CAB: A Reactive Wildcard Rule Caching System for Software-Defined Networks

Bo Yan, Yang Xu, Hongya Xing, Kang Xi, H. Jonathan Chao
New York University Polytechnic School of Engineering
{boven.yan, yang, hongya, kang.xi, chao}@nyu.edu

ABSTRACT

Software-Defined Networking (SDN) enables flexible flow control by caching policy rules at OpenFlow switches. Compared with exact-match rule caching, wildcard rule caching can better preserve the flow table space at switches. However, one of the challenges for wildcard rule caching is the dependency between rules, which is generated by caching wildcard rules overlapped in field space with different priorities. Failure to handle the rule dependency may lead to wrong matching decisions for newly arrived flows, or may introduce high storage overhead in flow table memory.

In this paper, we propose a wildcard rule caching system for SDN named CAching in Buckets (CAB). The main idea of CAB is to partition the field space into logical structures called buckets, and cache buckets along with all the associated rules. Through CAB, we resolve the rule dependency problem with small storage overhead. Compared to previous schemes, CAB reduces the flow setup requests by an order of magnitude, saves control bandwidth by a half, and significantly reduce average flow setup time.

Categories and Subject Descriptors

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

General Terms

Design, Standardization

Keywords

SDN, Wildcard Rule Caching, Buckets

1. INTRODUCTION

Software-Defined Networking (SDN) enables network innovations and provides flexible flow control over network traffic. SDN proposes a variety of network policies for access control, traffic engineering, and energy efficiency to enhance the management of the network. These policies can be realized through the rules placed in the flow tables of OpenFlow switches to direct traffic forwarding.

A rule can be stored either as an exact-match rule [1] [2] or a wildcard rule [4] [11] in the switch. Compared with exact-match rules, wildcard rules improve the reusability of rules in the flow table and reduce the number of flow setup requests to the controller, enhancing the scalability of the system. However, wildcard rules are typically supported by Ternary Content Addressable Memory (TCAM), which is highly limited in capacity. The flow table in a commodity switch is reported to support only a few thousand wildcard rules [2]. To improve scalability, recent study suggests either proactively allocating rules on multiple switches to load balance the flow table consumption [6] or reactively caching rules [2] [11] on each switch. Compared with proactive schemes, the reactive approach dynamically caches active rules in switches on demand, which saves flow table space and enables rapid reaction to traffic dynamics.

Reactively caching wildcard rules in switches creates several challenges. First, the cache miss rate needs to be controlled to improve network performance. Packets suffering from a cache miss will experience a 2-ms latency compared to a 5-ns latency with a cache hit [2]. A high cache miss rate also leads to frequent invocations to the controller, and consumes the limited control network bandwidth. Second, dependency between rules complicates the caching process. Since rules overlap in field space with different priorities, simply caching the requested rule can generate false packet forwarding [11]. To guarantee the semantic correctness of rules cached in switches, extra storage overhead is required, which increases the chance of flow table overflow.

To address the problem, we propose a novel reactive wildcard rule caching system named CAching in Buckets (CAB). The main idea of CAB is to divide the geometric representation of the rule set or the *field space* into many small logical structures called *buckets*, and to associate each rule with one or multiple buckets according to its location in the field space. To ensure semantic correctness, rules associated with the same bucket are always cached together.

The main contributions of our work are summarized as follows:

1. We propose a design architecture of CAB that improves the efficiency of flow table usage and provides the semantic correctness guarantee when dependency among wildcard rules exists. We design a two-stage flow table pipeline for switches that support CAB, and

HotSDN'14, August 22, 2014, Chicago, IL, USA.
Copyright 2014 ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620732.

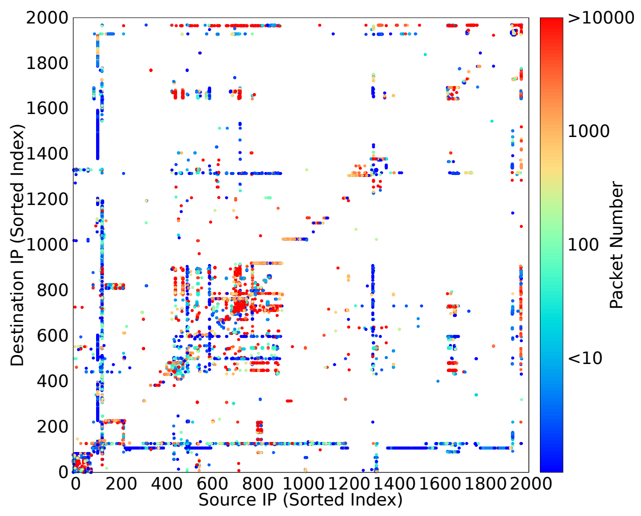

Figure 1: Traffic locality

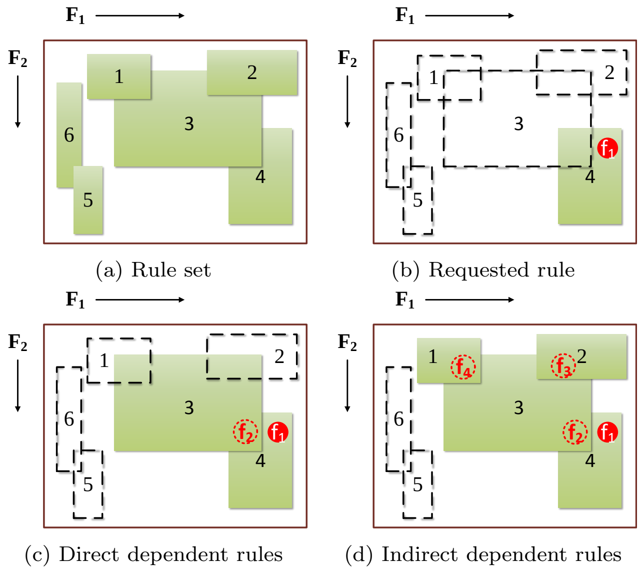

Figure 2: Rule dependency problem (the solid rectangles represent the rules that are cached in the switch, while the hollow ones represent those not cached)

is fully compatible with the OpenFlow switch specification 1.4 [7].

2. We analyze the bucket generation problem, and develop a heuristic bucket generation algorithm to improve system performance.

3. To evaluate the performance of CAB, we compare CAB with different rule caching schemes through simulations. The results show that CAB prevails in reducing flow setup requests, controlling bandwidth consumption, and introducing the lowest flow setup latency. We also discuss the parameter tuning of CAB, which shows promise for a line of research.

The rest of the paper is organized as follows. In Section 2, we introduce in detail the motivation and challenges of this work. Section 3 presents the caching mechanism of CAB and the switch implementation, followed by discussions on bucket generation in Section 4. Section 5 shows the performance evaluation. We conclude the paper and discuss some future work in Section 6.

2. MOTIVATION AND CHALLENGES

2.1 Traffic Locality

A key observation that motivates caching wildcard rules instead of exact-match rules is the traffic locality in a variety of network scenarios. Here we define the term traffic locality as, during a short period of time, traffic is localized in close proximity to the field space. We analyzed the traffic traces from a working enterprise data center from New York City Department of Education (NYCDoE-DC). The host pairs of the traces on a core-to-aggregate link of a switch with a duration of 5 minutes are plotted in Figure 1. We can see that most traffic is localized in several blocks. Similar evidence of traffic locality has been observed by previous studies in data centers [5] and ISP networks [3].

Suppose we cache exact-match rules. Almost every flow passing the switch will trigger a rule installation request to the controller. In scenarios such as NYCDoE-DC, the number of new requests generated amounts to 2000 per port per second. Previous work [10] reports that there can be up to 200k flow setup requests for a 4000-server cluster. Caching exact-match rules easily overwhelms the controllers' ability to processing requests and exhausts the limited control network bandwidth.

By contrast, caching wildcard rules can exploit the traffic locality to reduce cache misses. The flows in the 'hot' blocks in traces may share the same wildcard rules or neighboring ones. Each rule can be re-used frequently within short periods of time. As flows referring to the same wildcard rule aggregately generate one single request to the controller, the controller load and control bandwidth consumption are effectively reduced.

2.2 Rule Dependency Problem

Wildcard rules are assigned different priorities to avoid conflicts as they may overlap in the field space. However, this generates a dependency problem for rule caching. To guarantee semantic correctness of packet matching, extra memory cost has to be introduced to the flow table, which increases the chance of flow table overflow. We use the example in Figure 2 to demonstrate this problem. Consider a rule set with six rules over a two dimensional field space. Assume initially the flow table for a certain switch is empty. When the first flow f_1 arrives, the switch needs to invoke the controller to install the corresponding Rule 4 (Figure 2b). However, simply installing Rule 4 would cause potential wrong matching decisions at the switch in the future. Suppose later another flow f_2 arrives at the switch and hits the space where Rule 3 and 4 overlaps. It will locally match with Rule 4 instead of the expected Rule 3, since the latter is not cached in the switch. This indicates that when caching Rule 4, the switch should also cache Rule 3 to guarantee correct matching in the future (Figure 2c). For the same reason, Rules 1 and 2 also need to be cached along with

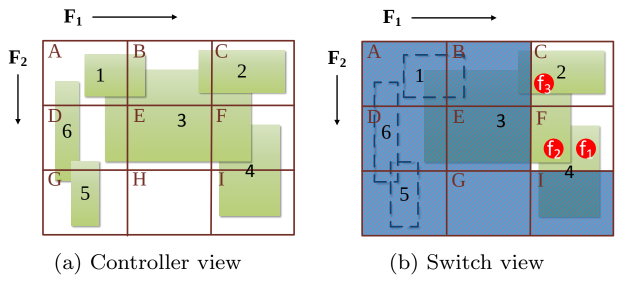

Figure 3: Caching rules in bucket

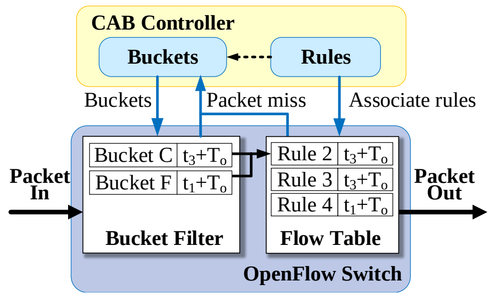

Figure 4: CAB implementation

Rule 3, or flows f_3 and f_4 will get wrong matching in the switch (Figure 2d). Therefore, Rules $1, 2$ and 3 need to be installed whenever Rule 4 is cached, regardless of the future flow arrivals. The problem lies in that Rule 3 has higher priority and overlaps with Rule 4, which is a directly *dependent rule* of Rule 4. While Rules 1 and 2 are dependent with Rule 3, they are indirectly dependent with Rule 4.

Due to rule dependency, in the naive way, all the dependent rules, directly and indirectly, need to be cached along with the requested rule. Using rule sets generated by Classbench [9], we found that the average number of dependent rules reaches up to 350 for each rule for a set with 8k rules. In the worst case, the default rule is dependent on all the rules, which amounts to the size of the whole set. Therefore, caching all dependent rules of a matched rule can easily overflow the switch flow table and is considered infeasible.

Another approach to resolve the dependency is to convert rules to new micro rules without overlapping [3] [11]. However, since each rule has tens of overlapped rules on average, slicing them into non-overlapping micro rules generates quite a lot of entries. This tends to increase the number of entries cached in the flow table, which increases the chance of overwhelming the TCAM memory in the switch and adds complexity for the controller to store and update the rules. In designing CAB, we resolve the rule dependency problem without modifying the original rule set.

3. CAB DESIGN

3.1 Caching Rules in Buckets

The core idea of CAB is to use a filtering structure named bucket to guarantee correct packet matching and control the number of rules necessary to be cached in the switch. In CAB, the field space of packet headers is partitioned into small hyper-rectangles or buckets as shown in Figure 3a. Each bucket can be represented by a wildcard rule that can be stored in the TCAM. We define the *associated rules* of a certain bucket as the rules overlap with the bucket in the field space. Whenever a flow setup request comes, the controller finds the requested bucket and caches it along with all the associated rules onto the switch. For instance, in the example of Figure 3b, Buckets C and F are cached with Rules $2, 3$ and 4. Note that within each bucket, the rules have the same semantic views as in the controller, while a packet failed to hit a bucket will trigger an installation to satisfy the request (Figure 3a). Therefore, the switch can always guarantee correct forwarding of packets and resolves the rule dependency problem.

In terms of memory cost, the bucket design brings several other benefits. First, through buckets we can control the rules installed for each request. As shown in Figure 3b, for f_1 only Rules 3 and 4 along with Bucket F need to be cached, instead of having all the dependent rules of Rule 4 (i.e., Rules $1, 2$ and 3) cached at one time. Second, we can benefit from traffic locality by caching neighboring rules in buckets. For instance, as f_1 and f_2 approach in proximity. f_1 generates only one request to install Bucket F. Later, f_2 can be forwarded at the line rate without generating a request to the controller or experiencing a setup latency. Third, all rules are still cached in their original form without being split. This avoids generating too many micro rules, which would otherwise consume a lot of flow table memory.

3.2 Switch Implementation

To support the idea of CAB, the switch is implemented by a two-stage table pipeline [7] consisting of a bucket filter and a flow table (Figure 4). All the buckets cached in the bucket filter have the same priority with the same action that directs the matching packet to the next stage flow table. The rules are cached in the flow table according to their original priority. A packet that failed to find a matching bucket in the bucket filter will have its header encapsulated and forwarded to the controller. When receiving a request, the controller will perform a bucket search and install the bucket and all its associated rules on the switch.

In CAB, the bucket is the atomic unit in the installation and destruction of cached entries. Note that multiple buckets can share the same rules. When deleting a rule, we must ensure that all the buckets that are associated with the rule have been deleted. We guarantee this consistency by assigning the same timeout value T_o to the bucket and all its associated rules. If a rule is found to have been cached earlier along with another bucket, the timeout value is refreshed according to the most recent installation.

A concrete example: We consider the flow arrivals as Figure 3b shows. Assume initially both the bucket filter and the flow table are empty. At t_1, a flow f_1 arrives at the switch, resulting in a flow setup request sent to the controller. The controller installs Bucket F in the bucket filter and its associated Rules 3 and 4 in the flow table. The timeout is set as $t_1 + T_o$. At t_2, flow f_2 arrives and passes the bucket filter as it matches with Bucket F. Then it is forwarded to the flow table and matches with Rule 3. Later at t_3, another flow f_3 arrives at the switch. Since f_3 doesn't match with any buckets in the bucket filter, it triggers the installation of Bucket C and Rule 2 with the timeout set as $t_3 + T_o$. As Rule 3 is already in the switch, the timeout is

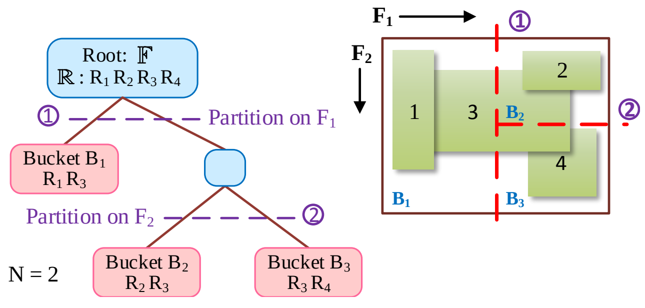

Figure 5: Bucket generation decision tree

refreshed to $t_3 + T_o$. Figure 4 depicts the table status after t_3.

The switch implementation of CAB is fully compatible with the flow table pipeline specified in the OpenFlow switch specification 1.4 [7]. While the bucket filter and the flow table are two tables logically, they can be implemented by two separate TCAM chips or one TCAM chip with multiple lookups.

4. BUCKET GENERATION

4.1 Bucket Generation Problem

The generation of buckets is handled by a software based controller. Bucket generation deals with the following problem: given a policy rule set $\mathbb{R}$, partition the d-dimensional field space $\mathbb{F}$ into a bucket set $\mathbb{B}$. An optimal set of buckets generated shall get the lowest cache miss rate when the OpenFlow switch caches rules with the buckets generated to process the arriving packets.

We observe that the size of buckets directly affects the cache performance. In our discussion, the size of a bucket is defined as the number of rules associated with it. A larger bucket can potentially reduce the flow setup requests, since more flows arriving at the switch tend to share the same bucket and aggregately generate fewer requests. However, it can also lead to wasted space in the flow table (some rules are installed but not matched by any flows) and increase the chance of flow table overflow. Furthermore, a larger bucket will trigger installation of more rules each time it is requested, which consumes more control bandwidth. On the other hand, a small bucket may reduce the unused rules cached in switches, but the switch tends to cache more buckets in the bucket filter, which also consumes space in TCAM. In CAB, we bound the size of each bucket with a fixed value N. The optimal choice of N depends on both the distribution of rules and traffic pattern. In reality, we tune the value using historical traces. The effect of different N is discussed in detail in Section 5.

4.2 Bucket Generation Algorithm

The bucket generation algorithm design shares similarity with earlier packet classification work such as HiCuts and HyperCuts [8], all generating decision trees for field space partitioning. In bucket generation, we specifically taking into account that buckets needs to be represented by wildcard rules so as to be stored in TCAM. This requires us to conduct binary or multi-binary cuts on each dimension.

As shown in Figure 5 , We start from the root node of the tree which represents the $d-$dimensional field space $\mathbb{F}$ associated with the whole rule set $\mathbb{R}$. Then we recursively partition the node from the tree into child node with smaller size (defined as the number of associated rules of the node). Each node therefore represents a hyper-rectangle in the field space. The partitioning terminates till the size of the node non-greater than the predetermined N, and the leaf node is marked as a bucket. Each time we select m out of d dimensions to partition a node. Suppose we denote a candidate m-combination to partition a node S as c_i (i indexed from 0 to $\binom{d}{m} - 1$), we'll generate 2^m child node denoted $S_k^{c_i}$. For instance, in Figure 5 we partition $m = 1$ dimension each time on a $d = 2$ dimensional field, and each time we generate 2^1 child nodes.

Algorithm 1 Bucket Generation Algorithm

```
Input: Rule set $\mathbb{R}$, Node $S$, Size bound $N$
Output: Decision tree $T$, Bucket Set $\mathbb{B}$
 1: function PARTITION($\mathbb{R}, S, N$)
 2:     for each $c_i$ made from $d$ dimensions do
 3:         Partition on dimensions of $c_i$, and calculate the
            cost $Cost_{c_i}(S)$
 4:     end for
 5:     Find $c_i$ where the $Cost_{c_i}(S)$ is minimized, and ap-
        pend $S_0^{c_i}...S_{2^m-1}^{c_i}$ to $T$
 6:     for each child hyper-rectangle $S_k^{c_i}$ do
 7:         if $A(S_k^{c_i}) \leq N$ then
 8:             Append $S_k^{c_i}$ to $\mathbb{B}$
 9:             return
10:         else PARTITION($\mathbb{R}, S_k^{c_i}, N$)
11:         end if
12:     end for
13: end function
```

To determine the c_i to partition on, we consider both the sum and the deviation of the sizes of nodes generated by a trial cut on c_i. We choose the c_i that minimize a cost function defined as follows,

$$Cost_{c_i}(S) = \sum_{k=0}^{2^m-1} A(S_k^{c_i}) + \delta \sum_{k=0}^{2^m-1} (A(S_k^{c_i}) - \sum_{k=0}^{2^m-1} A(S_k^{c_i})/2^m)$$

where $A(S_k^{c_i})$ stands for the number of associated rules or the size of hyper-rectangle $S_k^{c_i}$, and δ is a positive value to adjust the weight between the sum and the deviation. Note that by minimizing the sum of the node, we can reduce the redundancy of having the same rules associate with different buckets. By balancing the size of the nodes generated, we can avoid having some nodes having a relatively large size, which increase the depth of the tree.

In our implementation, each time we cut $m = 2$ dimensions over the 5-tuple field space, which is a good tradeoff between the decision tree search time and storage overhead. Suppose we choose to partition on one dimension each time as in Figure 5, the tree will become very deep and prolong the decision tree search time. By contrast, if we set a larger m, say $m = 5$, each partition will generate $2^5 = 32$ children, leading to a large memory overhead with many duplicated rules in buckets. δ is tuned to minimize the number of buckets.

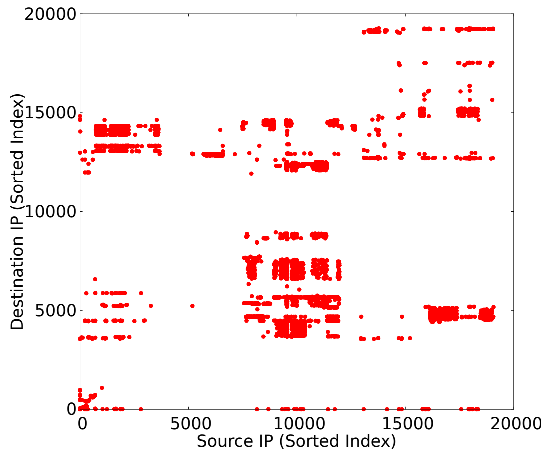

Figure 6: Synthetic traces generated

5. PERFORMANCE EVALUATION

5.1 Simulation Setup

Rules and traces generation: As we cannot obtain a real rule set with enough scale for the simulation, a synthetic rule set with 4k rules is generated using ClassBench [9] for the tests. Although we do have real traces from NYCDoE-DC, directly using it with synthetic rules is infeasible. Most traffic hits the default rules due to inconsistency between traces and rules. To bridge the gap, we develop a header mapping technique that maps the real traces on to synthetic rules[1]. As we do this, the statistics of flows (packet size, inter-arrival rate, and flow duration) are preserved. Trace pruning and interpolation are designed for tuning traffic load. A snapshot of the synthetic trace generated is shown in Figure 6 with a duration of 5 minutes. We can see a close resemblance of the layout of synthetic traces to the real ones in Figure 1.

Device parameters: In the tests we simulate a single OpenFlow switch with direct connection to the controller. The TCAM capacity is set to support 1500 entries, each costing 288 bits memory. For simplicity, the RTT for the flow setup upon a cache miss is set fixed to be 2ms while the forwarding delay at a line rate is 5ns [2]. Queuing delays of rule installation are not covered in our simulation.

Schemes for testing: We evaluate the following three rule caching schemes along with CAB to compare performance. All the schemes guarantee the semantic correctness of packet matching.

1. Caching exact-match rules (CEM) is proposed in Ethane [1] and DevoFlow [2] to control all or partial flows in the network. CEM suggests caching the exact-match rules of a flow when corresponding entry is absent in the flow table. In the simulation we consider exact-match rules to be supported by SRAMs with 200k-entry capacity.

2. Caching micro-rules (CMR) is proposed by Q. Dong *et al.* [3] and applied in DIFANE [11]. CMR partitions the rule set into new micro-rules without overlapping and caches them in the switches.

[1]source codes available at `https://github.com/bovenyan`

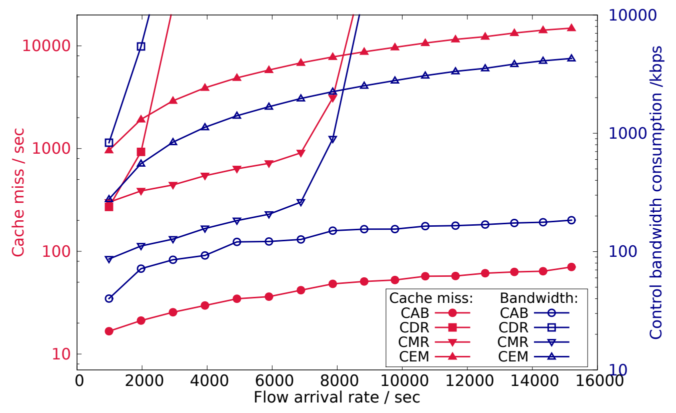

Figure 7: Resource consumption

Table 1: Average flow setup time

CAB	CMR	CDR	CEM
37.9 us	66.0 us	553.4us	1965.3 us

3. Caching dependent rules (CDR) works as a naive benchmark. When a flow request is received, CDR installs the requested rule and all its dependent rules to the flow table.

5.2 Simulation Results

Resource Consumption: We measured the cache miss rate and the control bandwidth consumption of four different rule caching schemes with varying flow arrival rates. We tested on traces with a duration of 20 minutes. The results are shown in Figure 7. Since rarely do flows share exactly the same 5-tuple address, each flow arrival contributes to a cache miss in CEM. CDR requires all the dependent rules to be installed; therefore, it quickly overflows the memory. CMR benefits from caching micro range rules and performs better than the prior two. However, as too many micro rules are generated and cached, it also overwhelms the memory shortly after. By contrast, CAB generates more than one tenth cache misses of CMR, and the bandwidth consumption is more than a half less. Although eventually it overflows the table (not shown in the graph), it can stably support an arrival rate of 15000 flows/sec.

Flow setup time: Table 1 lists the average flow setup times given the arrival rate of 1000 flows/sec. We don't present results under higher arrival rate because CDR and CMR experience flow table overflow, rendering unpredictably long setup times due to request queuing and control bandwidth congestion. As expected, CAB achieves the lowest flow setup latency. CMR also enjoys nice performance since micro rules are frequently reused by localized traffic. CDR is inferior to the prior two as caching all dependent rules increases cache misses. For CEM, since almost each flow needs to be forwarded to the controller, the average setup time is close to the RTT of a flow setup. Considering the results denote the average delay on each flow per switch, the difference on even the micro second scale means a lot.

Effects of tuning bucket size: Figure 8 presents the effect of tuning the bound of bucket size N on cache miss and rules installed per second. We test on traces with an arrival rate of 7000 flows/sec. For a smaller bucket size,

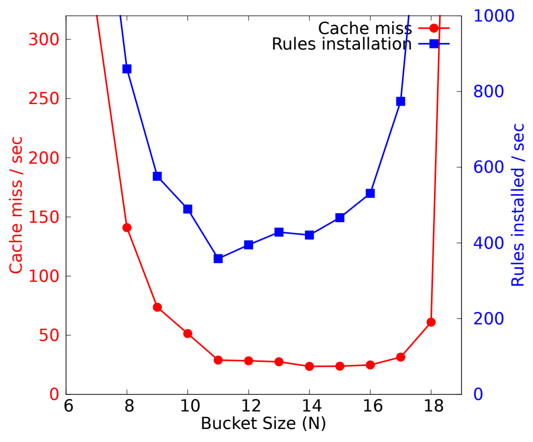

Figure 8: Effect of tuning bucket size

cache miss is high because we install more buckets. Increasing the bucket size reduces the cache miss within a range, as each bucket is reused by more flows. Meanwhile, the number of rules installed per second increases because a larger bucket has more associated rules. More flow table space is consumed to store the rules. Beyond a certain bound($N > 16$), the table is overflowed and cache misses increase significantly. A proper selection of N ranges from 9 to 16. The tuning of bucket size shows the potential of dynamic bucket generation, where the size of bucket changes over time to accommodate the time-varying traffic.

6. CONCLUSION AND FUTURE WORK

In this paper, we present CAB, a novel reactive wildcard rule caching system. CAB resolves rule dependency while achieving efficient use of control network bandwidth, and reducing controller processing load and flow setup latency. We plan to further look into more dynamic scenarios where the traffic pattern changes more frequently and buckets are generated and tuned dynamically. A prototype evaluation of the system is on schedule to further investigate implementation issues.

7. REFERENCES

[1] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking (TON)*, 17(4):1270–1283, 2009.

[2] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[3] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal. Wire speed packet classification without tcams: a few more registers (and a bit of logic) are enough. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 253–264. ACM, 2007.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[5] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[6] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 157–170. USENIX Association, 2013.

[7] OpenFlow Switch Specification. Version 1.4.0 (wire protocol 0x05). `https://www.opennetworking.org/`, Oct.14 2013.

[8] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proceedings of the ACM SIGCOMM 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224. ACM, 2003.

[9] D. E. Taylor and J. S. Turner. Classbench: a packet classification benchmark. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2068–2079. IEEE, 2005.

[10] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2012.

[11] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 41(4):351–362, 2011.