Wire Speed Packet Classification Without TCAMs: A Few More Registers (And A Bit of Logic) Are Enough

Qunfeng Dong
University of Wisconsin

Suman Banerjee
University of Wisconsin

Jia Wang
AT&T Labs - Research

Dheeraj Agrawal
University of Wisconsin

ABSTRACT

Packet classification is the foundation of many Internet functions such as QoS and security. A long thread of research has proposed efficient software-based solutions to this problem. Such software solutions are attractive because they require cheap memory systems for implementation, thus bringing down the overall cost of the system. In contrast, hardware-based solutions use more expensive memory systems, e.g., TCAMs, but are often preferred by router vendors for their faster classification speeds. The goal of this paper is to find a 'best-of-both-worlds' solution — a solution that incurs a cost of a software-based system and has a speed of a hardware-based one. Our proposed solution, called *smart rule cache* achieves this goal by using minimal hardware — a few additional registers — to cache *evolving* rules which preserve classification semantics, and additional logic to match incoming packets to these rules. Using real traffic traces and real rule sets from a tier-1 ISP, we show such a setup is sufficient to achieve very high hit ratios for fast classification in hardware. Cache miss ratios are $2 \sim 4$ orders of magnitude lower than flow cache schemes. Given its low cost and good performance, we believe our solution may create significant impact on current industry practice.

Categories and Subject Descriptors

C.2.6 [**Computer Communication Networks**]: Internetworking—*Routers*

General Terms

Algorithms, Design, Performance

Keywords

Packet Classification, Rule Cache, Rule Evolution

1. INTRODUCTION

As the foundation of many Internet functions such as QoS and security, packet classification involves matching each incoming packet against a set of rules defined over some packet header fields. For each packet header field F, a rule specifies a range literal $F \in [a, b]$. When matching a packet against a rule, each literal in the rule is evaluated on the corresponding packet header field. If every literal is evaluated to be `true`, the rule is considered to *match* the packet. Besides the literals, each rule also specifies a decision. Although a packet may match more than one rule, there is a strict ordering among rules and the goal is to find the *first* matching rule, i.e., the one with the highest priority.

Packet classification as a theory problem is inherently hard. Overmars and van der Stappen [18] have shown that for packet classification over $d > 3$ packet header fields, the best known algorithms have either $O(\log n)$ search time at the cost of $O(n^d)$ space or $O(\log^{d-1} n)$ search time at the cost of $O(n)$ space, where n is the number of rules in the rule set. While fast network processors have been successfully designed to keep up with wire speeds, the widening gap between memory access speeds and wire speeds represents an increasingly tough challenge to pure software solutions.[1]

Therefore, most router vendors favor hardware solutions based on *Ternary Content Addressable Memory (TCAM)* [17] for its fast speed. Basically, TCAMs can compare a given search key (i.e., a packet) with all entries (i.e., stored rules) in parallel and returns the first matching entry in one single clock cycle. However, as a more complex technology, TCAM is more expensive and more power consuming than conventional DRAM/SRAM-based systems. Moreover, TCAM is well known to suffer size explosion due to inefficient range specification [6]. As wire speeds and rule set size rapidly increase, pure TCAM-based solutions will become increasingly expensive.

To summarize, hardware solutions are attractive for their ability to classify packets at wire speeds, but are quite expensive and are a significant part of the cost of a line card;[2] on the other hand, software solutions reduce expensive hardware costs (since they can be implemented in much less expensive DRAMs), but can rarely match the speed of hardware solutions. In this paper, we therefore, address the following challenging problem — *is it possible to design a classification system that has a cost similar to a software-based system and speed of a hardware-based system?* We answer this question in the affirmative and present an approach which can provide the best-of-both-worlds solution to packet classification.

Our approach called *smart rule cache* has the following attractive properties. First, it uses just a few cache entries to cache a few specially-crafted rules. A unique aspect of our proposal is that we do not necessarily cache an exact rule from the rule set. Instead, we cache independently constructed rules that are derived from the se-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMETRICS'07, June 12–16, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-639-4/07/0006 ...$5.00.

[1] In [7], Estan and Varghese report that DRAM speeds improve $7\% \sim 9\%$ per year while wire speeds improve 100% per year.

[2] TCAMs installed on a line card typically cost hundreds of dollars, and can be more expensive if we target higher wire speeds.

Rule I :	$(F_1 \in [30, 70]) \wedge (F_2 \in [40, 60]) \rightarrow permit$
Rule II :	$(F_1 \in [10, 80]) \wedge (F_2 \in [20, 45]) \rightarrow permit$
Rule III :	$(F_1 \in [25, 75]) \wedge (F_2 \in [55, 85]) \rightarrow permit$
Rule IV :	$(F_1 \in [0, 100]) \wedge (F_2 \in [0, 100]) \rightarrow deny$

Table 1: A rule set of 4 rules. Rules ordered by priority.

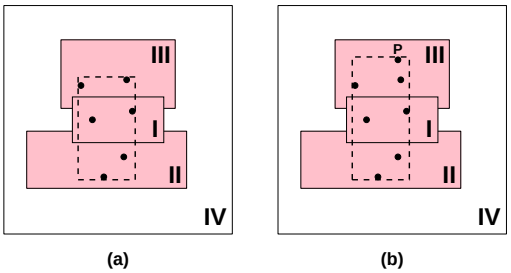

Figure 1: Caching an independently defined and dynamically evolving rule based on the rule set in Table 1.

mantics of the rule set. In order to preserve correctness, we ensure that such rules preserve *semantic equivalence* of the classification task. Second, the cached rules *evolve* over time. This rule evolution process is driven by (changing) characteristics of incoming traffic that are continuously learned by smart rule cache.

Use of fast caches for fast packet classification is naturally appealing and has been studied in the past, e.g., flow cache schemes [28, 4]. (In this paper, a *flow* corresponds to a set of all packets with the same *projection*, and the projection of a packet is defined as the d-tuple consisting of the values of the d packet header fields specified in the rule set.) In flow cache schemes, the cache is used to store the projection and decision of recently observed packets, with the expectation of speeding up the classification of succeeding packets with the same projections. Given that most flows are short-lived [19], it is not uncommon for individual routers to observe millions of concurrent flows [8] and we expect this number to only grow with time. Hence, a large flow cache would be necessary to achieve high and stable cache hit ratios when using such flow cache schemes. For instance, using 16K cache entries, the flow cache scheme proposed in [28] delivers a cache miss ratio of 8% on a sample trace with less than 14,000 concurrent flows. A more recent flow cache scheme [4] uses a 4KB size cache and reports a cache miss ratio of 4.85% on a sample trace containing up to 567 concurrent flows.

1.1 A motivating example

In this section, we illustrate our proposed smart rule cache approach through a simple example. Through this process, we also present an intuitive understanding why such an approach is naturally superior to flow cache schemes. Consider the rule set shown in Table 1, which is also pictorially illustrated in Figure 1. In the figure, the two packet header fields, F_1 and F_2, are represented along x and y axes, respectively. The shaded boxes correspond to rules whose decision is `permit` whereas the white boxes correspond to rules whose decision is `deny`. In the scenario depicted in Figure 1(a), there are six flows observed by the router, each represented by a corresponding dot. Each of Rules I, II, and III matches two flows. We now make three observations:

Cache rules instead of flows If we cache any one of the first three rules (instead of caching any of these flows), a greater fraction of packets will be classified using the cache.[3] This simple observation is reinforced by recent study [5], which reports a strong Zipf-like pattern in the usage of rules in rule sets of a tier-1 ISP, where a very small number of rules match most of incoming traffic. Moreover, cache hit ratio can also be much more stable. Because a popular rule in cache can match a series of flows and hence continues to remain in cache. In contrast, a flow cache may suffer severe thrashing. Such stability also implies enhanced robustness against malicious attacks. Although an attacker can forge a large number of short-lived flows to occupy and thrash a flow cache, it is much harder for the forged flows to match sufficiently many rules that are not needed by legitimate flows with sufficiently many hits. Even if the attacker may manage to figure out the rule set, it is extremely difficult for the attacker to obtain accurate real time knowledge of concurrent legitimate flows. Section 3.4 presents quantitative analysis on the security property of our smart rule cache.

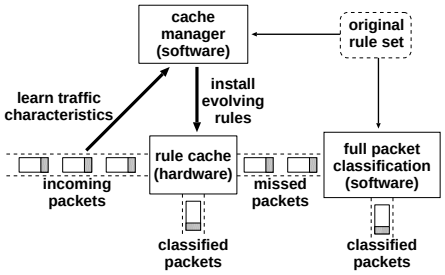

Figure 2: Framework of the smart rule cache.

Construct new rules for better cache performance If we construct a new rule, Rule X: $(F_1 \in [32, 55]) \wedge (F_2 \in [23, 68]) \rightarrow permit$, as illustrated by the dashed box in Figure 1(a), this single rule is able to match all six flows and execute the same action. Thus, caching this single new rule is adequate.

Evolve cached rules over time Now consider the scenario (as shown in Figure 1(b)) where a new flow, P, starts. Rule X will not match this flow. But we now construct another new rule, Rule Y: $(F_1 \in [32, 55]) \wedge (F_2 \in [23, 80]) \rightarrow permit$, as illustrated by the dashed box in Figure 1(b), and this new rule will continue to match all the seven flows. Thus, by evolving the cached rule (from Rule X to Rule Y) based on incoming traffic pattern, we can continue to match a great fraction of the incoming traffic with a single rule.

Based on these insights, we propose smart rule cache, where the classification task can occur in two stages, as shown in Figure 2. The first stage occurs in the small on-chip *rule cache*, which is composed of a few registers and corresponding hardware logic. Each rule cache entry stores an evolving rule and the hardware logic is used to match packets against the stored rule. Rule cache entries are organized in such a way that allows parallel search across all cached rules. The search ends with either the right decision or a cache miss, within one clock cycle. A *cache manager* module, implemented in software, is responsible for creating and continuously updating (i.e., evolving) the rules in cache. The goal of the cache manager is to minimize the number of packets that are not classifiable by the rules in cache, which are then pass to the second stage of the classification process, where they are matched against the entire original rule set by a full-fledged backup classifier, preferably implemented in software. While this software classification is a slower operation, our results using real traffic traces and rule sets from a tier-1 ISP indicate that a good cache manager design would require less than 0.07% of packets to take this slower path.

[3] Caching a flow and caching a rule both involve caching some values of those relevant packet header fields plus the decision. Therefore, the cache space per entry is comparable.

1.2 Challenges and results

Although the basic idea is conceptually clear, a number of key problems remain to be addressed.

(1) What (not which!) rules should be placed in the cache? In the motivating example in Figure 1, we have only created an evolving rule with `permit` as its decision. But in general, the cache manager can create rules with any decision to effectively reduce cache miss ratio. For example, we may create and cache `deny` rules to quickly deny a lot of malicious flows.

(2) How should rules in cache evolve in response to incoming traffic pattern changes?

(3) How can we guarantee the semantic integrity of the rule cache? Namely, for each incoming packet, how can we ensure that the decision output by the rule cache is always consistent with the original rule set? In flow cache, this is not a problem. But in rule cache, this issue needs to be carefully handled due to the priority-based ordering among rules. For example, caching Rule IV only in Table 1 suffices to match all the flows but gives the wrong decision.

(4) How can we smooth out the effect of cache management delay on cache hit performance? To minimize the cost, we only require low cost and slow memory for cache management. Therefore, cache management delay can be long (compared with the packet classification speed we target). The updated rule cache will not be available until after cache management. This means potentially decreased cache hit ratios during cache management delays.

In this paper, we present effective solutions to these design problems and evaluate the performance of our smart rule cache using real rule sets and traffic traces from a tier-1 ISP. We show that even for backbone routers carrying 10^5 concurrent flows, a small rule cache composed of just a few entries has been enough to deliver stable cache hit ratios above 99.93%. Such a small cache can be easily implemented in network processors to perform wire speed packet classification, at negligible cost. For 40Gbps OC-768, the volume of missed traffic is less than 0.03Gbps, which can be easily classified using a software classifier. Both the software classifier and the software cache manager can be implemented in low cost DRAM. Given its negligible implementation cost and superior performance, we believe our smart rule cache represents a cost efficient solution for wire speed packet classification. Moreover, we believe the value of this solution will only increase as the gap between wire speeds and memory access speeds keeps widening.

1.3 Roadmap

The rest of the paper is organized as follows. We first present preliminaries of packet classification in Section 2. The basic design of smart rule cache is then described in Section 3. Some effective optimization techniques are proposed in Section 4. We evaluate the performance of our smart rule cache using real traffic traces and real rule sets from a tier-1 ISP and present the results in Section 5. After reviewing related work in Section 6, we conclude the paper in Section 7.

2. PRELIMINARIES

A rule set is an ordered set $R = \{r_1, r_2, \cdots, r_n\}$ of rules. Each rule r_i is composed of two parts: a *predicate* and a *decision* (or *action*). The predicate is a conjunction of d literals defined over d packet header fields. In the most generalized form, each literal can be written as a range literal $F_j \in [l_j, h_j]$, where F_j denotes a packet header field. A rule r_i defined over d packet header fields is thus written as $\bigwedge_{j=1}^{d} (F_j \in [l_j, h_j]) \rightarrow decision$.

The industry standard of packet classification comes from Cisco Access Control Lists (ACLs) [1]. Currently, the predicate of each rule may specify a literal on each of the following five packet header fields: source IP address, destination IP address, source port, destination port, and protocol type. For convenience, we define the *projection* of a packet to be the d-tuple consisting of the packet's d header fields specified in the rule set. A rule and a packet are considered to *match* if the conjunctive predicate of the rule is evaluated to be `true` on the projection of the packet. If a rule is the first rule in the rule set that matches a packet, the action it specifies is performed on the packet.

Either explicitly or implicitly, rule sets contain a default rule that matches every incoming packet. If none of the preceding rules matches a packet, the action of the default rule is performed on the packet. Thus, each rule set covers the entire d-dimensional space defined over the d packet header fields specified in that rule set. The domain of each dimension is the domain of the corresponding packet header field. For example, the dimension corresponding to the 32-bit source IP address field has a domain of $[0, 2^{32} - 1]$.

Within this d-dimensional space, the conjunctive predicate of each rule delimits a d-dimensional hypercube, which we refer to as the *definition region* of the rule. We can think of the decision of a rule as a "*color*" that colors the definition region of that rule. For simplicity, we refer to it as the color of that rule. A rule set as an ordered set of rules essentially defines a coloring of the entire d-dimensional space, which we refer to as the *semantics* of the rule set. The projection of a packet/flow can be viewed as the coordinate of a specific point in the d-dimensional space, which we often use to represent the packet/flow. (Recall that a *flow* corresponds to a set of all packets with the same projection.) Each point in the d-dimensional space may be contained in the definition region of multiple rules. The color of a point is defined to be the color of the first rule whose definition region contains that point.

As we have pointed out in Section 1, we need to ensure that the rules stored in the rule cache are consistent with the rule set in semantics. To facilitate the enforcement of this semantic integrity, we need an efficient data structure to represent the rule set's semantics for verification. In this paper, we use such an efficient data structure called *pruned packet decision diagram (PPDD)*. Given a rule set, we obtain its PPDD by trimming its *standard packet decision diagram (SPDD)*, which is proposed by Liu and Gouda in [15]. The SPDD f of a rule set defined over packet header fields $F_1, F_2, \cdots, F_d$ is a directed tree that has the following properties.

1. Each node v in f has a label $F(v)$. If v is a leaf node, $F(v)$ specifies an action. If v is an internal node, $F(v)$ specifies a packet header field.

2. Each internal node v has a set $E(v)$ of outgoing edges pointing to its children and only one incoming edge from its parent. Each edge $e \in E(v)$ has a label $I(e)$, which denotes a non-empty subset of the domain of field $F(v)$. In general, $I(e)$ can be represented as a set of non-overlapping ranges. For any two edges $e \neq e'$ in $E(v)$, $I(e) \cap I(e') = \phi$. Meanwhile, $\cup_{e \in E(v)} I(e)$ is the entire domain of the packet header field $F(v)$ (denoted by $D(F(v))$. Namely, the labels of v's outgoing edges form a partition of $D(F(v))$.

3. On the path from the root to any leaf node (which is referred to as a *decision path*), there are exactly d internal nodes. The label of the ith internal node denotes the ith packet header field F_i, i.e., the ith dimension of the d-dimensional space. Recall that the label of the leaf node denotes the decision. The decision path, denoted by $v_1 e_1 v_2 e_2 \cdots v_d e_d v_{d+1}$, actually represents the rule $\bigwedge_{i=1}^{d} (F_i \in I(e_i)) \rightarrow F(v_{d+1})$.

For the example rule set in Table 2, its SPDD is given in Figure 3(a). To facilitate discussion, we start with a more regular form of SPDD as shown in Figure 3(b). Compared with the original form of SPDD in Figure 3(a), the regular form of SPDD possesses the additional property that *the label of each edge denotes a single*

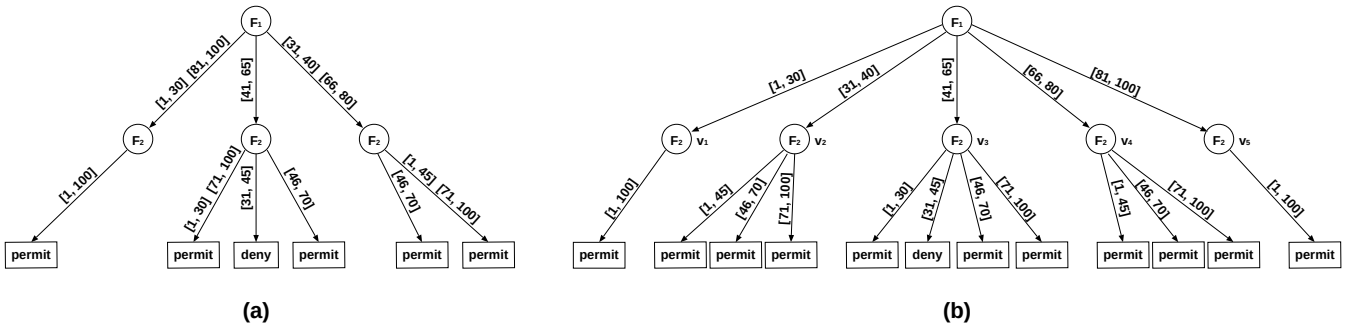

Figure 3: SPDD of the rule set in Table 2.

r_1:	$(F_1 \in [31, 80]) \wedge (F_2 \in [46, 70]) \rightarrow permit$
r_2:	$(F_1 \in [41, 65]) \wedge (F_2 \in [31, 50]) \rightarrow deny$
r_3:	$(F_1 \in [1, 100]) \wedge (F_2 \in [1, 100]) \rightarrow permit$

Table 2: An example rule set.

range. In the sequel, we use "SPDD" to denote the regular form of SPDD for simplicity.

Let F_i denote the ith dimension of the d-dimensional space. In general, each node v_i in a decision path $v_1e_1v_2e_2\cdots v_de_dv_{d+1}$ can be viewed as representing the d-dimensional hypercube:

$$H_{v_i} = \left(\bigwedge_{j=1}^{i-1}(F_j \in I(e_j))\right) \bigwedge \left(\bigwedge_{j=i}^{d}(F_j \in D(F_j))\right).$$

When context is clear, we use "node v" and "the hyper-cube represented by node v" interchangeably for ease of presentation. It is not hard to verify that for any internal node v in the SPDD, v's children form a partition of H_v. Furthermore, it can be verified that all the leaf descendants of v also form a partition of H_v. As a special case, all the leaf nodes in the SPDD form a partition of the entire d-dimensional space, which is represented by the root node of the SPDD. Recall that each leaf node is labeled with a decision. *Together, all the leaf nodes actually define a coloring of the d-dimensional space, which is consistent with the semantics of the rule set.* To verify this semantic integrity of the SPDD, we refer interested readers to [15], which also contains a detailed algorithm for building the SPDD of a given rule set.

Given the semantic integrity of SPDD, if needed we can classify any packet by checking through a decision path from the root to some leaf node. At the ith internal node v_i on the path, we follow the outgoing edge whose label contains the value of field F_i in the packet header. Let δ denote the number of ranges denoted by the outgoing edges. The number of memory accesses needed to pick the right outgoing edge is bounded by $O(\delta)$. The number of memory accesses needed to classify a packet is thus bounded by $O(d\Delta)$, where Δ is the maximum δ value over all nodes in the SPDD. In the regular form of SPDD, Δ is the maximum fanout of any node in the SPDD.

As the size of SPDD can be potentially large for large rule sets, we propose to obtain the PPDD of a rule set by trimming its SPDD. Our proposed algorithm is presented in Section 4. As we will see, PPDD preserves the semantic integrity of SPDD but contains fewer and shorter decision paths. Therefore, PPDD can also be used to classify each incoming packet (using $O(d\Delta)$ memory accesses), and its average performance is much better than SPDD.

Computing and optimizing the PPDD is a one-time preprocessing task before packet classification. The PPDD remains valid throughout the packet classification process until the semantics of the rule set has changed. In practice, rule sets are not modified very frequently, especially compared with the classification speeds we target. Therefore, the time spent on building the PPDD should not raise any concern on the packet classification performance of smart rule cache. Nonetheless, we point out that our algorithm for trimming SPDD to obtain a PPDD is quite simple and efficient.

In this paper, our primary concerns are cache hit ratio and hardware cost. To help deliver high and stable hit ratios, we would rather spend enough preprocessing time to build as good a PPDD as possible. As we will see in Section 3, the semantic integrity of the smart rule cache is ensured by making the stored rules semantically consistent with the SPDD/PPDD. To improve cost efficiency, if necessary low cost DRAMs can be used to store the computed PPDD as well as other cache management related data structures. Actually, all these data structures are stored in low cost DRAMs in our evaluation. Thus, our results demonstrate the performance of smart rule cache in such a cost efficient solution.

3. DESIGN

Our smart rule cache design consists of two parts: a small *rule cache* (the hardware component) and a *cache manager* (the software component). The rule cache is a small number of on-chip cache entries each storing an evolving rule. Each cache entry consists of a register storing the evolving rule and some simple logic for matching incoming packets against the stored rule. The cache entries are designed to match each incoming packet in parallel. Synchronized with the network processor, the rule cache is able to report either a cache miss or the right decision on the packet in a single network processor cycle. Such a simple hardware design of the rule cache is presented in Section 3.3. This small rule cache is the only additional hardware needed by our smart rule cache design. Its size and simplicity make it easy to implement in network processors at negligible cost.

The core part of smart rule cache is the cache manager. On one hand, its effective and efficient management of the rule cache decides the cache hit ratios that can be delivered. Basically, the cache manager decides cache hit performance by placing the right rules into the rule cache and dynamically evolving those rules in response to incoming traffic pattern changes. On the other hand, as the cost of the rule cache is negligible, the overall cost of smart rule cache is largely decided by the cost of implementing the cache manager. Thus, it is critical to design a cost efficient cache manager that requires as little additional resource as possible. As we will see in Section 3.1 and Section 3.2, our design of the cache manager requires nothing more than a small amount of low cost memory such as DRAM. Through evaluation using low cost DRAM-based

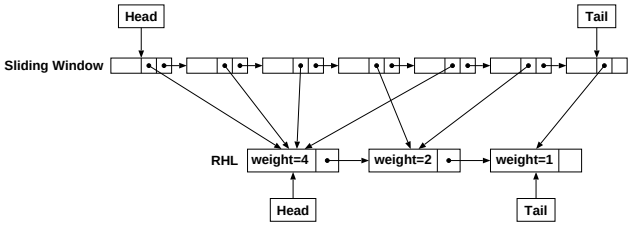

Figure 4: Data structures of smart rule cache.

systems, we demonstrate that smart rule cache is able to deliver extremely high hit ratios on real traffic traces and real rule sets obtained from backbone routers of a tier-1 ISP. Nonetheless, users are free to equip line cards with more powerful network processors and more fast memories to achieve even better performance.

To achieve good performance, the cache manager needs to collect sample packets to acquire knowledge about incoming traffic. We discuss detailed sampling strategies in Section 5.4. Following each traffic sampling is cache management. The cache manager conducts relevant statistics on the sample packets stored in a *sliding window*, which contains the most recent w sample packets (w is the sliding window size). In particular, the cache manager needs to find out all distinct flows and their frequency (which we will refer to as *weight*) in the sliding window. The cache manager uses this flow weight statistics to (1) maintain a list of evolving rules and (2) determine which rules should be switched into/out of the rule cache in order to maximize cache hit ratio.

In this section, we first present relevant data structures for cache management in Section 3.1 and then present detailed algorithms for cache management in Section 3.2. A simple hardware design for the rule cache in Section 3.3. We conduct a preliminary quantitative analysis on the security property of smart rule cache in Section 3.4.

3.1 Data structures

Sliding window: The data structure of the sliding window is straightforward – a First-In-First-Out (FIFO) queue of the w sample packets in the sliding window (as shown in Figure 4) is most appropriate. For each sample packet, its corresponding element in the queue records its projection and whether it is a cache hit or cache miss.

Evolving rules: The cache manager maintains a data structure called *regular hyper-cube list (RHL)*, which is of central importance in our design. Basically, each RHL element is an evolving rule to be placed into the rule cache. The RHL is *regular* in that it possesses the following key properties.

(I) *Each RHL element represents an evolving rule whose definition region is a d-dimensional hyper-cube.* When context is clear, we use "hyper-cube", "evolving rule", and "RHL element" interchangeably for ease of presentation.

(II) *Each hyper-cube in the RHL is colored by one single color in the coloring of the d-dimensional space defined by the original rule set.* Thus, by assigning each evolving rule that corresponding color, it is guaranteed that each evolving rule can be stored in a single entry in the rule cache and is semantically consistent with the original rule set.

(III) *Each sample packet in the sliding window is assigned to one evolving rule that matches it.* This ensures the RHL contains all the sampled information. The *weight* of each evolving rule is defined to be its number of assigned sample packets. To keep track of this assignment, we add a pointer to each sample record, pointing to the RHL element it is assigned to, as shown in Figure 4.

(IV) *Evolving rules either have the same action or are non-overlapping.* This greatly simplifies cache management, because the ordering of evolving rules in the rule cache is not important and hence we can place each evolving rule in an arbitrary cache entry. As we will see shortly, this also greatly simplifies the hardware design of rule cache. Because it guarantees that if multiple cache entries match the same packet, they must have the same decision.

The data structure of an evolving rule stores its range along each dimension, color, weight, cache entry index (if it is in cache) and its current position in the RHL (for use in cache management). Intuitively, we should try to maximize the total weight of those evolving rules in cache. We thus sort the RHL in non-increasing order of weight. Assume the rule cache consists of m entries. Property IV allows us to simply cache the first m elements of the RHL, and the semantic integrity of the rule cache is guaranteed.

3.2 Cache management

To be precise, cache management refers to the operations performed by the cache manager to update relevant data structures and the rule cache after obtaining a new sample packet. Here, we present a detailed description of these cache management operations.

Delete the oldest sample: On obtaining a new sample packet, we first remove the oldest sample packet from the sliding window. Following its pointer to the evolving rule H it is assigned to, we decrement the weight of H by one. These operations take $O(1)$ time.

(1) If the weight of H comes down to zero, it is removed from the RHL, which also takes $O(1)$ time. If H is currently in cache, its cache entry is replaced with the first evolving rule H' that is currently not in cache (if such an H' exists). In the worst case, locating H' in the RHL takes $O\left(\min(m, n)\right)$ time, where n is the length of the RHL. In our evaluation, we have observed that n almost never exceeds 10 and hence locating H' can be done very quickly.

(2) If the weight of H is still positive, we move H toward the tail of the RHL until the weight of its successor (if any) is no larger than its own weight. In the worst case, this position adjustment operation takes $O(n)$ time. If H is originally in cache (i.e., top m in the RHL) but not top m in the RHL after position adjustment, we should place the new mth evolving rule H' into the cache entry of H. In particular, when moving H toward the tail of the RHL, if H is currently the mth element and is about to switch with the $(m+1)$th element, we place the $(m+1)$th element into the cache entry of H.

Insert the new sample: After removing the oldest sample packet from the sliding window, we append the new sample packet to the tail of the sliding window, which takes $O(1)$ time. Then, we check through the RHL to find the first evolving rule H that matches the new sample packet.

(1) If such an H is found, its weight is incremented by one and we assign the new sample packet to H. To keep the RHL sorted by weight, we move H toward the head of the RHL until the weight of its predecessor is no less than its own weight. If H is currently not in cache but ranks top m in the RHL after position adjustment, we should place H into the cache entry of the new $(m+1)$th evolving rule. In particular, when moving H toward the head of the RHL, if H is currently the $(m+1)$th element and is about to switch with the mth element H', we place H into the cache entry of H'.

(2) If none of the evolving rules already matches the new sample packet, we need to obtain an evolving rule that matches the new sample packet in order to preserve property III. There are two possible ways to achieve that: expanding an existing evolving rule or creating a new evolving rule. We prefer to cover sample packets

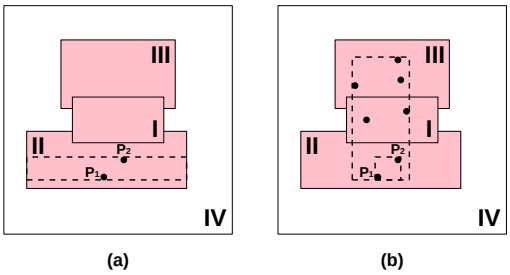

Figure 5: An example of maximally/minimally expanding an evolving rule to cover new sample packets, based on the rule set in Table 1.

using as few evolving rules as possible. Because intuitively that will enable a small cache to cover as many incoming flows as possible. For the same reason, we also prefer to cover new sample packets with the topmost evolving rules. Therefore, we go through the RHL and check each evolving rule to see if it can be expanded to match the new sample packet while preserving properties I, II and IV. If none of the existing evolving rules can be expanded, we create a new evolving rule matching exactly the new sample packet only and append it to the tail of the RHL. It takes $O(d)$ time to create a new evolving rule and $O(1)$ time to append it to the RHL. As we have discussed in Section 2, $d = 5$ in Cisco ACL, which is the *de facto* industry standard.

Expanding a hyper-cube H to cover a point p while preserving property I is straightforward. Assume on the ith dimension, the range of H is denoted by $[l_i, h_i]$ and the coordinate of p is x_i. If $x_i < l_i$, we decrease l_i to x_i. If $x_i > h_i$, we decrease h_i to x_i. If $x_i \in [l_i, h_i]$, there is no need to expand H along the ith dimension. In total, expanding H to contain p takes $O(d)$ time.

> **Discussion:** Here, we adopt an expand-remove approach to evolving existing rules. While we do not shrink rules, a rule can be removed if it no longer matches any sample packet in the sliding window (e.g. due to traffic pattern changes). The reason of adopting this expand-remove approach is two-fold. First, shrinking rules is much more sophisticated to implement. Second, as we will show in Section 3.4, the key security properties of smart rule cache directly stem from this expand-remove approach.
>
> When expanding a hyper-cube H, we minimally expand it along each dimension to obtain a hyper-cube H' that contains the new sample packet. However, one may suspect that, if instead we maximally expand H along each dimension, then hopefully the expanded hyper-cube H' will be able to match more incoming packets later on. To better understand the design choice, it is worth noting that we are actually solving an online optimization problem, where the input is unpredictable incoming traffic and the objective is to optimize cache hit ratio. While such an aggressive expanding strategy has some merits in its own right, we prefer the design choice of minimally expanding hyper-cubes because that leaves us more flexibility on subsequently expanding existing evolving rules.
>
> For example, let us again consider the rule set in Table 1. Initially, there is no evolving rule and here comes the first flow (denoted by P_1 in Figure 5). The cache manager creates an evolving rule H to cover precisely that point only. When the second flow (denoted by P_2 in Figure 5) appears, let us assume we maximally expand H along X-axis to be the dashed box in Figure 5(a). Later on, there start five other flows, denoted by those unlabeled points in Figure 5(b). We will not be able to further expand H to cover these new flows, due to the semantic integrity constraint imposed by property II. At least one more evolving rule has to be created to cover these new flows. In contrast, if upon appearance of the second flow we expand H to be the small dashed box in Figure 5(b), later on we shall be able to further expand H to be the large dashed box in Figure 5(b), which covers all the flows. One evolving rule is enough.

Verifying if the expanded hyper-cube H' satisfies property IV is not difficult. We can simply go through the RHL and check each evolving rule to see whether it overlaps with H' but has a different color from H'. In total, this operation takes $O(nd)$ time.

Now it only remains to verify whether H' satisfies property II. This is where the SPDD of the rule set can be used. Recall that the leaf nodes of an SPDD form a partition of the entire d-dimensional space and define a coloring that is consistent with the semantics of the original rule set. Therefore, property II is preserved if and only if all the leaf nodes overlapping with H' have the same color as H'. This can be easily verified by traversing the SPDD and check the color of each leaf node overlapping with H'. However, this straightforward solution can potentially take a long time and hence result in a long cache management delay. We propose effective optimization techniques in Section 4.

3.3 Hardware design of the rule cache

For each incoming packet, the rule cache should either report a cache miss or output the correct decision on that packet. For wire speed packet classification, we require this to be done within one network processor cycle. In this section, we present a simple hardware design of the rule cache to achieve this design objective. Basically, each cache entry is composed of two parts: a register for storing an evolving rule and some simple logic for matching packets against the stored rule. Cache entries are organized in such a way that allows parallel search within one processor cycle.

First of all, each cache entry should be able to determine whether the stored rule matches the incoming packet or not. Testing whether a hyper-cube (i.e., rule) contains a certain point (i.e., packet) is actually a special case of testing overlapping hyper-cubes, since a point can also be expressed as a "hyper-cube". Testing overlapping hyper-cubes can be implemented using the more basic function of testing overlapping ranges: two hyper-cubes overlap if and only if they overlap on every dimension. Consider two hyper-cubes H_1 and H_2. Assume their ranges along the ith dimension are $[a_i, b_i]$ and $[x_i, y_i]$, respectively. Testing whether $[a_i, b_i]$ and $[x_i, y_i]$ overlap can be done with the simple *Overlapping Ranges Tester (ORT)*, as shown in Figure 6. Using one ORT for testing each dimension, testing overlapping hyper-cubes can be easily done within one processor cycle using d ORTs in parallel. Such an *Overlapping Hyper-cubes Tester (OHT)* design is shown in Figure 7.

Assume the value of ith field in the incoming packet header is x_i and the range specified by the stored rule on that field is $[a_i, b_i]$. The entire design of a cache entry is shown in the dashed box in Figure 8. The decision of the cached rule is stored as a k-bit positive integer (e.g. $A_1, A_2, \cdots, A_k$ in Figure 8). `0` is reserved for `cache miss`. Each one of the k bits $A_1, A_2, \cdots, A_k$ is logically ANDed with the output of the OHT. This yields the final k-bit output of that cache entry, which is either `cache miss` (i.e., all `0`s) if

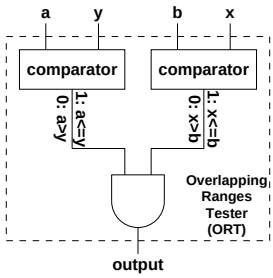

Figure 6: ORT design.

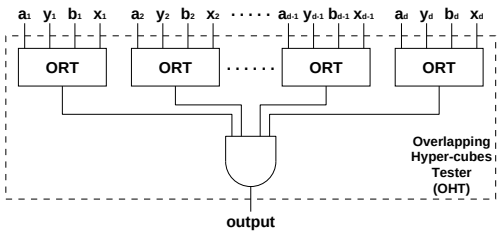

Figure 7: Hardware design of OHT.

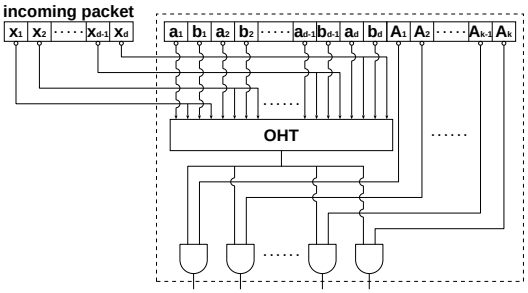

Figure 8: Hardware design of a cache entry.

the output of its OHT is `0` or the stored rule's decision $A_1 A_2 \cdots A_k$ if the output of its OHT is `1`.

It is easy to input a packet to all cache entries in parallel. Given simultaneous outputs from all cache entries, we need to ensure that the rule cache eventually presents the right output. Property II and property IV of the RHL play a critical role in making a simple solution possible. As we have discussed in Section 3.2, property IV of the RHL guarantees that if two or more stored rules match a packet, they must have the same decision. Moreover, property II of the RHL guarantees that their decision is consistent with the rule set. Thus, we can simply bit-wise OR the k-bit output from all cache entries, which yields the final output of the rule cache. If none of the cache entries matches the incoming packet, the rule cache outputs `cache miss` (i.e., all `0`s). Otherwise, the rule cache will output the right decision.

The entire rule cache works as follows. The projection of the incoming packet is input to all cache entries simultaneously. The cache entries try to match the incoming packet in parallel and simultaneously report their matching result (i.e., either a decision or a `cache miss`), which are bit-wise ORed to yield the final output of the rule cache. If we only need a few cache entries, such a simple and small rule cache can be easily implemented in network processors at negligible cost. Synchronized with the network processor, the rule cache is able to output its matching result within one network processor cycle.

An issue that has not been addressed so far is cache update. Basically, to update a cache entry we only need to rewrite its register, which stores the evolving rule. Since the rule cache is synchronized with the network processor, this can be easily done within a processor cycle. Only one packet will not be able to match the cache entry being updated. As each cache management execution updates at most one cache entry and lasts for no less than one millisecond in our evaluation, the percentage of packets that are affected by cache update is very low. Assuming OC-768 (40Gbps) and a packet size of 500 bytes, ten thousand packets will pass through during a one-millisecond cache management delay. That is, only one out of ten thousand packets will be affected by cache update. Nonetheless, if a disturbance-free solution is preferred, we can use two identical rule caches to achieve seamless hot-swap. The two rule caches can be controlled using a simple 0/1 switch. Directing incoming packets to one of them automatically disables the other for update.

3.4 Security analysis

As a preliminary security analysis, we hereby derive an upper bound on the additional cache miss ratio of legal traffic that can be caused by an attacker. To derive such a bound, we conduct our analysis under the following adversary model.

Adversary model: We assume an adversary who is perfectly informed of the rule set, cache size, cache management algorithm and concurrent flows in a real time manner. We also assume the adversary can arbitrarily decide the content of those sampled attacking packets to baffle the cache manager, based on its perfect knowledge and hence equally perfect prediction of the cache management consequence. Furthermore, we assume the tough situation where the rule cache never has enough entries to accommodate all evolving rules.

Analysis: When combating such a perfectly informed adversary, a commonly employed weapon is randomness. Here, our cache manager employs a random sampling strategy. Using this random sampling strategy, the probability with which a flow will be sampled is precisely the percentage of its traffic volume in the aggregate traffic traversing the router line card. While we assume the adversary can arbitrarily decide the content of those sampled attacking packets to baffle the cache manager, the presumed bottomline of randomness prevents the adversary from deciding which packets are going to be sampled by our random sampling process.

Let us first look at the moment when the adversary is about to launch its attack. Suppose there are $n > m$ RHL elements, R_1, R_2, $\cdots$, R_n, sorted in non-increasing order of their weight. Let w_1, w_2, $\cdots$, w_n denote their normalized weight, respectively. The first m RHL elements will be cached and the cache hit ratio of legal traffic is given by $\sum_{i=1}^{m} w_i$.

Now, suppose the adversary injects attacking traffic at its maximum possible rate, and its generated attacking traffic accounts for a percentage of δ in the aggregate traffic. Recall that the cache manager prefers to associate sampled packets with existing RHL elements, in non-increasing order of their weight. New RHL elements are created only if it has to. Suppose we now have $l \geq n$ RHL elements, R'_1, R'_2, $\cdots$, R'_l, sorted in non-increasing order of their weight. For each R_i, let w_i^+ and w_i^- denote the portion of its normalized weight contributed by sampled legal packets and sampled attacking packets, respectively. Consider any R_i of the n existing RHL elements. Let us assume it is (possibly expanded into) the new RHL element R'_j. Due to the dilution caused by the attacking traffic, the random sampling strategy makes $w_j^+ = (1-\delta)w_i$. The cache hit ratio of legal traffic achieved by this new RHL is given by $\sum_{i=1}^{m} \frac{w_i^+}{1-\delta}$.

Among the top m new RHL elements, R'_1, R'_2, $\cdots$, R'_m, let us assume without loss of generality that k of them, R'_{i_1}, R'_{i_2}, $\cdots$, R'_{i_k}, were not among the original top m RHL elements, R_1, R_2, $\cdots$, R_m. Accordingly, there must be k other new RHL elements, R'_{j_1}, R'_{j_2}, $\cdots$, R'_{j_k}, that are not currently among top m but were originally among top m. Since the original RHL is sorted in non-increasing order of weight, we know for any $1 \leq d \leq k$, it must be the case that $w_{i_d}^+ \leq w_{j_d}^+$. Similarly, since the new RHL is also sorted in non-increasing order of weight, it must be the case that

$$w_{i_d}^- + w_{i_d}^+ \geq w_{j_d}^- + w_{j_d}^+ \geq w_{j_d}^+ \Longrightarrow w_{i_d}^- \geq w_{j_d}^+ - w_{i_d}^+.$$

Summing this inequality over all $d \in [1, k]$ gives us the following key inequality:

$$\sum_{i=1}^{m} [(1-\delta)\, w_i] - \sum_{i=1}^{m} w_i^+ = \sum_{d=1}^{k} \left(w_{j_d}^+ - w_{i_d}^+\right) \leq \sum_{d=1}^{k} w_{i_d}^- \leq \sum_{i=1}^{l} w_i^- = \delta.$$

Dividing both sides by $1-\delta$ leads us to our final conclusion:

$$\sum_{i=1}^{m} w_i - \sum_{i=1}^{m} \frac{w_i^+}{1-\delta} \leq \frac{\delta}{1-\delta} \quad (1)$$

The left side of Equation (1) is precisely the increase in the cache miss ratio of legal traffic, caused by the adversary, which is at most $\frac{\delta}{1-\delta}$. For instance, if the attacking traffic generated by an attacker accounts for 10% of the aggregate traffic traversing a router line card, the resulting increase in the cache miss ratio of legal traffic is at most 11.1%.

4. SPDD OPTIMIZATION

As we will see in Section 5.1, the SPDD of large rule sets can be potentially very large if not built in an appropriate way. Verifying property II by traversing a large SPDD can result in long cache management delays, which may decrease cache hit ratio. In this section, we propose effective techniques for optimizing SPDD. In Section 4.1, we present an algorithm for trimming the SPDD without violating its semantic integrity. The obtained data structure is called *Pruned Packet Decision Diagram (PPDD)*. In Section 4.2, we propose that an appropriate ordering of packet header fields for building the SPDD can lead to a much smaller SPDD and PPDD.

4.1 Pruned packet decision diagram (PPDD)

Our motivating observation is that we may significantly decrease the number of SPDD nodes we have to visit in order to verify property II, by employing various early detection techniques. The first early detection technique is quite straightforward. Assume we are currently at node u in the SPDD. For each child v of node u, we need to explore the subtree rooted at v (denoted by T_v) only if H_v overlaps with the expanded hyper-cube H'. Because v's leaf descendants form a partition of H_v. If H_v does not overlap with H', none of v's leaf descendants can overlap with H'. Therefore, there is no need to explore T_v. For example, assume H' is defined by $(F_1 \in [45, 70]) \wedge (F_2 \in [35, 45])$. In the SPDD in Figure 3(b), there is no need to explore the subtrees rooted at v_1, v_2 and v_5, since they cannot contain any leaf node overlapping with H'.

Now suppose H_v overlaps with H' and hence we may need to explore T_v. The following two early detection techniques can be employed to further avoid exploring T_v. (1) If H_v is colored by a single color that is the same as H', we can determine without exploring T_v that T_v cannot contain any leaf node with a color different from H'. For example, assume that H is defined by $(F_1 \in [45, 60]) \wedge (F_2 \in [10, 25])$ with decision `permit` and the expanded H' is defined by $(F_1 \in [25, 60]) \wedge (F_2 \in [10, 25])$ with the same decision. In the example SPDD in Figure 3(b), there is no need to explore T_{v_1} and T_{v_2}, since H_{v_1} and H_{v_2} are both colored by the same single color `permit`. (2) If H_v is colored by a single color that is different from H', then T_v must contain some leaf node that overlaps with H' and has a different color from H'. Thus, we can immediately fail the verification of property II without exploring T_v. For example, assume that H is defined by $(F_1 \in [45, 60]) \wedge (F_2 \in [35, 45])$ with decision `deny` and H' is defined by $(F_1 \in [45, 70]) \wedge (F_2 \in [35, 45])$. In the example SPDD in Figure 3(b), we can immediately fail the verification of property II without exploring T_{v_4} since H_{v_4} is colored by a single color `permit` that is different from H'.

```
int SPDD2PPDD (node root)
if (root is a leaf node)
  root.color = root.label;
  return root.color;
prune = true;
color = ∞;
for (each child of root)
  if (color == ∞)
    color = SPDD2PPDD(child);
    if (color == -1)
      prune = false;
  else if (color != SPDD2PPDD(child))
    prune = false;
if (!prune)
  root.color = -1;
  return -1;
for (each child of root)
  dispose child;
root.color = color;
root.label = root.color;
return root.color;
```

Table 3: Algorithm for trimming SPDD to obtain PPDD.

The above two early detection techniques require some additional information: for each node v in the SPDD, we need to know whether H_v is colored by a single color and if yes what is that color. This information can be easily obtained through a simple extension of the SPDD. In particular, we mark each node v in the SPDD with an additional field `color`. Assume the decisions specified in the rule set are encoded as non-negative integers. If H_v is colored by more than one color, we assign `-1` to the `color` field of node v. Otherwise, the `color` field of node v is assigned the color that colors H_v. This additional information can be easily computed in a single bottom-up pass of the SPDD. In particular, the `color` field of each leaf node v is the same as its label $F(v)$, which denotes a decision. If all the children of an internal node v have the same `color` value, node v is also assigned the same `color` value. Otherwise, the `color` field of node v is assigned `-1`.

According to the early detection techniques described above, we will explore the subtree T_v rooted at a node v only if node v's `color` value is `-1`. This implies that we can safely remove the descendants of a node v if v's `color` field value is not `-1`. That will make node v a leaf node and we label node v with its `color` value, which is the same as the decision of all the leaf descendants of node v. This trimming operation can also be done in a single bottom-up pass of the SPDD and can be easily implemented as a simple recursive function, as shown in Table 3.

Our discussion so far has been based on the regular form of SPDD. However, recall that there is only one difference between the regular form of SPDD and the original form of SPDD: the label of each edge can contain multiple ranges in the latter but contains only one range in the former. Since the trimming algorithm in Table 3 ignores the label of edges, it is clearly applicable to the original form of SPDD as well. The PPDDs obtained by trimming the SPDDs in Figure 3 are shown in Figure 9.

4.2 Ordering packet header fields

Based on the PPDD we now have, some further optimizations are definitely possible. For example, in the PPDD in Figure 9(b), v_1 and v_2 can be merged into one node, v_4 and v_5 can be merged

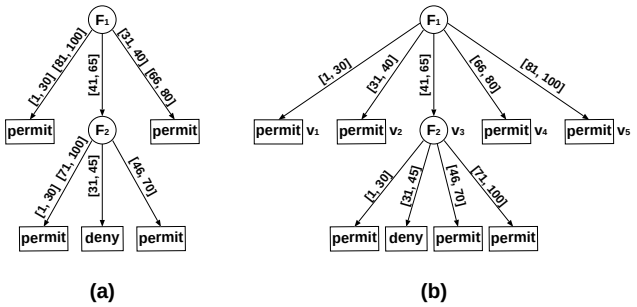

Figure 9: PPDDs obtained by trimming SPDDs in Figure 3.

r_1:	$(F_1 \in [1, 100]) \wedge (F_2 \in [1, 25]) \rightarrow permit$
r_2:	$(F_1 \in [1, 100]) \wedge (F_2 \in [26, 50]) \rightarrow deny$
r_3:	$(F_1 \in [51, 100]) \wedge (F_2 \in [51, 75]) \rightarrow permit$
r_4:	$(F_1 \in [76, 100]) \wedge (F_2 \in [76, 100]) \rightarrow deny$
r_5:	$(F_1 \in [1, 100]) \wedge (F_2 \in [1, 100]) \rightarrow permit$

Table 4: An example rule set.

into one node, and the right two children of v_3 can be merged into one node. For another example, we can merge v_1, v_2, v_4, and v_5 into a single node in the original form of the SPDD. Although that does not reduce the number of ranges, that does reduce the number of nodes. However, as we have limited space, we prefer to leave such less important optimizations in the extended version of this paper. Instead, we stick to the regular form and present another more fundamental optimization technique: we can significantly reduce the size (number of nodes) of a PPDD by building the SPDD according to an appropriate ordering of the packet header fields. Notice that in the regular form, the number of ranges is the same as the number of edges, which is the number of nodes minus one.

Consider the rule set in Table 4. If we use F_1 as the first dimension and F_2 as the second dimension, the resulting SPDD contains 15 nodes as shown in Figure 10(a). This SPDD cannot be pruned and hence the PPDD is of the same size. Interestingly, if we switch the order of F_1 and F_2, the resulting SPDD will contain only 11 nodes (shown in Figure 10(b)). After trimming the first four leaf nodes, the new PPDD will contain only 7 nodes. As we will see in Section 5, the effect of a good ordering of packet header fields on real rule sets (which typically use five packet header fields) can be much more significant than its effect on such a 2-dimensional simple rule set.

In general, it is not easy to figure out the optimal ordering of packet header fields that will lead to a PPDD of minimum size. However, as we have discussed in Section 2, building the PPDD is a one time preprocessing task and it is worth spending time on building as good a PPDD as we can. Given that, a straightforward solution is to try out as many possible orderings as we can and keep the minimum size PPDD we have so far. In our evaluation, for real rule sets containing thousands of rules, it takes only a few seconds to build the SPDD and PPDD according to a certain ordering of packet header fields. Given five packet header fields, there are $5! = 120$ possible orderings, which take about ten minutes to check.

In future work, we are interested to search for more efficient algorithms for finding the optimal ordering of packet header fields. For practical interest, after checking a number of real rule sets containing up to thousands of rules, we have found the following ordering of packet header fields to be quite effective: (1) protocol type; (2) source IP address; (3) destination IP address; (4) source port; and (5) destination port. For a considerable portion of the real rule sets, this (not necessarily the best) ordering already reduces the PPDD size by $1 \sim 2$ orders of magnitude. For the other rule sets, their PPDD size is reduced by at least a factor of 2. We report detailed evaluation results in Section 5.1.

	Trace 1	Trace 2	Trace 3	Trace 4
Trace length (sec)	4793	5008	4645	5016
Number of flows	9.95M	5.86M	9.67M	10.83M
Max # concurrent flows	164420	143166	103591	176160
Max flow length	11821	28119	1485	24041
(pkt, sec)	334.10	519.47	164.76	520.30
Avg flow length	8.20	8.66	6.91	9.22
(pkt, sec)	62.65	92.05	39.46	72.00
% TCP flows	92.52	92.97	93.37	91.62
% UDP flows	6.28	6.26	6.06	7.64
% other flows	1.20	0.77	0.57	0.74

Table 5: Statistics of sampled traffic traces (1/21/2006).

5. EVALUATION

We evaluate the performance of our smart rule cache using 4 real traffic traces and 10 real rule sets obtained from a tier-1 ISP backbone network. The traffic traces are collected by NetFlow using $1/\alpha$ packet sampling at a number of links connected to edge routers, where α is a constant. For each flow, NetFlow maintains a record containing a number of fields including the source and destination IP addresses, source and destination routing prefixes, source and destination ASes, source and destination port numbers, the protocol type, type of service, flow starting and finishing timestamps, number of bytes and number of packets transmitted. Each traffic trace lasts about one day. The real rule sets include packet filters configured at corresponding router interfaces. Each rule set contains hundreds or thousands of rules. The decision of rules is either `permit` or `deny`. In Section 5.5, we will extend these rule sets to have more diversified decisions and evaluate the performance of smart rule cache using such extended rule sets.

In the sampled traces, the maximum number of concurrent flows is less than 10^5. As we target more than 10^5 concurrent flows, we compact the sampled traces into shorter traces by possibly advancing flows such that the maximum number of concurrent flows is great than 10^5. Let the start time of a sampled trace be 0. If the start time of a flow is t_0, its start time in the compacted trace will be $t_0' = t_0$ `MOD` 4500 (in seconds). Its end time in the compacted trace will be $t_1' = t_0' + T$, where T is the duration of the flow. Some statistics of the resulting traces are given in Table 5. As we can see, most flows are likely to be short-lived flows, which represents a serious challenge to cache schemes. We believe this characteristics of the traces makes our evaluation results more reliable.

5.1 PPDD

We conduct simulations on the rule sets to evaluate the effectiveness of using a better ordering of packet header fields and the effectiveness of using PPDD. The default ordering we use is: (1) source IP address; (2) destination IP address; (3) source port; (4) destination port; (5) protocol type. (It is worth emphasizing that, although the rule sets we use for evaluation are defined over this standard 5-tuple, all our proposed techniques of smart rule cache are applicable to rule sets defined over any number of packet header fields.) Through simulations, we find the following ordering performs quite well: (1) protocol type; (2) source IP address; (3) destination IP address; (4) source port; (5) destination port. To evaluate the effectiveness of a better ordering, we report the PPDD size (i.e., number of nodes in the PPDD) achieved by both orderings in Table 6.

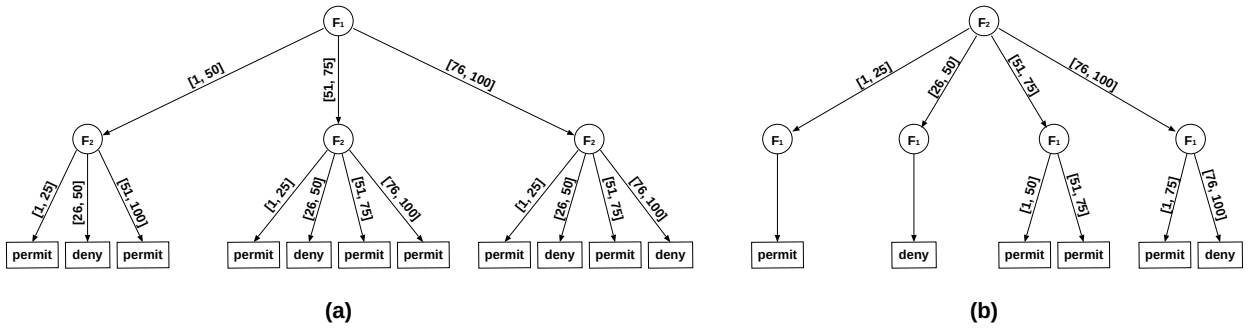

Figure 10: SPDD and PPDD of the rule set in Table 4 derived from different orderings of packet header fields.

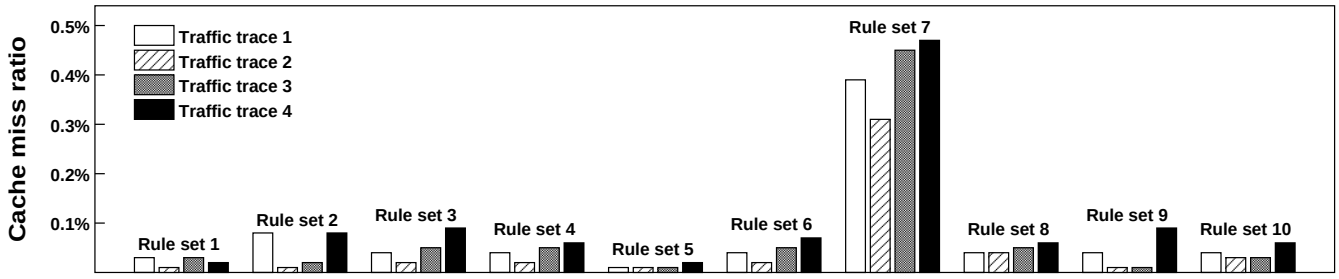

Figure 11: Cumulative cache miss ratios achieved using a single cache entry and a sliding window of 1024 packets.

Rule set	# rules	Default ordering	Better ordering
1	373	6703	3336
2	617	218602	8449
3	378	6099	3002
4	226	4003	1981
5	391	349990	14897
6	203	3736	1869
7	2755	27865	13768
8	666	203101	7378
9	539	5389	2677
10	628	6539	3187

Table 6: PPDD size achieved by the default ordering and a better ordering of packet header fields.

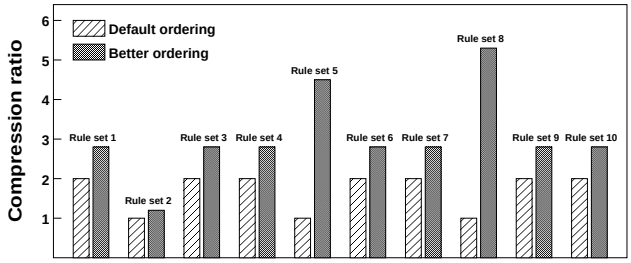

Figure 12: Compression ratios achieved by the default ordering and a better ordering of packet header fields.

For rule sets 2 and 8, the better ordering reduces their PPDD size by two orders of magnitude. The PPDD size of rule set 5 is reduced by one order of magnitude. For the other rule sets, the better ordering reduces their PPDD size by at least a factor of 2. Although these real rule sets each contains as many as thousands of rules, with the better ordering of packet header fields, their PPDD size never exceeds 15K. In our simulations, we use a sliding window size of 1024 packets[4] and we find that the length of the RHL never exceeds 10. Both are much smaller than the PPDD. Therefore, the memory requirement of smart rule cache is dominated by PPDD and hence is very small.

To evaluate the effectiveness of using PPDD instead of SPDD, we define the ratio between the size of an SPDD and the size of its PPDD as the *compression ratio* and report the compression ratios achieved by both orderings in Figure 12. It is clear that PPDDs are much smaller than SPDDs. Moreover, using the better ordering of packet header fields universally enhances the effectiveness of using PPDD.

[4]In our experiments, we vary the sliding window size from 1 to 4096 and do not observe perceptible change in the performance of smart rule cache when the sliding window size if between 64 and 4096. To be conservative, we have been using a sliding window size of 1024 for all our experimentation.

5.2 Cache management delay

As we have previously discussed, cache management delay can potentially impact cache hit ratio. Because during cache management, incoming packets are still matched against the old rule cache. Only after cache management is done, the updated rule cache is available for matching incoming packets. To obtain reliable simulation results, we carefully simulate the cache management delay for each new sample packet.

In our simulations, we keep track of two clocks simultaneously. One clock is the *physical clock* of the machine running our simulations, which can be read through a system call. The other clock we maintain is the *logical clock* of the traffic trace – each packet in the trace has its time of emergence in the trace. Right before cache management starts, we read the physical clock time t_1 and record the current logical time t_0 in the traffic trace. Upon completion of cache management, we read the physical clock time again and record it as t_2. $\Delta t = t_2 - t_1$ is taken as the cache management delay. We do not update the rule cache until logical time $t' = t_0 + \Delta t$

in the traffic trace. Packets emerging before t' in the traffic trace are matched against the old cache. In our simulations, we record the delay of every cache management execution. The observed average cache management delays are no less than one millisecond.

5.3 Results

To conduct an extensive evaluation of smart rule cache, we run each traffic trace through each rule set and simulate smart rule cache at per packet level in that context. Using a single cache entry and a sliding window of 1024 packets, we report the cumulative cache miss ratios observed on individual pairs of traffic trace and rule set in Figure 11. The cache miss ratios are calculated after a warm-up stage, which lasts for five minutes and one million packets, whichever comes later. The cumulative miss ratio of a traffic trace accounts for all packets after the warm-up stage. As we can see in Figure 11, the cache miss ratios observed on all 40 pairs of traffic trace and rule set never exceed 0.5%. Actually, on all rule sets except rule set 7, the cache miss ratios never exceed 0.1%. This represents a decrease in cache miss ratio by two orders of magnitude, compared with the cache miss ratios reported in [28, 4].

Note that, the use of sampled traffic traces does not invalid our results. We demonstrate this via simulations based on "enriched" traffic traces. Given the sampling factor of α, we keep the inter-packet interval of each flow unchanged and evenly inject $\alpha - 1$ packets between each pair of successive packets of each flow. This gives us a traffic trace with α times as many packets as the original trace. We observed same cache hit ratios on enriched traces as we observed on sampled traces.

5.4 Tuning sampling strategy

Although the cache miss ratios reported in Figure 11 have been extremely low, we still find the relatively higher cache miss ratios observed on rule set 7 quite intriguing. So we ask the question "Is there any specific reason underlying this, other than the maybe special characteristics of rule set 7?" After careful analysis and extensive experiments, the answer turns out to be "yes". The sampling strategy plays a decisive role there. For the results in Figure 11, our sampling strategy is to immediately collect the next incoming packet after cache management is completed. This straightforward strategy seems not bad, as it allows the cache manager to sample incoming traffic as frequently as possible. However, sampling more frequently does not mean the cache manager will obtain more useful knowledge. To effectively evolve the rules to capture missed flows, the cache manager needs to sample missed packets. Packets hitting the rule cache add no additional useful knowledge about incoming traffic. Because the cache manager ignores incoming traffic during cache management, sampled packets are its only source of knowledge. As the cache miss ratio has been quite low, such a blind sampling strategy makes the cache manager oblivious of missed flows with high probability. Therefore, the rules cannot be effectively evolved to capture the missed flows and hence cache miss ratio cannot be further reduced.

To further decrease the cache miss ratios and to verify the correctness of this understanding, we have designed and evaluated a smarter sampling strategy. After cache management is completed, we wait for a fixed number of packets (which we refer to as *sampling interval*) before collecting the next sample packet.If some packet during the sampling interval results in a cache miss, we take that packet as our next sample and restart cache management immediately.

Using rule set 7 and traffic trace 2, we evaluate the performance of smart rule cache with different sampling intervals and report the results in Figure 13. With an appropriate choice of sampling interval, this smart sampling strategy reduces the cache miss ratio by two orders of magnitude. Using traffic trace 2 and a sampling interval of 512 packets, we also evaluated the performance of smart rule cache with smart sampling on other rule sets. The observed cache miss ratios are between 0.0158% and 0.0003%. Compared with the cache miss ratios reported in [28] and [4], this represents a decrease in cache miss ratio by $2 \sim 4$ orders of magnitude. That means the workload on the full-fledged packet classifier is reduced by $2 \sim 4$ orders of magnitude, which in turn means potentially shorter packet classification delays experienced by missed packets and the possibility of using less efficient but cheaper solutions.

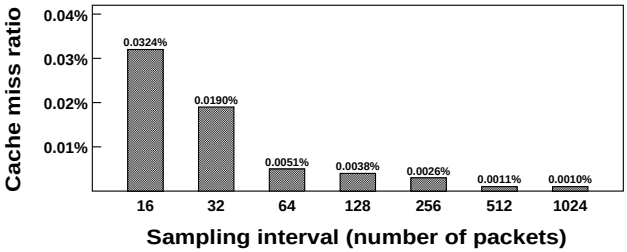

Figure 13: Effect of sampling interval on cache miss ratio. Cache size = 1. Sliding window size = 1024.

Rule set	Number of cache entries			
	1	2	3	4
3	4.19×10^{-1}	1.41×10^{-1}	1.84×10^{-3}	6.52×10^{-4}
4	3.85×10^{-1}	3.00×10^{-2}	1.17×10^{-3}	6.26×10^{-4}
6	3.89×10^{-1}	3.03×10^{-2}	1.17×10^{-3}	6.07×10^{-4}

Table 7: Cumulative cache miss ratios observed on different rule sets with different number of cache entries.

5.5 More complicated rule sets

So far our simulation has been based on real rule sets used for packet filtering, each specifying two possible decisions: `permit` and `deny`. While packet filtering is a globally deployed application of wire speed packet classification, there are also many other applications such as QoS and security that specify much more diversified decisions. To evaluate the effectiveness of smart rule cache on such applications, we also conducted simulations based on such rule sets. As we do not have access to any such real rule sets, we extend the real rule sets we have been using by randomly assigning one of 1024 different decisions to each rule. In practice, it is unlikely that more than 1024 different decisions will be specified.

Using traffic trace 2 and a sampling interval of 1024 packets, we evaluate the performance of smart rule cache on the extended rule sets. For rule sets 1, 2, 5, 7, 8, 9 and 10, the cache hit performance of smart rule cache using one cache entry has degraded very slightly, by a negligible amount. For rule sets 3, 4 and 6, we do observe some impact on the performance of smart rule cache. We present the cumulative cache miss ratios observed with different numbers of cache entries in Table 7. As we can see, using as few as 4 cache entries, our smart rule cache is still able to reduce cache miss ratio to the order of 10^{-4}.

We also conducted the same simulation for smaller numbers of different decisions. To reduce cache miss ratio to the order of 10^{-4}, the number of cache entries needed appears to grow no faster than logarithmically. For example, for rule set 3 with up to 2, 4, 16 and 1024 different decisions, we need 1, 2, 3 and 4 cache entries, respectively.

6. RELATED WORK

Packet classification on multiple fields was first studied in [13] and [23]. Since then, there have been two lines of research on designing efficient packet classification schemes. A long thread of research [13, 23, 9, 24, 10, 27, 20, 3, 2, 26, 21, 12, 25, 5, 11] is devoted to designing efficient algorithms for packet classification. The other thread of research focuses on designing efficient packet classification schemes based on TCAMs [16, 30, 22, 29, 14, 6].

Instead of proposing new packet classification schemes, in this paper we focus on designing a high performance cache scheme for cost efficient wire speed packet classification. Two flow cache schemes have been previously proposed in [28] and [4], respectively. These flow cache schemes cache recently observed flows to speed up the classification of succeeding packets in those flows. However, the increasingly large number of concurrent flows witnessed by backbone routers present serious threat to the performance of flow cache schemes. Based on the notion of rule evolution, our proposed smart rule cache has been able to handle many more concurrent flows, requires much smaller cache size and delivers much higher cache hit ratios.

In [5], Cohen and Lund propose to reorder rules based on popularity. Although their goal is to reduce the expected time of sequentially searching through a rule set to classify packets, this technique can actually be used to reorder rules and then cache the top m rules. In that sense, their proposal shares some common observation with rule cache. However, simply reordering given rules is still far from our smart rule cache. In smart rule cache, rules in cache are not necessarily present in the given rule set and dynamically evolve in response to incoming traffic pattern changes. Use of such independently defined and constantly evolving rules is decisive to the success of smart rule cache.

More recently, Hamed *et al.* [11] propose to add some "early reject" rules to the beginning of firewall packet filters, in pursuit of the same goal of reducing the expected time needed to sequentially search through a rule set. Compared with the proposal by Cohen and Lund, Hamed *et al.* have gone one step further in that the early reject rules they add are not necessarily in the rule set. However, the key idea of dynamically evolving rules is still absent. Moreover, in identifying early reject rules, they have not been able to take a systematic approach based on the semantics of the rule set. Instead, their approach are based on the specific values that are explicitly specified in the rules. This greatly limits the flexibility and effectiveness of added early reject rules.

7. CONCLUSIONS

Cost efficient wire speed packet classification is an important topic of research. On one hand, the only widening gap between wire speeds and memory access speeds represents an increasingly tough challenge to software solutions. On the other hand, the rapidly increasing wire speeds, rule set size and range specifications make TCAM-based hardware solutions increasingly expensive. In this paper, we propose to use a smart on-chip rule cache with a low cost backup classifier in DRAM as a viable, cost efficient option. A key contribution of this work is the notion of a few evolving rules that reside in the rule cache. Although the evolving rules depend on the given rule set, it is usually not identical to any individual rule in the rule set. In addition, they evolve with changes in incoming traffic patterns. Through evaluation based on real traffic traces and real rule sets from backbone routers of a tier-1 ISP, we demonstrate our smart rule cache can achieve stable cache miss ratios at the order of 10^{-4}, using just a few cache entries. Such a small cache can be easily implemented in network processors to keep up with wire speeds, at negligible cost. As cache miss ratios are extremely low, missed packets can be classified using a low cost backup classifier. We believe the value of our smart rule cache design will only increase with wire speeds and TCAM costs.

8. REFERENCES

[1] *Controlling Network Access With Access Control Lists*, 2004. http://cisco.com/univercd/cc/td/doc/product/lan/cat6000/mod_icn/fwsm/fwsm_2_2/fwsm_cfg/mngacl.pdf.

[2] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: is there an alternative to CAMs? In *IEEE INFOCOM*, 2003.

[3] F. Baboescu and G. Varghese. Scalable packet classification. In *ACM SIGCOMM*, 2001.

[4] F. Chang, W. C. Feng, and K. Li. Approximate caches for packet classification. In *IEEE INFOCOM*, 2004.

[5] E. Cohen and C. Lund. Packet classification in large ISPs: Design and evaluation of decision tree classifiers. In *ACM SIGMETRICS*, 2005.

[6] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *ACM SIGMETRICS*, 2006.

[7] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *ACM SIGCOMM*, 2002.

[8] C. Estan and G. Varghese. Data streaming in computer networking. In *Workshop on Management and Processing of Data Streams*, 2003.

[9] P. Gupta and N. McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, August 1999.

[10] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects*, 1999.

[11] H. Hamed, A. El-Atawy, and E. Al-Shaer. Adaptive statistical optimization techniques for firewall packet filtering. In *IEEE INFOCOM*, 2006.

[12] M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell. Directions in packet classification for network processors. In *NP2 Workshop*, 2003.

[13] T. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM*, 1998.

[14] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with Ternary CAMs. In *ACM SIGCOMM*, 2005.

[15] A. X. Liu and M. G. Gouda. Removing redundancy from packet classifiers. Technical Report TR-04-26, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas, U.S.A., June 2004.

[16] H. Liu. Efficient mapping of range classifier into Ternary-CAM. In *Hot Interconnects*, 2002.

[17] R. K. Montoye. Apparatus for storing "don't care" in a content addressable memory cell. United States Patent 5,319,590, June 1994.

[18] M. H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. *Journal of Algorithms*, 21(3):629–656, November 1996.

[19] C. Partridge. Locality and route caches. In *NSF Workshop on Internet Statistics Measurement and Analysis*, February 1999.

[20] L. Qiu, G. Varghese, and S. Suri. Fast firewall implemention for software and hardware based routers. In *IEEE ICNP*, 2001.

[21] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, 2003.

[22] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended tcams. In *IEEE ICNP*, 2003.

[23] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *ACM SIGCOMM*, pages 191–202, September 1998.

[24] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Packet classification using tuple space search. In *ACM SIGCOMM*, 1999.

[25] D. E. Taylor and J. S. Turner. Scalable packet classification using distributed crossproducting of field labels. In *IEEE INFOCOM*, 2005.

[26] J. van Lunteren and T. Engbersen. Fast and scalable packet classification. *IEEE Journal on Selected Areas in Communications*, 21(4):560–571, 2003.

[27] T. Y. Woo. A modular approach to packet classification: Algorithms and results. In *IEEE INFOCOM*, 2000.

[28] J. Xu, M. Singhal, and J. Degroat. A novel cache architecture to support layer-four packet classfication at memory access speeds. In *IEEE INFOCOM*, 2000.

[29] F. Yu and R. H. Katz. Efficient multi-match packet classification with TCAM. In *Hot Interconnects*, 2004.

[30] F. Zane, G. Narlikar, and A. Basu. Coolcams: Power-efficient tcams for forwarding engines. In *IEEE INFOCOM*, 2003.