JitVector: Just-in-Time Code Generation for Network Packet Classification

Samuel Brack Sven Hager Björn Scheuermann

Computer Engineering Group
Humboldt University of Berlin, Germany
Email: {samuel.brack, hagersve, scheuermann}@informatik.hu-berlin.de

***Abstract*—Network packet classification plays a pivotal role in packet-switched networks; it is at the heart of many functions including firewalling, QoS routing, and OpenFlow-based switching. However, packet classification is a hard problem, as packets must be classified within a short time frame. Existing classification techniques use sophisticated data structures which are traversed by generic search algorithms—that is, the algorithm is static, while the data structure reflects the configuration of the classifier. In this paper, we propose to break up the strict separation between data structure and algorithm by specializing the algorithm's implementation on the specific classification rules. We demonstrate the feasibility of our approach by introducing JitVector, which builds upon the well-known bit vector classification algorithm, but generates instance-specific machine code at runtime. In our evaluation, which also includes an integration into the OpenFlow Reference Switch, we show that JitVector achieves significant performance gains over an equivalent generic search scheme.**

***Index Terms*—Packet Classification; JIT; Bit Vector Search**

I. Introduction

Network packet classification is a central building block for many important functions in packet-switched networks [8], [15]. In packet classification, incoming packets are categorized using a specified classification rule set (e. g., firewall rules or an OpenFlow flow table). Classification aims for high data rates, in order to not cause a bottleneck in the network. Accordingly, the research community has proposed a wide variety of algorithmic schemes in order to accelerate the matching process performed during packet classification [6]–[8], [13], [15]. Although these techniques differ largely in detail, they all have in common that they translate a user-specified rule set into a data structure which is subsequently traversed by a generic search algorithm in order to classify incoming packets. Hence, these approaches strictly distinguish between data structure and search algorithm.

In this paper, we take the opposite approach and explore the potential of a classification algorithm which is specialized in the specific instance, i. e., for a specific rule set. The motivation behind this approach is that a specialized algorithm does not need to interpret the rule set at runtime, because the rule set is already incorporated in the algorithm's implementation. This can be achieved by leveraging a just-in-time compiler, which re-generates the matching code at runtime upon updates in the used rule set. Thus, we integrate parts of the search structure in the algorithm's machine code itself by using immediate operands in order to avoid additional memory accesses to be made during packet classification.

In order to evaluate this approach, we created *JitVector*, a classification technique which builds upon the well-known bit vector algorithm [9], but specializes its implementation based on the current rule set: x86_64 machine code for a tailored classifier is automatically generated at runtime. We examined JitVector both in an isolated environment as well as in a real system, namely the *OpenFlow Reference Switch implementation (ORS)* [10], [11]. We demonstrate (1) that JitVector achieves significantly better matching performance than the equivalent generic bit vector algorithm, and (2) that JitVector can increase the throughput of the ORS by more than one order of magnitude.

The remainder of the paper is structured as follows: In Sections II and III, we describe related work and introduce the packet classification problem, respectively. Next, we review the bit vector algorithm and describe the JitVector approach in Section IV, before we evaluate it in Section V. Finally, Section VI concludes this paper and describes future work.

II. Related Work

The concept of just-in-time compilation has been used in the past to increase the performance of virtual machines used for packet filtering [4], [5]. However, while these works are concerned with the efficient execution of high-level program code, in this paper we study the effects of specializing a dedicated classification algorithm on one of its inputs, namely the rule set used for packet classification.

During the last two decades, a lot of different classification schemes have been proposed, including decision tree algorithms, tuple space search, or decompositional methods [7]. As the name suggests, decision tree algorithms like HiCuts or Efficuts transform the rule set used for packet classification into multi-dimensional decision trees which can be subsequently traversed in order to classify incoming network packets [8], [15]. Tuple space search partitions the rule set into equivalence classes, so called tuples, which are searched successively using a fast hash function for each packet classification [13]. Decompositional schemes like the bit vector search reduce

978-1-4673-6770-7/15/$31.00 ©2015 IEEE

multi-dimensional packet classification to one-dimensional problems that can be solved independently. Subsequently, the partial solutions obtained by the one-dimensional searches are combined in order to compute the matching rule [9].

All of the above schemes have in common that they transform the specified rule set into a data structure which is amenable to be searched by a generic algorithm. In contrast, the JitVector approach proposed in this paper specializes the used search algorithm and thus generates instance-specific code with respect to the currently active rule set.

III. Problem Statement

Consider a network packet with a tuple $T = \langle h_1 \in H_1, \ldots, h_d \in H_d \rangle$ of d regarded header fields, where $H_1, \ldots, H_d$ are the domains of possible header values. Also, let $R = \langle R_1, \ldots, R_n \rangle$ be a *rule set* consisting of n rules. Each rule R_i specifies d *checks* $c_j^i : H_j \to \{true, false\}$. The goal of the packet classification problem is to find the smallest index $i \in \{1, \ldots, n\}$ for which rule R_i *matches* the regarded header fields T, i. e., for which it holds that $c_1^i(h_1) \wedge \ldots \wedge c_d^i(h_d)$. In practical applications, the checks specified by each rule are often simple equality, prefix, or range checks on fields like IP addresses, port numbers, or protocol numbers [9], [13]. Accordingly, we assume for the remainder of this paper that the domains $H_1, \ldots, H_d$ are intervals of consecutive non-negative integers which contain all possible values for the respective header field.

IV. The JitVector Approach

In this section, we first review the bit vector algorithm as our starting point. Subsequently, we describe how we built a runtime code generation engine which emits machine code for specific classifier instances, based on the bit vector concept.

A. Bit Vector Search

The bit vector scheme is a decompositional technique which consists of two basic steps [9]. First, the d-dimensional lookup problem is decomposed into d one-dimensional searches. Each of the one-dimensional searches yields a bit vector of size n which are combined in the second step to compute the index of the first matching rule. The bit vectors must be precomputed for each search dimension before the actual packet classification. We illustrate this preprocessing phase based on the two-dimensional rule set shown in Table I, which lists the rules with descending priority.

Figure 1 shows the geometric representation of the rule set from Table I, which is a collection of two-dimensional rectangles. For each dimension j, the end points of the rectangles are projected onto the jth axis, thereby partitioning the axis into a sequence S_j of at most $2n+1$ intervals. Then, for each $I \in S_j$, a bit vector V_I of n bits is created, whose ith bit is set to 1 iff I intersects with rule R_i in dimension j, as sketched in Figure 1. Accordingly, the space requirements for the bit vector algorithm are in $O(dn^2)$.

TABLE I: A two-dimensional rule set

Rule index	Field 1 (Domain: $[0, 15]$)	Field 2 (Domain: $[0, 7]$)
1	$[3, 11]$	$[4, 7]$
2	$[1, 5]$	$[2, 5]$
3	$[8, 13]$	$[0, 3]$

Once the bit vectors and intervals have been computed, the matching rules can be computed efficiently by locating the corresponding bit vector for each regarded header field through a binary search over the projection intervals. In our two-dimensional example shown in Figure 1, the header tuple $P_1 = (4, 3)$ falls into interval $[3, 5)$ for field 1 and into $[3, 4)$ for field 2. Subsequently, the bitwise AND of the d extracted bit vectors is computed, which yields a result vector V_{res} whose set bits indicate the positions of each rule that matches in every dimension. Finally, the index of the highest prioritized rule can be found by searching the index of the most significant set bit in the result vector V_{res}, which can be done in $O(n/w)$ time for a machine-specific constant w. Again, this process is illustrated in Figure 1.

B. JitVector Specialization

The JitVector approach combines the bit vector algorithm with the concept of function specialization through partial evaluation. Dedicated machine code is generated after rule set updates. This machine code performs the binary searches for each inspected header field dimension. Thus, JitVector essentially integrates parts of the search data structure into its own implementation. The motivation behind this approach is that the intervals and bit vectors remain static until the next rule set update, i. e. typically for a relatively long time (in relation to the rate of incoming packets). That is, the binary searches performed during packet classification always operate on the same sequence of intervals, with respect to each dimension. In essence, we spend some additional effort after a rule set update (for code generation) in order to further increase the performance afterwards (during classification). If we regard the binary search which is performed in dimension j on the interval sequence S_j using the jth header field h_j as an abstract procedure `bsearch` with $\texttt{bsearch}(S_j, h_j) \to \texttt{index}_{\text{bitvector}}$, then we can specialize `bsearch` on the rarely changing input parameter S_j, which yields a new procedure $\texttt{bsearch}_{S_j}(h_j) \to \texttt{index}_{\text{bitvector}}$. In fact, $\texttt{bsearch}_{S_j}(h_j)$ is equivalent to $\texttt{bsearch}(S_j, h_j)$, but is partially evaluated for the interval sequence S_j.

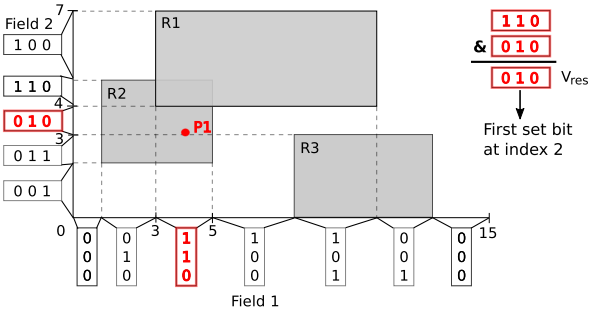


Fig. 1: Geometric representation of rule set and bit vectors.

Algorithm 1 Generating a specialized binary search.

```
1: function GenSearch(delims[], low, high)
2:     // delims[] contains the interval delimiters.
3:     // The register reg contains the header value.
4:     mid ← low + ⌊(high − low)/2⌋
5:     if high == low then
6:         EmitCode(Compare: reg < delims[low])
7:         EmitCode(JumpIfTrue: CASE2)
8:         EmitCode(Return: low)
9:         EmitCode(Return: low-1)                    ▷ CASE2
10:        return Number of written bytes in EmitCode
11:    end if
12:    leftTree ← GenSearch(delims[],low,mid-1)
13:    rightTree ← GenSearch(delims[],mid+1,high)
14:    EmitCode(Compare: reg < delims[mid])
15:    EmitCode(JumpIfTrue: LEFT)
16:    EmitCode(Compare: delims[mid+1] < reg)
17:    EmitCode(JumpIfTrue: RIGHT)
18:    EmitCode(Return: mid)
19:    EmitCode(leftTree)                             ▷ LEFT
20:    EmitCode(rightTree)                            ▷ RIGHT
21:    return Number of written bytes in EmitCode
22: end function
```

Such specialized functions can be generated by linearizing the binary search trees represented by the interval sequences. These linearizations can be written as native machine code into executable memory. This is depicted by Algorithm 1, which takes an interval sequence `delims[]` and recursively emits the binary tree structure as a sequence of x86_64 instructions. An important detail of Algorithm 1 is that it inserts the interval boundaries used for comparisons as immediate operands in the generated machine instructions, as depicted in lines 6, 14, and 16. This and the fact that the search key is held in a register `reg` has the implication that a specialized binary search does not require any data memory accesses during its operation. Thus, despite the fact that the JitVector approach has the same theoretical worst case performance as the plain bit vector algorithm, the matching performance is significantly increased, which we demonstrate in our evaluation.

V. Evaluation

We evaluated the JitVector approach both in an isolated scenario as well as in a real system, namely the *OpenFlow Reference Switch (ORS)*. Both the isolated system and the ORS are implemented in C. All experiments in our evaluation were conducted on an otherwise idle computer with an Intel Core 2 Duo CPU and 3 GB of RAM running Linux 3.18.6. Measurements involving the ORS were executed in a virtual network created by the *mininet* tool [3].

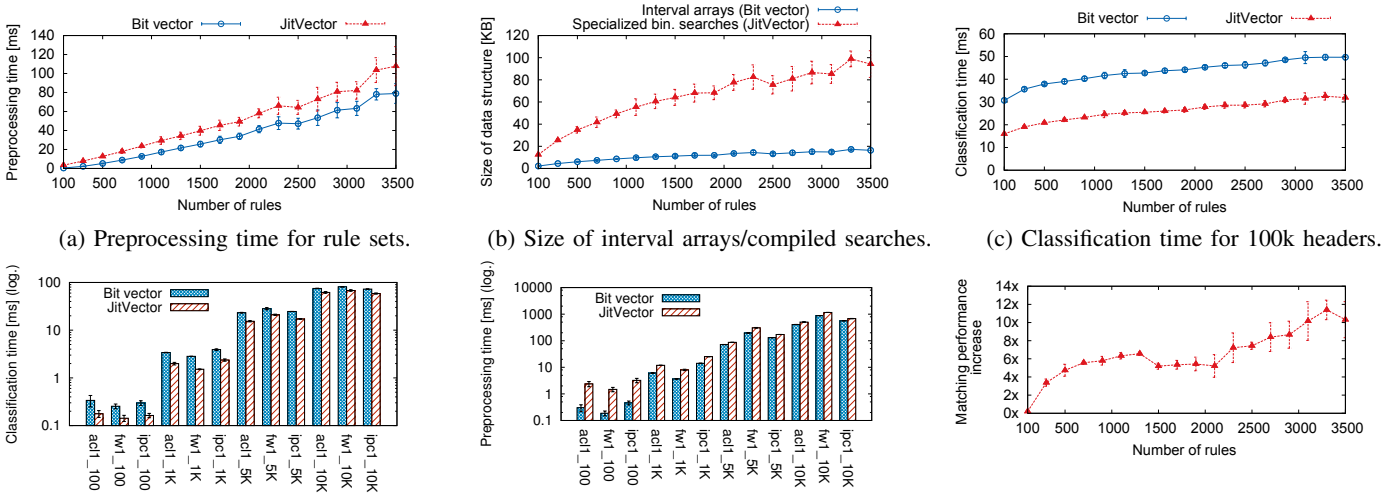
A. Measurements in the Isolated Environment

In our first series of measurements, we evaluated the JitVector approach in an isolated environment. It does not classify real network packets, but instead concentrates on the classification task itself, which is performed for a set of predefined rule sets and corresponding packet header traces. Hence, we can compare the main performance characteristics of the bit vector and JitVector approaches, such as classification speed and preprocessing time, without adding I/O noise. We performed our measurements as follows: first, we generated rule sets consisting of 100 up to 3 500 rules in steps of 200. For each rule set, we also generated a trace of 100 000 header values, uniformly distributed over the specified rules. We employed the *ClassBench* tool [14] for both rule set and trace generation. ClassBench is a benchmark generator for packet classification algorithms that is capable of creating rule sets of an arbitrary size based on seed files which describe the structure of real filter sets. Here, we used the `ipc` (*ipchains*) seed files which are publicly available at [2]. Each generated rule defines checks on IPv4 source and destination addresses, the transport protocol, and source and destination port ranges.

Next, we classified each header tuple in the generated traces using the corresponding rule set with both the bit vector and JitVector algorithms in order to measure the preprocessing times, the classification speed, and the size of the resulting data structures/functions. Each experiment was repeated ten times. Mean values and standard deviations of the observed quantities are shown in Figures 2a to 2c. Figure 2a indicates that JitVector has longer preprocessing times than the plain bit vector approach due to the transformation of the interval sequences into specialized functions. The amount of memory needed to store these functions exceeds the size of the interval arrays by a factor between 5 and 6, as each specialized function contains not only the interval delimiters, but also the instructions of the unrolled binary search, as indicated by Figure 2b. However, Figure 2c depicts that this overhead is rewarded by a significant speedup in terms of classification performance as a result of the faster execution of the partially evaluated functions.

We also investigated the amount of L1 cache misses for the JitVector and bit vector algorithms using `cachegrind` [1]. For all rule set/trace pairs in the generated data set, the JitVector approach had about four times more instruction cache misses and 1.3 to 1.7 times less data cache misses than the bit vector algorithm. However, for both JitVector and bit vector, the absolute number of data cache misses was four orders of magnitudes higher than the number of instruction cache misses, which renders the instruction cache misses negligible.

In addition to our self-generated rule sets and traces, we also evaluated the JitVector approach for twelve publicly available data sets [12], which were likewise generated by ClassBench. Beyond `ipc` rules, these benchmarks also include `acl` (access control list) and `fw` (firewall) rule sets which mainly differ in the number of wildcarded fields. These rule sets are available in four size classes (100, 1k, 5k, 10k); the corresponding trace files include roughly ten times as many headers. For each of the twelve rule sets, we classified the corresponding traces ten times both with the JitVector and bit vector algorithms and measured classification, preprocessing times, and memory requirements. The former two quantities are illustrated by Figures 2d and 2e. Again, the figures confirm that JitVector provides better classification performance than the bit vector algorithm, at the cost of some additional preprocessing. Also like above, the memory requirements of JitVector for all publicly available rule sets are 5 to 6 times higher than those of the bit vector algorithm.



(a) Preprocessing time for rule sets.

(b) Size of interval arrays/compiled searches.

(c) Classification time for 100k headers.

(d) Classification times with public rule sets.

(e) Preprocessing time for public rule sets.

(f) Performance increase in the ORS.

Fig. 2: Evaluation results.

B. Measurements in the ORS

In order to investigate the performance of the JitVector algorithm in a more realistic scenario, we integrated it into the OpenFlow Reference Switch (ORS, version 1.0) [11]. The ORS employs a matching engine which performs a basic linear search to classify incoming packets. We explored the achievable throughput gains by replacing this matching engine by the JitVector approach. Different from the previous setting, the ORS uses twelve instead of five header fields for packet classification. For this reason, we could not use ClassBench for rule set generation. Instead, we generated random rule sets with a fixed IPv4 destination address A as well as uniformly distributed UDP traces destined for virtual host A. We then sent minimal-sized UDP packets according to the traces from a virtual sending host via the ORS to host A for a duration of ten seconds. We counted the number of packets which were received by A. We repeated this experiment ten times for both the linear search and the JitVector algorithm. Figure 2f shows the averages as well as the standard deviations of the quotients $packet\_count_{\text{JitVector}} / packet\_count_{\text{linear search}}$. Although we observe fluctuations in the graph, which may be introduced by unfortunate rule set/trace combinations or I/O effects, the figure also underlines a clear gain in performance.

VI. Conclusion and Future Work

In this work, we explored the potential of classification algorithm specialization through the example of JitVector, a classification algorithm which makes use of dynamic code generation in order to increase its matching performance. In contrast to existing work in this field, JitVector specializes its own implementation on the currently used rule set. We demonstrated that JitVector provides better classification performance than an equivalent generic algorithm. Furthermore, we integrated JitVector into the OpenFlow Reference Switch and thereby increased its throughput by over an order of magnitude. Future work includes optimization of the generated machine code by, e. g., considering the CPU's branch prediction strategies or exploiting the potential of SIMD instructions.

Acknowledgment

This work was funded by the BMWi (German Federal Ministry of Economics and Energy) in the context of the HARDFIRE project.

References

[1] "Cachegrind website," http://valgrind.org/docs/manual/cg-manual.html, last access: July 10, 2015.

[2] "ClassBench website," http://www.arl.wustl.edu/classbench/, last access: April 9, 2015.

[3] "Mininet: An instant virtual network on your laptop (or other PC)," http://www.mininet.org, last access: April 9, 2015.

[4] A. Begel, S. McCanne, and S. Graham, "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture," in *SIGCOMM '99*, Aug. 1999, pp. 123–134.

[5] D. Engler and M. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *SIGCOMM '96*, Aug. 1996, pp. 53–59.

[6] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *SIGCOMM '99*, Aug. 1999, pp. 147–160.

[7] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network: The Magazine of Global Internetworking*, vol. 15, no. 2, pp. 24–32, Mar. 2001.

[8] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *HOTI '99*, Aug. 1999, pp. 34–41.

[9] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *SIGCOMM '98*, Aug. 1998, pp. 203–214.

[10] N. McKeown *et al.*, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[11] A. Nygren *et al.*, "OpenFlow switch specification," Open Networking Foundation, Tech. Rep., Oct. 2013.

[12] H. Song, "Evaluation of packet classification algorithms," http://www.arl.wustl.edu/~hs1/PClassEval.html, website includes publicly available rule sets, last access: April 9, 2015.

[13] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM '99*, Aug. 1999, pp. 135–146.

[14] D. Taylor and J. Turner, "Classbench: a packet classification benchmark," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 499–511, Jun. 2007.

[15] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "Efficuts: Optimizing packet classification for memory and throughput," in *SIGCOMM '10*, Aug. 2010, pp. 207–218.