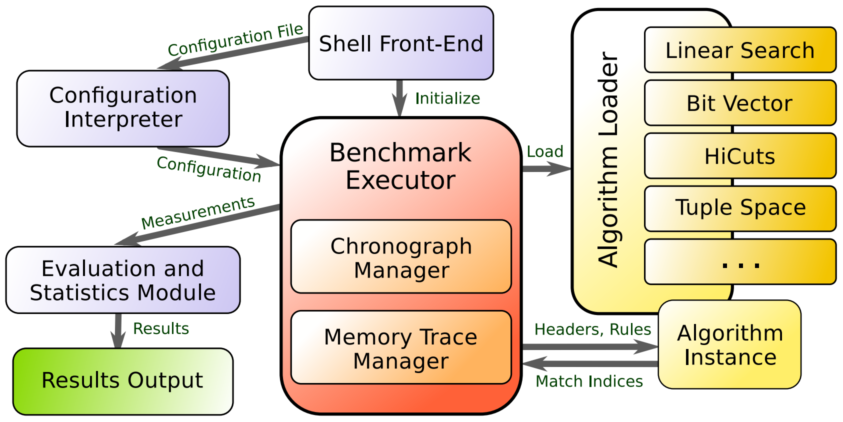


Figure 1: Overview of CATE's software components and their interactions.

The framework itself is separated by design from implementations of classification algorithms. On this account, the object code of each benchmarked classification algorithm is dynamically loaded by the *algorithm loader* module. Therefore, users can add, delete, or modify an algorithm implementation by utilizing CATE's generic data primitives without the need to recompile the entire framework. The connection of the framework to implementations of classification algorithms is an interface with a small number of member functions, namely

- `setRules` for setting a rule set,
- `setParameters` for defining specific algorithm parameters,
- `classify` for classifying packet headers,
- `updateRule` for adding, modifying, or removing a rule during a benchmark run.

Results are collected during all test runs and are evaluated afterwards by an *evaluation and statistics module*. It calculates specific values, for example mean, median, and standard deviation values by taking into consideration all repetitions of the same benchmark. When all simulation runs are completed, documents with summaries of each benchmark are generated, together with raw data to enable a custom analysis of certain characteristics that are not covered by the framework.

In its current version, CATE supports the execution of packet classification algorithms on general-purpose CPUs. Therefore, running the benchmark executor on specialized hardware, such as GPUs or Field Programmable Gate Arrays, is currently not possible. These architectures require very different time and memory measurement techniques. However, in such scenarios, certain CATE modules could still be used to generate test data sets for benchmark execution or to analyze and visualize obtained measurement data.

In Listing 1, we show an exemplary Lua script which could be used in order to benchmark the HiCuts algorithm. Two parameters specific to HiCuts are defined in line 2 and the common five-tuple is specified as the field structure in line 3. Next, in lines 5 to 15, a rule set consisting of two rules which correspond to rules no. 1 and no. 4 in Table 1, is created. In lines 17 to 21, a header trace of two example headers is specified. Finally, the benchmark is registered in line 24.

The main advantage of using Lua lies in variability and power available for the user. For instance, rules or header traces can also be composed programmatically or read from a file. Furthermore, an extensive benchmark can contain many different algorithms, each with its own set of specific parameters, a huge number of rule sets as well as header traces, and different field structures. CATE's scriptable front-end allows to break down this complexity to a single Lua script with a few nested loops.

```
-- Specify HiCuts algorithm with custom parameters (binth and space factor)
alg = createAlgorithm("HiCuts5tpl.so", { 16, 3.0 })
strc = {32, 32, 8, 16, 16} -- bits per field: IP, IP, protocol, port, port

ruleset = createRuleset() -- Specify a rule set
addRuleToRuleset(ruleset, { -- 1st rule
    ruleAtomPrefix(ipv4Toi("10.42.0.1"), maskToi(16)), -- 10.42.0.1/16
    ruleAtomExact(ipv4Toi("10.42.0.17")), -- 10.42.0.17/32
    ruleAtomExact(6), -- TCP
    ruleAtomRange(0, 0xFFFF), ruleAtomExact(80) }) -- ports
addRuleToRuleset(ruleset, { -- 2nd rule
    ruleAtomPrefix(0, maskToi(0)), -- IP src.
    ruleAtomPrefix(0, maskToi(0)), -- IP dest.
    ruleAtomRange(0, 0xFF),   -- protocol
    ruleAtomRange(0, 0xFFFF), ruleAtomRange(0, 0xFFFF) }) -- ports

hdrs = createHeaderset() -- Specify packet headers
addHeaderToHeaderset(hdrs, -- 1st header
  {ipv4Toi("10.42.14.19"), ipv4Toi("10.42.0.17"), 6, 5994, 80})
addHeaderToHeaderset(hdrs, -- 2nd header
  {ipv4Toi("10.42.0.4"), ipv4Toi("10.42.0.17"), 6, 3724, 8080})

-- Combine all elements to a benchmark configuration
registerBenchmark("HiCuts, 10 runs", alg, strc, ruleset, hdrs, 10)
```

Listing 1: Small example of a Lua script in CATE with two rules and two headers.

4 CLASSIFICATION ALGORITHMS

Before we dive into the evaluation of the CATE framework, in this section we briefly familiarize the reader with some of the most prominent packet classification schemes which were analyzed with CATE. As proposed by Taylor (2005), classification techniques can be subdivided in four generic groups, namely *exhaustive search*, *decomposition*, *decision tree* and *tuple space*, as depicted in Figure 2.

Linear Search The linear search is an instance of the exhaustive search techniques and the most straightforward packet classification technique. As the name suggests, this algorithm searches all rules in a rule set linearly according to their priority, until the first matching rule is found. While the search can be parallelized using adequate hardware components, such as GPUs or TCAMs (Taylor 2005, Varvello et al. 2014), a software implementation running on a general purpose CPU has a time and space complexity of $O(dn)$. This linear relation of search time with rule set size has lead to the development of more sophisticated algorithms with better search time complexity, but usually at the price of higher space requirements.

Bit Vector Search The *bit vector scheme* is a decomposition technique, which reduces the d-dimensional packet classification problem to d one-dimensional binary searches, followed by a subsequent combination step (Lakshman and Stiliadis 1998). For each dimension j, each of the binary searches yields a bit vector of n bits. Here, bit i in vector j indicates whether rule i could be a possible match in dimension j. Next, the index of the first matching rule is computed by combining all found bit vectors by bitwise AND operations to a result vector. The index of the first matching rule corresponds to the index of the most significant set bit in the result vector. Although the bit vector search performance is still linear due to the search for the most significant bit, in practice it can exploit bit-parallelism because many operations can be encoded into a single instruction (e.g., the bitwise ANDs) on a machine-specific word width (e.g., 32 or 64). The good search performance comes at the cost of memory requirements which are quadratic in the number of rules. However, we will show in Section 5 that the bit vector space usage tends to be linear if the rule set contains many rules with wildcarded checks.

HiCuts Decision tree algorithms, such as HiCuts, exploit the geometric view of the rule set, which is a collection of d-dimensional hypercubes, and transform the rules into one or several multi-dimensional search tree structures (Gupta and McKeown 2000, Singh et al. 2003, Vamanan et al. 2010). A decision tree

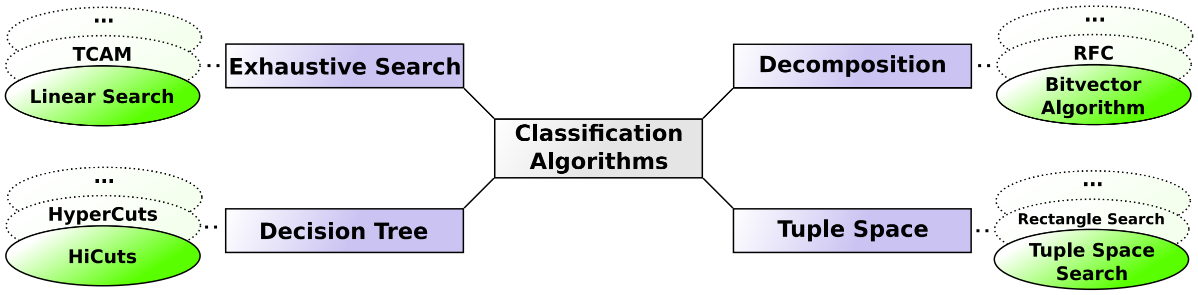


Figure 2: A categorization of classification algorithms (based on Taylor (2005)).

is constructed recursively by firstly creating a single root node which contains the entire regarded rule set. Then, the node is cut along one or several dimensions in order to create child nodes which each contain only a subset of the rule set. This procedure terminates when the number of rules in the current node is lower than a predefined threshold. Given a packet header, a decision tree is traversed until a leaf node is reached, whose rules are then searched linearly. In general, decision tree approaches to packet classification are considered to be extremely fast during packet classification, but also very memory-intensive (Gupta and McKeown 2001, Vamanan et al. 2010). We confirm this in Section 5 for the HiCuts algorithm, which yields the best classification performance, but also the largest and most unpredictable memory requirements in comparison to the other three algorithms regarded in our evaluation. Furthermore, we demonstrate that HiCuts appears to scale very well with an increasing number of regarded header fields, in contrast to every other approach investigated in this work.

Tuple Space Search Tuple space search partitions the rule set based on certain rule properties into a group of equivalence classes, so-called *tuples* (Srinivasan et al. 1999). Each tuple can be searched efficiently with an adequate technique, such as hashing. Hence, the first matching rule can be found by probing each tuple and returning the smallest matching index i. The classification performance as well as the space requirements depend on the concrete search scheme used to probe the tuples. The performance gain of tuple space search is based on the assumption that the number of tuples is far less than the number of rules. In fact, our evaluation confirms that the performance of tuple space search heavily depends on the structure of the rule set.

5 EXPERIMENTS AND EVALUATION

In this section, we present three different simulation scenarios: First, we measure and analyze the probe effect imposed on measured time durations for activated memory usage metering in order to evaluate our proposed framework CATE. Second and third, we examine the influence of changes in the input parameters of the four previously discussed example classification algorithms and obtain new insights into their sensitivity.

5.1 System Setup

All simulation runs were executed on an x86 Intel i5 M430 CPU with 2.27 GHz and Linux 3.13. The operating system runs in a virtual machine (32 bit) to limit the amount of RAM to 1024 MB and to enforce single core operation. To avoid side effects, all other active processes and users were excluded from the system during execution. Even though we took care to exclude external influences as far as possible, system interrupts or intransitive operating system scheduling actions may influence to some extent the measured time durations. Therefore, we configured CATE to repeat time measurements eight times per simulation. The arithmetic mean with a corresponding standard deviation is calculated. Memory results are, in contrast, independent from the number of repetitions, because all four algorithm implementations are strictly deterministic so that no statistical fluctuations can occur.

5.2 Rule Sets and Headers

We use *ClassBench*, as introduced in (Taylor and Turner 2007), to generate synthetic rules for rule sets and the corresponding headers. ClassBench is a tool specialized on the common five-tuple, and is provided with a collection of seed files which contain a sophisticated representation of real rule sets' characteristics. We employ the publicly available seed files `fw1` and `acl1` of firewalls and access control lists (Song, H. 2007), and vary the sizes for the generated rule sets from 200 to 5000 rules in steps of 200. In all simulation configurations, 100,000 headers are classified. In the following subsections, we refer to both types of the generated synthetic rule sets and the corresponding headers by `fw1_`N and `acl1_`N, with $N \in \{200, 400, 600, ..., 5000\}$.

5.3 Measuring the Probe Effect

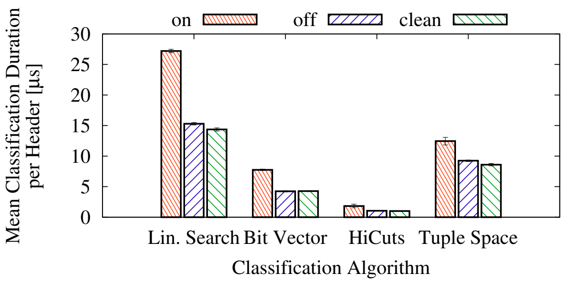
As described in Section 3, the CATE framework is intended to extract the key performance characteristics of classification algorithms. In general, CATE is able to collect three different types of quantities: time spans, memory footprints, and memory access patterns. We found that these three types of quantities are sufficient to measure all common performance characteristics of interest, including classification time, preprocessing time, and memory requirements. However, in order to extract the memory access pattern of a data structure used by a classification algorithm, such as, for instance, a decision tree, it is necessary to instrument the data structure in such a way that it logs memory allocations as well as read and write operations. Unfortunately, this inevitably leads to runtime overhead due to counter updates which, in turn, deteriorates the measurements for time durations, such as classification and preprocessing time. Hence, to achieve the most accurate measurement results, the classification code must be instrumented differently for each regarded performance aspect, which leads to code duplication. For example, in order to measure the most accurate classification time, no logging of memory operations must be performed concurrently and therefore, all memory logging primitives must be removed from the code in advance. While subtracting the duration of all memory operations from the measured overall classification time would lead to equivalent results in theory, this is hard to realize in practice: memory operations are performed very frequently and thus the cumulative time duration of all memory operations would be significantly distorted due to a necessarily limited precision of the time measurements.

CATE solves this problem by supporting a deactivation of logging memory accesses and allocations through a preprocessor directive. Although a small fraction of additional instrumentation code is still compiled into the executable, we will show in our results that the induced overhead is very small when compared to code without any instrumentations for memory measurements. Therefore, CATE enables the usage of the same piece of code for measuring time durations as well as memory access patterns.

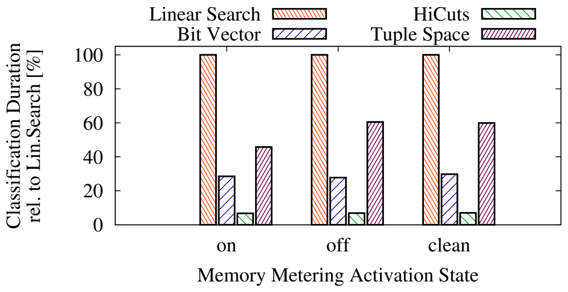
In the remainder of this section, we quantify and evaluate the imposed overhead between the activated memory measuring state (*on*), the disabled mode by a preprocessor directive (*off*), and the *clean* state which is achieved by manually removing all instrumentations for logging memory operations. For each activation state, we will execute the classification of the `fw1_2000` rule set with a five-tuple field structure. Here, we are only interested in the measured classification times and expect to observe highest duration values for activated memory metering activation state. The values for deactivated metering should be lower, and minimum values are expected for time durations when executing cleaned code.

Figure 3a shows the measurements for the mean classification duration of each combination of an algorithm with a memory metering activation state. For all time measurements, we show error bars with 95% confidence intervals. The probe effect is well noticeable when comparing the classification duration measurements for the enabled (*on*) and disabled (*off*) scenario. Between the disabled scenario and the cleaned code, we observe only small differences of at most 0.94 percent.

In addition, Figure 3b shows the classification performance of each regarded algorithm in relation to the linear search, grouped by the activation states. It can be seen that the differences between the performance relations of the disabled (*off*) and cleaned (*clean*) scenarios are negligibly small. Indeed, the



(a) Mean classification durations for different memory metering activation states grouped by the algorithms.



(b) Classification durations relative to the linear search grouped by the activation states.

Figure 3: Measurement results for analyzing the probe effect.

highest observable deviation was only 2.04 percent points for the bit vector scheme. In contrast, there is a clear difference of the relation between tuple space search and the other three algorithms in the *on* scenario, when compared to *off* and *clean* states. In summary, we conclude that the manual effort of cleaning an algorithms implementation code from instrumentation is not rewarded by significantly more accurate time measurements. Hence, in the following sections, we capture memory usage results with activated memory metering functionality and in order to perform time measurements, we disable memory metering by the preprocessor directive.

5.4 Variation of the Rule Set Size

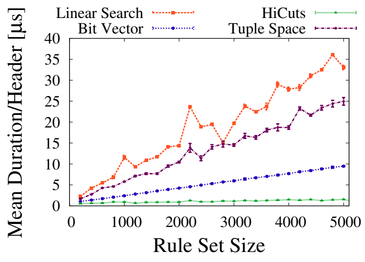
In this section we vary the sizes of rule sets and measure time durations and memory usage in order to compare the performance of all four algorithm implementations. To this end, we use the rule sets `fw1_200` to `fw1_5000` and `acl1_200` to `acl1_5000`. The field structure is the common five-tuple.

As the rule set size increases, more rules potentially need to be checked for each incoming packet header. The search effort therefore generally increases for all of our example algorithm implementations, as the time complexity of each of them depends on the total number of rules in the rule set. We expect to observe this in the data with classification time durations. Also, allocated memory space is expected to increase, for example linearly for linear search and quadratically for the bit vector.

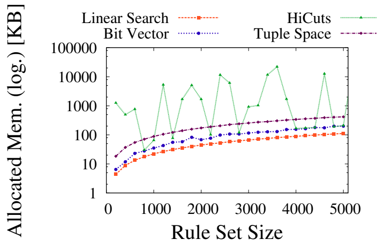
We present the results in Figure 4. In Figure 4a, the mean classification durations per processed header for each algorithm and rule set size for `fw1` are depicted. We show measurements of allocated memory for all algorithms in Figure 4b and to facilitate comparisons between the memory allocations of the linear search and the bit vector algorithm, we depict the results separately on linear vertical axes in Figure 4c. Analogical results for the `acl1` rule sets are presented in Figure 4d to Figure 4f.

Figures 4a and 4d indicate that the linear search performs worst and HiCuts performs best for each rule set size. The bit vector and tuple space search lie in between, with linearly increasing mean classification durations per header for increasing rule set sizes. For the `acl1` rule sets, tuple space search performs far better than for the `fw1` rule sets, as we see in Figure 4d compared to Figure 4a. It can even outperform the bit vector for `acl1`, which is due to a lower number of tuples in the `acl1` rule set structure than in `fw1`. For instance, we counted 26 tuples in `acl1_4000`, in contrast to 420 tuples in `fw1_4000`.

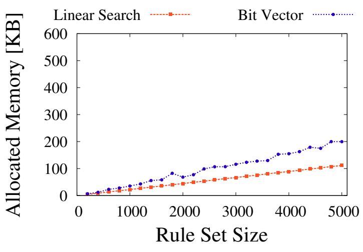
Figure 4b indicates that linear search allocates the lowest amount of memory, as expected. HiCuts has a space usage which is of magnitudes higher than of the other three algorithms, mainly due to its complex decision tree structure and the duplication of rules during the tree construction process. This is visible in Figure 4b as well as in Figure 4e. On the other hand, HiCuts outperforms other algorithms in terms of classification time, as depicted in Figures 4a and 4d. Furthermore, Figures 4b and 4e suggest that the size of the created HiCuts decision trees strongly depends on the structure of the rule set, and that HiCuts' memory allocation is difficult to estimate in advance.



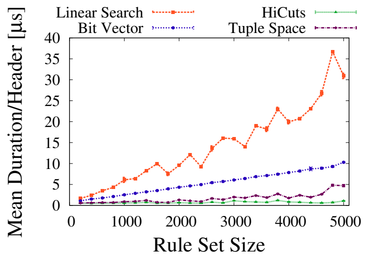
(a) Mean classification duration per header (`fw1`).



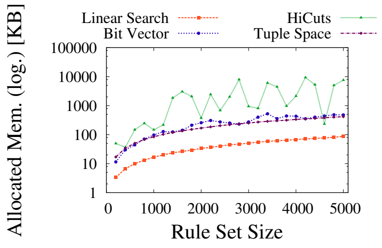
(b) Allocated memory by search structures (`fw1`).



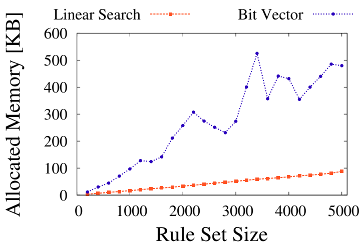
(c) Allocated memory for linear search and bit vector (`fw1`).



(d) Mean classification duration per header (`acl1`).



(e) Allocated memory by search structures (`acl1`).



(f) Allocated memory for linear search and bit vector (`acl1`).

Figure 4: Measurement results for the variation of the rule size.

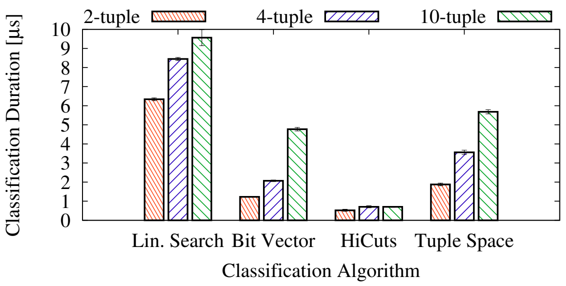
Interestingly, Figures 4c and 4f indicate that memory allocation by the bit vector scheme follows a linear trend, in contrast to the expected quadratic increase. This behavior can be explained by the huge gap in the number of wildcards between the `fw1` and `acl1` rule sets: e. g., `fw1_4000` contains 449 wildcard checks, whereas `acl1_4000` contains only 22 wildcard checks. In fact, the number of wildcards in rules has an impact on the constructed search structure of the bit vector algorithm. The more rules in a rule set have a wildcard in the same field, the lower is the amount of memory allocated, as multiple rules which are wildcarded in the concerning dimension can share the same bit vector. This finding indicates that the rule set structure has a strong influence on the memory footprint of the bit vector algorithm. Thus, a classification realized by the bit vector scheme may exhibit the same space requirements as a linear search if the used rule set contains a high number of wildcards, but offers far better performance.

5.5 Variation of the Field Structure

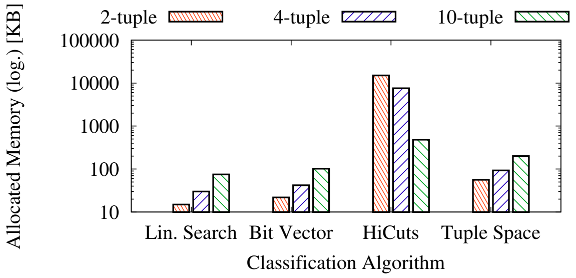
In the previous experiments, we used the common five-tuple as the field structure, according to many previous works in this domain (Srinivasan et al. 1999, Gupta and McKeown 2000, Singh et al. 2003, Vamanan et al. 2010). However, many recent applications demand more complex field structures. For instance, the OpenFlow standard specifies at least twelve regarded header fields (Stimpfling et al. 2013). This opens up the question how existing packet classification schemes scale with an increasing number of relevant header fields. Therefore, we evaluated memory usage and classification performance for three different synthetical field structures, namely a two-tuple with two 32 bit values, a four-tuple with four 32 bit values, and a ten-tuple with ten 32 bit values. The rule sets and header sets were created by the concatenation of multiple ClassBench output files with the `fw1` seed file and a size of 2000 rules.

In general, we expect the classification performance to decrease with increasing number of header fields, as more checks have to be performed for each processed header. Similarly, the memory footprint of the used classification algorithms should increase due to an increase in the number of checks per rule.

The results of this experiment are presented in Figure 5. Figures 5a and 5b show the mean durations per header as well as the memory footprints of the allocated data structures per algorithm for the different field structures, respectively. As expected, the mean classification duration values of all algorithms are lower



(a) Mean classification duration per header.



(b) Allocated memory by search structures.

Figure 5: Measurement results for different field structures (2-, 4- and 10-tuples), grouped by algorithms.

for the two-tuple measurements than for the four- or ten-tuple. However, Figure 5a reveals that HiCuts' classification performance decreases much less than those of the other three algorithms.

When it comes to the memory footprint of the allocated search structures, as shown in Figure 5b, we observe a similar trend: while the memory requirements increase for the linear search, bit vector search, and tuple space search, HiCuts' memory footprint significantly decreases with an increasing number of header fields. The reason for this behavior seems to be that during tree construction, the HiCuts algorithm has more degrees of freedom for choosing the optimal dimension to cut, which leads to a more compact decision tree. Therefore, HiCuts is the only algorithm in our evaluation where we could observe a positive impact on the memory footprint with an increasing size of the field structure.

As far as we are aware, similar investigations of the impact of different field structures on classification performance and space usage have not been done before. We note that the field structure can have great influence on the performance and memory usage characteristics of an algorithm.

6 RELATED WORK

As network packet processing is a performance-critical task in switches, routers, firewalls, and other networked systems, the research community has proposed a large body of benchmarks in this area of application. *CommBench* (Wolf and Franklin 2000), *NetBench* (Memik et al. 2001), and *NpBench* (Lee and John 2003) are benchmarks which aid in the analysis and design of network processors. Similarly, *PacketBench* (Ramaswamy and Wolf 2006) is a benchmark for generic packet processing applications. While these works consider network packet classification as one possible generic use case, they do not address the specific properties of classification algorithms. In contrast, the CATE framework is entirely specialized on the analysis and comparison of classification schemes and thus is able to measure and extract the key performance characteristics of classification schemes in a user-specified resolution. The ability to measure both execution times and memory footprints concisely in an algorithm-centric way distinguishes CATE from general-purpose profiling tools such as `valgrind` (Nethercote and Seward 2007) or `DTrace` (Cantrill et al. 2004). Although these tools can be used for performance and/or memory measurements, it is often hard to clearly map their output data to specific parts of the analyzed classification algorithm.

Of course, many previous works compared different performance aspects of a wide variety of classification algorithms (Singh et al. 2003, Vamanan et al. 2010, Varenni et al. 2005, Sahasranaman and Buddhikot 2001, Stimpfling et al. 2013). In order to do so, the authors had to implement their own specific benchmarking toolchains in order to conduct experiments, which is a tedious and cumbersome work. CATE is intended to help researchers and engineers in such use cases, where the specific characteristics of one or several classification algorithms must be analyzed or compared. It does so by providing a standardized interface for custom algorithm implementations and integrated techniques for measuring the performance of an algorithm. Furthermore, CATE generates summaries with statistical evaluations for each benchmark, as well as it produces data sets with measured raw data for further manual investigations.

The tools which are most closely related to CATE when it comes to packet classification benchmarks are *ClassBench* (Taylor and Turner 2007) and *FRuG* (Ganegedara et al. 2010). Their functionality, however, is orthogonal to CATE, as ClassBench and FRuG generate rule sets which can be used as input parameters to classification algorithms. In contrast, CATE implements a highly configurable runtime environment for the performance analysis of classification algorithms.

7 CONCLUSION

In this paper we presented CATE, a flexible benchmarking framework for packet classification algorithms targeted at general purpose CPUs. CATE allows to precisely measure the key performance characteristics of the evaluated algorithms, such as classification speed, memory usage, and preprocessing time, at a user-specified level of granularity. The benchmark runs performed within CATE are highly customizable because each benchmark configuration is specified as a Lua script. In addition, the CATE framework supports header field structures of an almost arbitrary user-specified format in order to open up a new degree of freedom in the evaluation of classification schemes. The CATE framework as well as an initial library of algorithm implementations are publicly available (CATE Project Website 2015). We plan to extend the algorithm library in the future and openly invite researchers to use the provided infrastructure.

We evaluated CATE by measuring the performance characteristics of four of the most well-studied classification algorithms, namely *linear search*, *bit vector search*, *HiCuts*, and *tuple space search*. During the measurements we put particular emphasis on the sensitivity of those algorithms to changes in the input parameters, such as the structure of the used rule sets and the number of header fields, which lead to interesting results: first, we find that the classification performance of tuple space search is far more sensitive to the rule set structure than the performance of all other discussed algorithms. Second, we measured that the memory usage of the bit vector algorithm decreases with an increasing number of wildcarded fields in the used rule set, which is contrary to HiCuts' space requirements. Finally, we observed that while HiCuts provides the best classification performance at the cost of the highest and most unpredictable space usage, it is the only algorithm in our evaluation whose time and space properties scale well with an increasing number of header fields.

REFERENCES

Cantrill, B., M. Shapiro, and A. Leventhal. 2004, June. "Dynamic Instrumentation of Production Systems". In *USENIX '04*, 15–28.

CATE Project Website 2015. "Open Source code of CATE with User Manual and Examples". http://gusew.github.io/cate.

Ganegedara, T., W. Jiang, and V. Prasanna. 2010, December. "FRuG: A Benchmark for Packet Forwarding in Future Networks". In *IPCCC '10*, 231–238.

Gupta, P., and N. McKeown. 2000, January. "Packet Classification using Hierarchical Intelligent Cuttings". *IEEE Micro* 20 (1): 34–41.

Gupta, P., and N. McKeown. 2001, March. "Algorithms for Packet Classification". *IEEE Network: The Magazine of Global Internetworking* 15 (2): 24–32.

Kirsch, A., M. Mitzenmacher, and G. Varghese. 2010. *Hash-Based Techniques for High-Speed Packet Processing*. Springer.

Lakshman, T., and D. Stiliadis. 1998, August. "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching". In *SIGCOMM '98*, 203–214.

Lee, B. K., and L. K. John. 2003, October. "NpBench: A Benchmark Suite for Control plane and Data plane Applications for Network Processors". In *ICCD '03*, 226–233.

Memik, G., W. H. Mangione-Smith, and W. Hu. 2001, November. "NetBench: A Benchmarking Suite for Network Processors". In *ICCAD '01*, 39–42.

Nethercote, N., and J. Seward. 2007, June. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In *PLDI '07*, 89–100.

Ramaswamy, R., and T. Wolf. 2006, October. "PacketBench: A Tool for Workload Characterization of Network Processing". In *WWC '06*.

Sahasranaman, V., and M. M. Buddhikot. 2001, November. "Comparative evaluation of software implementations of layer 4 packet classification schemes". In *ICNP '01*, 220–228.

Singh, S., F. Baboescu, G. Varghese, and J. Wang. 2003, August. "Packet Classification Using Multidimensional Cutting". In *SIGCOMM '03*, 213–224.

Song, H. 2007. "Evaluation of Packet Classification Algorithms". http://www.arl.wustl.edu/~hs1/PClassEval.html. Publicly available rulesets, last access: June 23, 2015.

Srinivasan, V., S. Suri, and G. Varghese. 1999, August. "Packet Classification Using Tuple Space Search". In *SIGCOMM '99*, 135–146.

Stimpfling, T., Y. Savaria, A. Béliveau, N. Bélanger, and O. Cherkaoui. 2013, June. "Optimal Packet Classification Applicable to the OpenFlow Context". In *HPPN '13*, 9–14.

Taylor, D. E. 2005, September. "Survey and Taxonomy of Packet Classification Techniques". *ACM Computing Surveys* 37:238–275.

Taylor, D. E., and J. S. Turner. 2007, June. "ClassBench: a packet classification benchmark". *IEEE/ACM Transactions on Networking* 15 (3).

Vamanan, B., G. Voskuilen, and T. N. Vijaykumar. 2010, August. "EffiCuts: Optimizing Packet Classification for Memory and Throughput". In *SIGCOMM '10*, Volume 41(4), 207–218.

Varenni, G., F. Stirano, E. Alessio, M. Baldi, L. Degioanni, and F. Risso. 2005, June. "Comparative Evaluation of Packet Classification Algorithms for Implementation on Resource Constrained Systems". In *ConTEL '05*, 135–139.

Varvello, M., R. Laufer, F. Zhang, and T. Lakshman. 2014, December. "Multi-Layer Packet Classification with Graphics Processing Units". In *CoNEXT '14*, 109–120.

Waldvogel, M. 2000. *Fast Longest Prefix Matching: Algorithms, Analysis, and Applications*. Ph. D. thesis, Swiss Federal Institute of Technology Zurich.

Wolf, T., and M. Franklin. 2000, April. "CommBench - A Telecommunications Benchmark for Network Processors". In *ISPASS '00*, 154–162.

Woo, T. Y. C. 2000, March. "A Modular Approach to Packet Classification: Algorithms and Results". In *INFOCOM '00*, Volume 3, 1213–1222.

AUTHOR BIOGRAPHIES

WLADISLAW GUSEW is a Ph.D. student in the Computer Engineering group at Humboldt University of Berlin, Germany. He received his M.S. in computer science from Humboldt University of Berlin in 2015. His interests include modeling and simulation in the domain of distributed computing and computer networks. His email address is gusewwla@informatik.hu-berlin.de.

SVEN HAGER is a Ph.D. student in the Computer Engineering Group at Humboldt University of Berlin, Germany. He received his M.S. in computer science from Heinrich Heine University Düsseldorf, Germany in 2013. Sven is active in the field of network packet classification, and his research interests include specializations of classification algorithms as well as rule set transformations. His email address is hagersve@informatik.hu-berlin.de.

BJÖRN SCHEUERMANN is a Full Professor and Chair of Computer Engineering at Humboldt University of Berlin, Germany. He studied mathematics and computer science and received his Ph.D. in 2007. After holding professorships in Düsseldorf, Würzburg and Bonn he joined Humboldt University in 2012. The focus of his scientific work is on performance, design, and security aspects of computer networks. Within this field, he works, for instance, on wireless communications, network privacy and anonymity, and network hardware design. His email address is scheuermann@informatik.hu-berlin.de.