Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers

Yadi Ma
University of Wisconsin
Madison, WI, USA
yadi@cs.wisc.edu

Suman Banerjee
University of Wisconsin
Madison, WI, USA
suman@cs.wisc.edu

Shan Lu
University of Wisconsin
Madison, WI, USA
shanlu@cs.wisc.edu

Cristian Estan*
NetLogic Microsystems
Mountain View, CA, USA
cestan@netlogicmicro.com

ABSTRACT

We present a software-based solution to the multi-dimensional packet classification problem which can operate at high line speeds, e.g., in excess of 10 Gbps, using high-end multi-core desktop platforms available today. Our solution, called Storm, leverages a common notion that a subset of rules are likely to be popular over short durations of time. By identifying a suitable set of popular rules one can significantly speed up existing software-based classification algorithms. A key aspect of our design is in partitioning processor resources into various relevant tasks, such as continuously computing the popular rules based on a sampled subset of traffic, fast classification for traffic that matches popular rules, dealing with packets that do not match the most popular rules, and traffic sampling. Our results show that by using a single 8-core Xeon processor desktop platform, it is possible to sustain classification rates of more than 15 Gbps for representative rule sets of size in excess of 5-dimensional 9000 rules, with no packet losses. This performance is significantly superior to a 8-way implementation of a state-of-the-art packet classification software system running on the same 8-core machine. Therefore, we believe that our design of packet classification functions can be a useful classification building block for RouteBricks-style designs, where a core router might be constructed as a mesh of regular desktop machines.

Categories and Subject Descriptors

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

General Terms

Algorithm, Design, Performance

Keywords

Packet classification, Storm, parallelism

1. INTRODUCTION

The problem of packet classification has a significant role in many important networking functions, both at the edge and in the core. Data packets received at a router's input port are classified to determine an action to be performed for each such packet. The action may include forwarding the packet, routing the packet to a particular application, providing expedited delivery of the packet, discarding the packet, etc. Packet classification is needed for services that require network traffic (i.e., packets) to be distinguished and isolated into different flows for suitable processing. There are many examples of such services: packet filtering, to deny all packets from a known source; policy routing, to route all VoIP traffic over a separate network; and traffic shaping, to ensure that no one source overloads the network. To implement these services, a packet classifier is provided with a set of rules along with associated actions to be taken for a packet that matches these rules.

Many common packet classification techniques utilize multiple packet header fields. Firewalls and NATs, for example, define rules based on the 5-tuple of IP addresses, port numbers, and protocol fields. The time or storage complexity of multi-dimensional packet classification grows exponentially with the number of fields [11] and makes it challenging to develop a fast-performing solution in software. To alleviate this problem, many classification systems rely on specialized hardware, such as Ternary Content Addressable Memory (TCAMs), that allow simultaneous match of a single packet against a large number of rules within a fixed number of clock cycles.

In this paper, we ask a complementary question. Given the emerging capabilities of the desktop platforms today, with multiple cores and large memory banks, what kind of packet classification speeds would be achievable by such platforms? Our question is partially motivated by the recent work in the RouteBricks project [4], where the authors have explored the feasibility of building high-speed

*Work done while this author was at UW-Madison.

SIGMETRICS'10, June 14–18, 2010, New York, New York, USA.
Copyright 2010 ACM 978-1-4503-0038-4/10/06 ...$10.00.

software routers using a mesh of off-the-shelf desktop-based servers that parallelize router functionality both across multiple servers and across multiple cores within a single server.

In particular, we ask the following question: *If we were to implement classification using state-of-the-arts desktops, given that it may find applications in router design (e.g., RouteBricks), then what classification speeds can be achieved on it, by designing a fully software-based classification system that can exploit the degree of parallelism provided by this particular platform?* In answering this question, we do not define a new classification *algorithm* that would outperform prior best known examples, e.g. HyperCuts [13]. Instead, our approach is to design a new *software system* for classification that takes existing classification algorithms and partitions critical tasks into multiple threads that effectively leverage desktop platforms to meet various computation and memory access needs. Our approach utilizes a common idea of caching — an idea that has been widely adopted by the networking and systems community. Techniques such as packet caches identify popular packet headers and their corresponding actions for faster IP lookups and packet classification, while techniques such as Smart Rule Cache (SRC) [5] cache popular rules in small amounts of TCAMs that are assumed to be present in router linecards. In this paper, we build upon the basic observation of SRC, and design a **S**oftware-**o**nly **r**ule cache for **m**ulti-core desktop platforms, or *Storm*.

1.1 Design overview of Storm

In designing Storm, we leverage the prior observation [5], that in any ruleset, there often exists a relatively small subset of popular rules. [1] SRC as a possible classification system based on this observation is shown in Figure 1, in which a small number of popular rules are stored in a TCAM. Each incoming packet is first classified against all rules in the TCAM. If the packet does not match any of the rules in the TCAM, it is forwarded to the network processor that compares the packet in software against all rules in the ruleset. The performance gains in SRC are achieved when a large fraction of packets are classified in the TCAM itself.

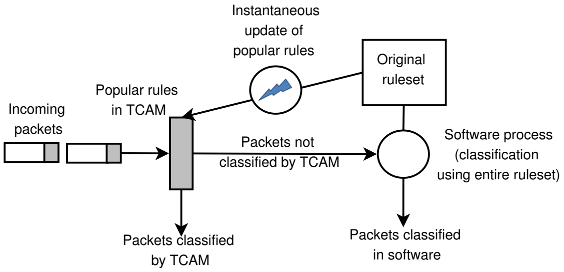
Instantaneous update of popular rules
Original ruleset
Popular rules in TCAM
Incoming packets
Packets not classified by TCAM
Software process (classification using entire ruleset)
Packets classified by TCAM
Packets classified in software

Figure 1: Framework of SRC, its use of TCAMs, and simplifying assumptions.

The SRC work assumes that the TCAM can be updated with the new set of rules instantaneously, and so the newly updated popular rules can be applied to the very next packet. Since the work was based on packet level simulations, the real impact of these issues on end-to-end performance metrics such as realizable traffic throughput was not investigated.

The goal of this work is to design an overall system that can make a rule caching system practical and usable in a state-of-the-art multi-core desktop platform, *a platform that does not carry any specialized matching hardware like TCAMs.* We do this by leveraging the availability of multiple cores in the system to partition various tasks of a rule caching system.

In particular, the overall approach in Storm has been to design a multi-threaded software-only design of the system that balances the throughput of multiple constituent components, such that the overall throughput of the entire system is maximized.

While our design of Storm is implemented and evaluated for a popular multi-core desktop platform, this software-only solution can be embedded in any other platform, including existing multi-core routing substrates, e.g. Cavium's family of Octeon-based network processor boards [10].

1.2 Use of parallelism in Storm

As mentioned, Storm is not a new packet classification algorithm, but rather is a parallelized software system that uses some best-known packet classification algorithm as one of its software components. Other components of Storm use the approach of matching packets against a small number of popular rules to significantly improve upon the classification speed provided by the chosen packet classification algorithm. In our work, we use HyperCuts [13] as an example of such a packet classification algorithm within Storm. The design of Storm has been able to improve the performance of HyperCuts by a factor of two or more.

Given a multi-core system, there are multiple natural ways of implementing an efficient multi-dimensional packet classification system — data parallelism, task parallelism, and pipeline parallelism. A simple form of parallelism is the data parallel approach, where each core executes an identical version of the same packet classification algorithm, e.g. HyperCuts. Incoming packets are sent to different cores in a round robin fashion. Such an approach can provide a throughput gain (over a single core system) that is proportional to the number of cores in the system. Storm, however, effectively combines all three forms of parallelism and performs significantly better than the data parallelism approach alone.

Storm is based on the idea that only a small subset of rules are likely to be popular, although the subset of popular rules can change frequently over time. Hence, it partitions the classification problem into four tasks: a) identify and cache rules that are deemed to be popular at the current instant, b) match incoming packets against these popular rules, c) perform continuous traffic sampling for continued re-computation of rule popularity, and d) perform a full-software classification using the entire set of rules, for those packets that cannot be classified using the popular rules alone.

While the SRC approach also involves the above set of tasks, its simplifying assumptions imply that tasks a) and c) are of zero cost, and task b) is implemented *in specialized hardware (TCAM).* Thus in SRC, task d) is the only software component, making multi-core systems somewhat irrelevant.

We now describe how different forms of parallelism are applied in Storm to achieve the various performance gains.

[1] The number of popular rules can be reduced even further through a set of operations that can optionally merge, add, or split existing rules in a manner that the semantics of classification is left unchanged.

- Task parallelism. In task parallelism, we identify certain software components that are completely independent of each other. In the case of Storm, the full software classifier is completely independent of the rule cache updater, and can be executed in parallel.

- Data parallelism. As described, this is a natural form of parallelism for many networking tasks, where different packets may be processed by identical task modules independently and in parallel. Given that tasks in Storm take different amounts of time and have various dependencies on other tasks, it is advantageous to dedicate additional computation resources to tasks that form a bottleneck. Hence, in some cases we dedicate multiple cores to execute the same identical task in parallel (e.g., multiple threads for the cache update task and multiple threads that perform matching against the rule cache). The number of cores for a given task is sometimes also adjusted dynamically to match throughput requirements of that task. This approach to data parallelism helps alleviate bottlenecks that might otherwise occur in the system.

- Pipeline parallelism. This is a third form of parallelism in which multiple tasks need to be executed in a specific pre-defined order for each incoming packet. For Storm, packets that cannot be matched by the rule cache, need to be then classified by the full software classifier. In such a structure, parallelism is exploited following a simple producer-consumer pattern between the two classification stages, each operating as a separate task.

Thus, Storm uses a combination of these parallelism alternatives to achieve its performance gains.

In summary, the following are the main contributions of this work:

- *A practical, multi-threaded, software-only packet classification system:* We present a software-only packet classification system that can take advantage of existing multi-core platforms. The implementation has been evaluated using real traffic traces and large real rulesets and on 8-core desktop platforms. For a ruleset consisting of more than 9000 5-dimensional rules, Storm achieves a sustained throughput of more than 15 Gbps, while a 8 simultaneous instances of HyperCuts on the same platform achieves a sustained throughput of less than 4 Gbps.

- *Design of dynamic balancing of computation resources to tasks:* A system typically has a bottleneck that determines the throughput limit. Storm has multiple tasks, and the task that is the bottleneck might be different depending on the traffic pattern and ruleset. Hence, to be continuously efficient, Storm tries to dynamically balance the amount of computation resources (threads) dedicated to each task.

2. PROBLEM STATEMENT OF PACKET CLASSIFICATION

Given a set of strictly ordered rules, the packet classification problem is to find out the first (highest priority) rule in the rule set that matches each incoming packet at a router. Each rule is associated with an action. After classification, the corresponding action will be performed for each packet.

Table 1: A simple example with 8 rules on 5 fields

Rule	F_1	F_2	F_3	F_4	F_5	Action
$R0$	000*	111*	10	*	UDP	$action_0$
$R1$	000*	10*	01	10	TCP	$action_1$
$R2$	000*	01*	*	11	TCP	$action_0$
$R3$	0*	1*	*	01	UDP	$action_2$
$R4$	0*	0*	10	*	UDP	$action_1$
$R5$	000*	0*	*	01	UDP	$action_1$
$R6$	*	*	*	*	UDP	$action_3$
$R7$	*	*	*	*	TCP	$action_4$

Table 2: A range based representation of rules in Table 1

Rule	F_1	F_2	F_3	F_4	F_5	Action
$R0$	0-1	14-15	2	0-3	0	$action_0$
$R1$	0-1	8-11	1	2	1	$action_1$
$R2$	0-1	4-7	0-3	3	1	$action_0$
$R3$	0-7	8-15	0-3	1	0	$action_2$
$R4$	0-7	0-7	2	0-3	0	$action_1$
$R5$	0-1	0-7	0-3	1	0	$action_1$
$R6$	0-15	0-15	0-3	0-3	0	$action_3$
$R7$	0-15	0-15	0-3	0-3	1	$action_4$

Suppose the rule database in a router contains a rule set of N rules, and each rule contains K fields. We consider the case where $K = 5$. The five fields are source IP address, destination IP address, source port, destination port and protocol type, respectively. There are three types of matches a field can have: exact match (protocol type), prefix match (source/destination IP address), or range match (source/destination port).

Table 1 shows a simple example with eight rules on five fields. In exact match, the header field of a packet should match the rule field exactly. For example, protocol type could be TCP or UDP. In a prefix match, the rule field should be a prefix of the header field. Suppose header field 2 of a packet is 1010. In Table 1, it matches the second field, F_2, of rule $R1$. In a range match, the header value should lie in the range specified by the rule.

If each of the header fields of a packet P matches each of the corresponding fields in a rule R, the packet P is said to match rule R. If P matches multiple rules, the first rule (with the minimum index) is returned.

Table 2 is a range representation for the set of rules in Table 1. Suppose the entire space for F_1 through F_5 is [0, 15], [0, 15], [0, 3], [0, 3], [0, 1], respectively.

3. STORM

Storm is a software-based solution of the multi-dimensional packet classification problem using multi-core desktop platforms. Although hardware implementation traditionally has a performance advantage, a pure-software implementation has the potential to become more appealing, because it is hardware-independent and thus provides better extensibility. In addition, the increasing amount of parallel computation resources available in desktop platforms provides an opportunity for software implementation to catch the line speed requirement (e.g., in excess of 10 Gbps).

3.1 Storm Design

Our design (shown in Figure 2) is guided by the three types of parallelism opportunities in Storm packet classification. The whole system includes three sets of working threads. They are:

- *Rule cache threads:* that match incoming packets against popular rules using a simple, software-based linear search (Task b). A linear search is sufficient since there are only a small number of popular rules at any time ($<$ 30).
- *Full software classifier threads:* that carry out full software-based classification (with the HyperCuts algorithm) using the entire ruleset (Task d). These threads act on the packets that could not be classified by the popular rules.
- *Sampling and rule cache updater threads:* that continuously sample incoming traffic, attempt to identify the evolving set of popular rules, and update the rule caches which are used by the rule cache threads (Tasks a and c).

The specific algorithms used to identify popular rules in the rule cache and how to evolve these rules (rule cache update task) is presented in section 3.2.

Data parallelism is leveraged by creating multiple instances of each of these threads and allowing them to operate on different incoming packets. Different threads are connected by shared queues in order to exploit the pipeline parallelism between them, e.g., the rule cache updater and the full software classification threads. The task of traffic sampling is relatively simple and is closely tied to the rule cache update component. After careful experimentation, we found it advantageous to fold it within the rule cache updater thread.

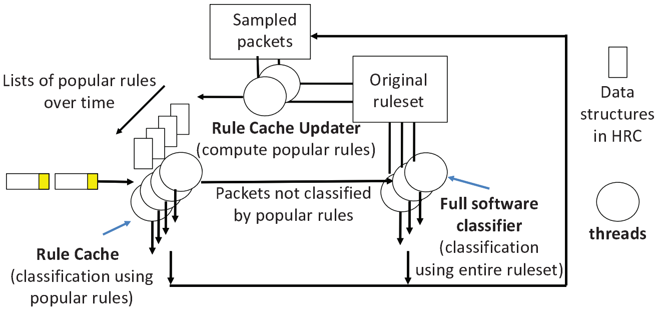
Sampled packets
Lists of popular rules over time
Original ruleset
Rule Cache Updater
(compute popular rules)
Packets not classified by popular rules
Full software classifier
(classification using entire ruleset)
Rule Cache
(classification using popular rules)
Data structures in HRC
threads

Figure 2: Architecture of Storm.

We also considered other design choices. One alternative design is shown in Figure 3. This design follows the data parallelism idea used by previous software router implementations [4, 9]. In this alternate design, we could take all the different task components of Storm, and fold them into a serial sequence of tasks. Each core implements a full version of Storm that acts on separate sets of incoming packets for data parallelism.

We prefer our design (Figure 2) for several reasons. First of all, this simple data-parallelism design does not take full advantage of Storm's three types of inherent parallelism. Second, it causes longer delay in cache-based packet classification. Under Storm, packets that miss the rule cache need a much longer processing time than packets that hit the cache. Our design leverages pipeline parallelism to guarantee short processing time for cache-hit packets. However, in the alternative design, every queuing packets will suffer a long delay and thus a high packet drop rate whenever there is a small burst of cache-miss packets. Third, some tasks are not suitable for parallelism. The sampler module that is responsible for calculating evolving rules and updating the rule cache is actually very difficult to parallel, which is not an issue in our design, but is a problem in the alternative design. Finally, our design also provides better extensibility by separating different tasks into different threads.

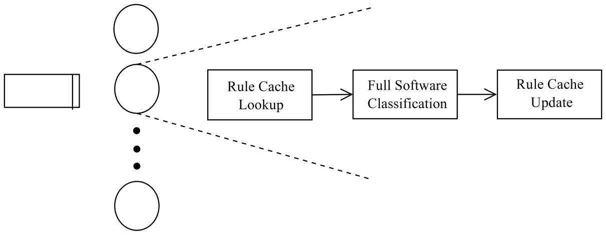

Figure 3: An alternate architecture that only exploit data parallelism.

3.1.1 *Thread Assignment*

The immediate question following our architecture design is how many threads should be assigned for the three different tasks: rule-cache-lookup, full-software-classification and rule-cache-updating. To answer this question we need to consider the underlying hardware configuration, the throughput and packet drop rate, and the implementation complexity. In the following, we present a theoretical model, followed by our static thread assignment and dynamic adjustment algorithms.

Theoretical Model.

We formalize this thread assignment problem into an optimization question with the system throughput as the optimization target. Specifically, the inverse of system throughput T can be approximated by:

$$\frac{1}{T} = \frac{d_1}{t_1} \times r + \frac{d_2}{t_2} \times (1 - r) + C,$$

where d_1 and d_2 are average delays of fast classification by rule cache and slow classification by the full software classifier. t_1 is the number of rule cache threads and t_2 is the number of full software classifier threads. N is the total number of threads available for these two stages (i.e., $t_1 + t_2 = N$). r is the average rule cache hit ratio and C is a constant number representing queuing delay, synchronization overhead, etc.

To maximize the system throughput and decide the optimal number of threads for each task, we take the partial derivative of t_1 on both sides and make it equal to 0:

$$\frac{\partial \frac{1}{T}}{\partial t_1} = -\frac{d_1 r}{t_1^2} + \frac{d_2(1-r)}{(N-t_1)^2} = 0.$$

Solving the above equation, we get

$$t_1 = N\frac{\sqrt{d_1 r}}{\sqrt{d_2(1-r)}+\sqrt{d_1 r}} \qquad (1)$$

This formula provides us with a theoretical guideline on thread assignment. Here, N is affected by the number of cores in the machine as well as the number of threads that are assigned for the rule cache updater. The cache hit ratio r is normally a stable value under our Storm algorithm. Its exact number can be affected by the rule cache updating task and the incoming packets.

We decided to use only one thread to carry out the sampling and rule cache updating task, because it is difficult to coordinate multiple threads in calculating evolving rules. We illustrate this point by an example. Suppose we have two rule cache updater threads and each of them keeps a separate copy of the evolving rules. After an update, each rule cache updater thread copies its own evolving rules to the rule cache. Initially, at time t_0, each rule cache updater thread starts with an empty evolving rule list. Suppose at time t_1, the first rule cache updater thread finishes its first update and creates an evolving rule r_1, which is copied to the rule cache. Shortly after, at time t_2, the second rule cache updater thread finishes its update and creates an evolving rule r'_1. At this point, if we copy r'_1 to the rule cache, we lose the result from the first update, i.e., r_1. Trying to merge the results will become complicated, since we may need to compare each evolving rule in a rule cache updater thread's evolving rule list to the rules in the rule cache to decide which rules are expanded and how to merge the rules. Experimental results show that using more than one rule cache updater threads does not improve cache hit ratio, rather it slows down the classification because of the rule cache updating and synchronization overhead. Since using only one rule cache updater thread already gives a cache hit ratio of above 95%, we decide to keep one rule cache updater thread.

Static thread assignments on 8-core machines.

Guided by the above theoretically analysis, we explore thread assignment on 8-core desktop platforms.

Our first decision is to control the total number of threads to be around 8 in order to avoid context switch overhead. We have decided to use only 1 thread to carry out the rule cache updating task. This gives N to be around 7.

Next, we measured the average delay for rule cache lookup, d_1, and the average delay of full software classification (HyperCuts), d_2, in our sequential implementation. They are about 200 ns and 2000 ns, respectively. We also measured the cache hit ratio for multiple rulesets and find it to be around 0.95. Plug these numbers into the above equation, we get $t_1 = 7\frac{\sqrt{200r}}{\sqrt{2000(1-r)}+\sqrt{200r}}$, between 3 and 4.

Another consideration we had is the synchronization overhead, a big component of the constant delay (C) in our formula. The queues between rule cache lookup and the full software classification are the main synchronization spots in Storm. As we will explain later, in order to minimize the synchronization overhead, we decide to make the number of cache threads an exact multiple of the number of full software classifiers, or the other way around.

Putting all these together, we get a few potential thread partitions for the three tasks in Storm: 3-3-1, 4-2-1, and 2-4-1 (the format is RuleCacheThread–FullSoftwareClassifierThread–RuleCacheUpdaterThread). As we will see in our experiments, our partitions have successfully achieved high throughput, low packet drop rates, and are overall much better than other partitions.

Dynamic thread assignment.

One concern with our static thread assignment is that it assumes stable total number of threads, N, and stable cache hit ratio, r, in our thread assignment equation 1. Although a stable N can be assumed when the machine is dedicated for routing, a stable cache hit ratio may not always be true depending on the packet workload. Therefore, we need to consider dynamically changing our thread assignment to accommodate the changing cache hit ratio.

We implement a simple dynamic thread assignment algorithm on a 8-core machine. Initially, we create 4 rule cache threads, 4 full software classification threads and 1 rule cache updating thread, and then switch between 4-2-1 and 2-4-1 accordingly. Depending on the available buffer size in the shared queues between rule cache threads and full software classifier threads, we decide to idle either 2 of the cache threads or 2 of the full software classifier threads (these threads are blocked without taking CPU resources). After each cache update, the rule cache updater thread will check the available buffer size. We choose to use the rule cache updater thread to do the check because it is independent of other threads. If we do this in a rule cache thread or a software classifier thread, it will slow down the job of the thread, which is unfair to the traffic of that thread. We define two thresholds, $r1$ and $r2$, where $0 \le r1 < r2 \le 1$. We also define the total buffer size in the shared queues to be B. If the currently available buffer size is larger than $r2 \times B$, we decide to switch to 2-4-1 by idling 2 of the rule cache threads and waking up 2 of the previously idled software classifier threads; otherwise, if the currently available buffer size is smaller than $r1 \times B$, we decide to switch to 4-2-1 by idling 2 of the software classifier threads and waking up 2 previously idled rule cache threads.

Thread assignments on other multi-core machines.

It is not difficult to extend our system to 16-core, 64-core, or other multi-core machines. Take 16-core machine for an example, according to the above equation 1, we get $t_1 = 15\frac{\sqrt{200r}}{\sqrt{2000(1-r)}+\sqrt{200r}}$, around 8. Considering the synchronization overhead, a potential thread partition for the three tasks in Storm is 8-8-1. Similar to the dynamic thread assignment algorithm, we can idle either 4 of the cache threads or 4 of the full software classifier threads and switch between 4-8-1 and 8-4-1 on a 16-core machine.

3.2 Rule Cache Updater

3.2.1 Overview of rule cache updater

The Storm packet classification occurs in two stages. The first stage occurs in the small rule cache, which is composed of a small number of rules, and the second stage occurs in a full software classifier. This high level approach is similar to SRC as shown in Figure 1. However, SRC uses a TCAM in the first stage, where the rule cache in Storm is implemented fully in software. Each entry in the rule cache stores an evolving rule. An evolving rule is a rule that is constructed based on sampled incoming traffic and updated over time.

Table 3: A simple ruleset on 2 fields.

Rule	*Field*$_1$	*Field*$_2$	Action
$R0$	1-9	4-10	$action_0$
$R1$	7-14	3-8	$action_1$
$R2$	3-11	1-6	$action_2$
$R3$	0-15	0-11	$action_3$

Linear search is used to match packets against the stored rules. No extra hardware is needed.

A *rule cache updater* thread is responsible for periodically sampling the incoming packets and updating the rules in the rule cache. While the SRC approach also needs a rule cache updater, the authors in [5] had assumed that rule cache update can occur instantaneously. The pruned packet decision diagram (PPDD) data structure used by the SRC approach turns out to be slower than adequate to keep the rule cache current. In Storm, our rule cache updater uses a mechanism that is loosely based on the HyperCuts packet classification algorithm [13]. [2] In our experiments, our new rule cache updater reduces the time for rule cache computation by two orders of magnitude when compared to the use of PPDD.

In this section, we present how to construct evolving rules and how to update the rules.

3.2.2 Construct evolving rules

To provide an example of how rule evolution is implemented, Table 3 is an illustrative example of an original ruleset having 4 rules to identify a packet based on two header fields. Each rule is associated with an action. Suppose $action_0$, $action_1$, $action_2$ and $action_3$ are 4 different actions. Figure 4 is a graphical representation of the original 4 rules. The two fields, $Field_1$ and $Field_2$ are represented along X- and Y-axes, respectively. The rectangles delineate the ranges of rules $R0$, $R1$, $R2$ and $R3$, respectively. Figure 4(a)-(f) illustrate an example of evolving rule generation, with each subfigure representing a newly sampled packet and the corresponding changes to the evolving rules, where $P1$ through $P6$ represent sampled packets and each evolving rule is represented by a dashed rectangle.

As shown in Figure 4(a), when the first packet $P1$ arrives, there is no evolving rule existing and $P1$ becomes the first evolving rule. Referring to Figure 4(b), the second packet $P2$ is sampled, having the same action as the first evolving rule. Accordingly, the rule cache updater attempts to expand the first evolving rule. Since the expanded evolving rule does not conflict with the semantics of the original rule set, the evolving rule expansion is successful as shown. When the third packet $P3$ arrives, the rule cache updater attempts to expand the first evolving. However, the proposed rule conflicts with the original ruleset. Thus, the first evolving rule can not be expanded and the second evolving rule is formed as shown in Figure 4(c). When the fourth packet $P4$ is received, the rule cache updater first attempts to expand the first evolving rule, which turns out to conflict with the original rule set. Then the second evolving rule is successfully expanded as shown in Figure 4(d). Similarly, when $P5$ arrives, the first two evolving can not be expanded and it forms the third evolving rule as shown in Figure 4(e). When $P6$ is sampled, the first two evolving rules can not be expanded while the third evolving rule is expanded successfully as shown in Figure 4(f).

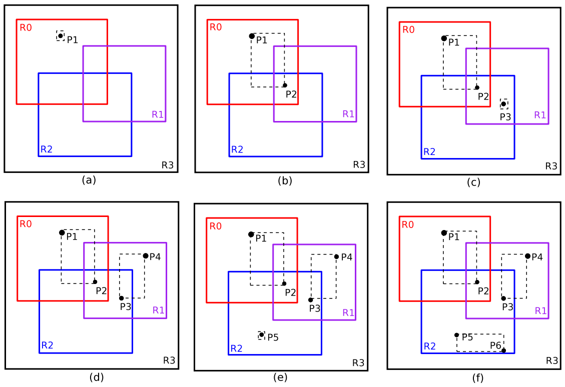

Figure 4: Constructing evolving rules, rule priority: $R0 > R1 > R2 > R3$**,** $action_0 \neq action_1 \neq action_2 \neq action_3$**.**

The evolving rules in the rule cache are required to satisfy five properties:

Each evolving rule represents a d*-dimensional hypercube.*

For the rules shown in Table 3, each evolving rule is shown and described as a 2-dimensional hypercube.

Each evolving rule is associated with a single action that is semantically consistent with the original ruleset.

By semantically consistent, we mean a packet being classified using rule cache will be associated with the same action as the packet would be matched with the full packet classifier and original ruleset.

Each sample packet in the sliding window is assigned to one evolving rule that matches it.

The weight of each evolving rule, stored in an evolving rule data structure, is defined to be its number of assigned sample packets.

Evolving rules either have the same action or are non-overlapping.

If two evolving rules overlap but have different actions, we can not decide what action should be assigned to a packet which falls in the overlapping range.

Each evolving rule lies entirely inside one of the rules in the original ruleset.

Define a d-dimensional evolving rule r to be:

$$\{[l_1, h_1], [l_2, h_2], \ldots, [l_d, h_d]\},$$

where l_i and h_i are lower and higher bounds on field i, $1 \leq i \leq d$. Define an original rule R to be:

$$\{[L_1, H_1], [L_2, H_2], \ldots, [L_d, H_d]\}.$$

[2] In particular, we use HyperCuts decision tree data structure in this rule cache updater design. This is a specific design choice in our system, but is independent of the choice of the algorithm used for full software classification (which can be any efficient algorithm, e.g., HiCuts [7], HyperCuts [13], or HyperSplit [12]).

Similarly, L_i and H_i are lower and higher bounds on field i, $1 \leq i \leq d$. We say r lies entirely inside R, if for each i, $1 \leq i \leq d$,

$$L_i \leq l_i \leq h_i \leq H_i.$$

If an evolving rule overlaps with multiple rules in the original ruleset, and these rules have the same action, the evolving rule should lie entirely inside any one of these rules. Otherwise, if these rules have different actions, the evolving rule should lie entirely inside the highest priority rule that it matches.

Restricting evolving rules to lie entirely within one rule of the original ruleset makes checking expanded rule for conflicts using HyperCuts decision tree much faster. By property 5, it greatly reduces the number of nodes to be checked in the HyperCuts decision tree. Besides, each leaf node in the HyperCuts tree contains a small and ordered list of original rules, but we usually do not need to check the complete list of rules. For example, if an expanded rule conflicts with an original rule R, then other original rules that have lower priority than R do not need to be checked. Using the five properties allows faster cache updating and hence higher cache hit ratio.

3.2.3 *Checking expanded rules for conflicts*

To implement evolving rule generation and updating, a rule cache updater thread samples and stores sampled packets which are used to update the evolving rules in rule cache. Further, the rule cache updater thread generates and stores proposed evolving rules pending determination of whether the proposed evolving rules would conflict with the five properties described above.

A sliding window, organized as a a First-In-First-Out(FIFO) queue, stores a number of recently sampled packets. The evolving rule list is a list of proposed evolving rules.

Each evolving rule stored in the evolving rule list that has been checked for conflicts will be transferred into rule cache during a cache update. Each evolving rule includes a weight field, which are updated according to incoming traffic. The rule cache updater thread uses the weight field to order a list of evolving rules in rule cache and to switch the most popular rules into rule cache, in order to maximize cache hit ratio.

The number of packets stored in sliding window is referred to as the sliding window size. Generally, the larger the sliding window size, the more the number of evolving rules. From our experiments, a sliding window size of 1024 generally result in around 20 evolving rules.

HyperCuts decision trees are constructed as stated in [13] on original rulesets. Readers familiar with HyperCuts may skip the following 3-4 paragraphs and move on to Algorithm 1. At each node in a HyperCuts tree, the set of current rules is split based on information from one or more fields in the rules. Each time a packet arrives, the decision tree is traversed based on information in the packet header to find a leaf node. A small number of matching rules that are stored in the leaf node are linearly traversed to find the highest priority rule that matches the packet.

Choosing cutting fields. HyperCuts chooses the set of fields for which the number of unique elements is greater than the mean of the number of unique elements for all the fields under consideration.

Picking number of cuts. For each of the cutting dimensions i, HyperCuts keeps track of information such as the mean and max number of rules in the child nodes, and the number of empty child nodes. A set of iterations are executed; at each step the current value of number of cuts $nc(i)$ is multiplied by two, until there is no significant change in the the mean or max number of rules in the child nodes, or there is a significant increase in the number of empty nodes. Then the last known best value is used as the chosen number of splits to be made along the dimension under consideration.

Figure 5 shows an example of building a HyperCuts decision tree for the rule set containing 8 rules as shown in table 1. The number of unique elements in fields F_1 through F_5 is 3, 6, 3, 4 and 2 respectively. The mean of 3, 6, 3, 4 and 2 is 3.2. Thus, the root node is split on F_2 and F_4, and 8 children nodes are formed as shown in the figure.

We implement a conflict checking algorithm (Algorithm 1), to determine whether a proposed evolving rule conflicts with the semantics of the original ruleset or the five properties.

Algorithm 1 CheckConflict(Cnode, ExpandedRule, MatchID, ConflictID)

```
if Cnode is a leaf node then
    for each rule $r$ in the rule list of Cnode do
        if $r.ID > min(MatchID, ConflictID)$ then
            return
        end if
        if $r$ overlaps with ExpandedRule then
            if $r.action \neq ExpandedRule.action$ then
                $ConflictID = r.ID$
                return
            end if
            if ExpandedRule lies entirely inside $r$ then
                $MatchID = r.ID$
                return
            end if
        end if
    end for
end if
if Cnode is not a leaf node then
    for each child $c$ of Cnode that overlaps with ExpandedRule do
        CheckConflict($c$, ExpandedRule, MatchID, ConflictID);
    end for
end if
```

Algorithm 1 is a recursive method, starting with an examination of the root node in HyperCuts tree to identify child nodes having rules that overlap with the proposed evolving rule. The algorithm runs recursively on each overlapping child node until a leaf node is identified. Each leaf node in a HyperCuts tree contains a small number of original rules, and the rules are ordered by priority. When a leaf node is reached, each rule in the leaf node is checked until a match or a conflict is found. By match we mean the expanded rule lies in the rule and they have the same action. While if the rule overlaps with the expanded rule but they have different actions, we say there is a conflict. When a match or conflict is found, we terminate from the current node and check the next overlapping leaf node for higher priority match rule or conflict rule. When all the overlapping leaf nodes are checked, we compare the highest priority match rule and conflict rule. If the priority of the match rule is higher than that of the conflict rule, the proposed evolving rule is approved and expanded. Otherwise, the proposed expanded rule is not permitted and the next rule in the evolving list is expanded and checked, or a new evolving rule is constructed if this is the last evolving rule in the evolving rule list.

Figure 5 shows an example of searching an evolving rule $r = (01*, 01*, 10, *, *, action1)$ for conflicts with the ruleset shown in Table 1 and five evolving rule properties. The function is called with Cnode set to root node, and MatchID and ConflictID both initialized to be ruleset size, 8 in this case. r overlaps with children 2 and 3 of the root node. Then 2 and 3 are checked recursively. When the algorithm returns, the match rule ID is 4, which is smaller than the conflict rule ID, the algorithm approves the expansion and terminates.

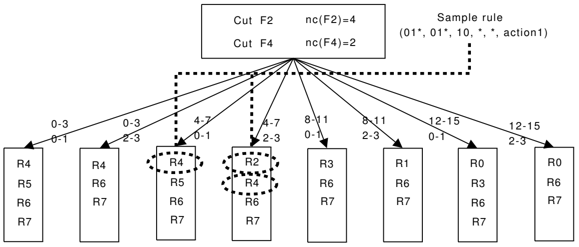

Figure 5: Check an expanded rule for conflicts using HyperCuts decision tree.

4. IMPLEMENTATION ISSUES AND LESSONS

We encountered many challenges in turning Storm design (Figure 2) into a high-performance implementation. In the following, we discuss some major challenges and our solutions.

How many queues shall we use between different threads? Our principle is to make sure that each queue only has one producer and one consumer. Our experience has shown that this can greatly decrease the contention and ease the coding.

How to split incoming packets to multiple queues? Consider a 8-core server handling two 10Gbps ports. If a port is tied to a single core, then each packet is necessarily touched by the core that polls in and splits the traffic, and then a core that actually processes the packet (our rule cache thread), which increases contention and hence increase delay. Fortunately, this problem can be addressed by exploiting a feature now available in most modern NICs and OS: multiple receive and transmit queues [4], which allows multiple queues of incoming traffic. We simulate this behavior by loading incoming packets to multiple queues.

How to avoid the conflict between rule cache lookup and update? Our solution is to have two copies of the rule cache: one for look up during classification and one for updating with more recent popular rules. The rule cache updater thread is responsible for calculating the evolving rules and updating the second rule cache copy with the new popular rules. It will atomically switch the pointers to these two rule cache copies after each cache update. This strategy avoids almost all contention, and improves the throughput by 55%-316%.

How to accommodate the speed difference between packet classification and rule cache updater? How to maintain the quality of the rule cache? In order to achieve high cache hit rate, it is critical to update the rule cache based on recent packet history. This is challenged by the fact that calculating evolving rules and packet classification are carried out independently in different threads, and the former is much slower than the latter. As a result, the queue between the rule cache classifier and the rule cache updater frequently gets full. In our initial design, when the queue is full, the more recent packets are discarded. As a result, the rule cache evolves based on relatively old packets, which severely hurts the quality of the rule cache. Making the queue larger does not solve the problem. Finally, we found a simple solution: instead of having a queue, just have one entry and always have the rule cache updater thread to use the latest packet to calculate the evolving rules. This design is very effective: our experiments show that the rule cache hit rate is generally higher than 95%.

Other implementation issues and data structure adjustment. Our initial implementation used link-lists to represent rules in cache. This turns out to be a bad choice in multi-threaded setting. We changed them to arrays instead to speedup the searching of rules. We also tried to select between blocking-lock, non-blocking spin-lock, and several non-lock data structures. It turned out pthread_spin_lock provides the best and most stable performance.

Platform impact. We evaluated Storm using Intel Xeon X5550 Gainestown, which is based on Nehalem architecture, and Intel Xeon X5440 Harpertown. We found from our experiments that Nehalem speed up the average throughput by 29%-53%. According to Intel, the performance improvements of Nehalem over previous Xeon processors are based mainly on: integrated memory controller supporting two or three memory channels of DDR3 SDRAM or four FB-DIMM channels; a new point-to-point processor interconnect QuickPath, replacing the legacy front side bus; and Hyper-threading. By hyper-threading, a processor is treated by the operating system as two processors instead of one.

We considered several parallel language and libraries, such as CILK, Intel Thread Building Block, shared-memory mapreduce, StreamIt. At the end, we still decide to stick to C language and POSIX pthread-library, because of their simplicity and portability.

We expect that future OS, hardware and parallel language progress can make future multi-threaded software router development easier. Advanced OS threading support, like Mac OS' recent Grand Central Dispatch, could make our dynamic thread assignment easier. Our design assumes homogeneous multi-core systems. We expect that future heterogeneous multi-core system would bring new opportunities.

5. EXPERIMENTAL RESULTS

In this section, we implement Storm and evaluate its performance. We use five real rulesets and real traffic traces obtained from a tier-1 ISP backbone network. The five rulesets R1, R2, R3, R4 and R5 contain 460, 711, 852, 1036 and

Table 4: A summary of the seven rulesets.

Rule set	Type	Size
R1	real	460
R2	real	711
R3	real	852
R4	real	1036
R5	real	1802
R6	synthetic	4415
R7	synthetic	9603

1802 rules, respectively. Since we do not have larger real rulesets, we also evaluate Storm using two synthetic large rulesets R6 and R7, which contains 4415 and 9603 rules, respectively. These synthetic rulesets are generated using the ClassBench ruleset generator [17]. All the rules in the rulesets are 5 dimensional tuples composed of source and destination IP addresses, source and destination port numbers and protocol type. Besides, each rule is associated with an action. The action of a rule is either permit or deny. All the seven rulesets are summarized in table 4. The five traces (T1, T2, T3, T4, and T5) each contain about 7 million packets. To generate different incoming traffic rates using these traces, we decrease the time separation between consecutive packets. We experimented with all rulesets and all traffic traces, and in different plots and tables below, present results from all of them. However, when unstated, the trace used for the corresponding experiment was a a single representative trace, T1.

We evaluated Storm using Intel Xeon X5550 Gainestown, which is a 8-core Nehalem desktop platform. The packet size is assumed to be 128 bytes (1024 bits). The total buffer size in the shared queues between rule cache threads is set to 1000 packets.

To evaluate the throughput of Storm, we use HyperCuts-1 and HyperCuts-8 as baselines for comparison. HyperCuts-1 utilizes a single core and executes a single version of HyperCuts. It does not exploit any parallelism opportunity. HyperCuts-8 leverages data parallelism, where each of the 8 cores executes an identical version of the HyperCuts software classifier.

The throughput we measure is the *maximum throughput achievable* by each scheme without packet loss, beyond which packet drops will occur. For HyperCuts and Storm, the packet loss occurs when the incoming packet queue fills up. For Storm, drops could also happen when the software classifier is overloaded. In our experiments, we demonstrate that the packet drops for each approach occur at a different incoming rate, with Storm significantly outperforming HyperCuts-8.

5.1 Maximum Achievable Throughput

We define the maximum achievable throughput of a particular scheme as the maximum incoming traffic rate that leads to no packet losses for that particular scheme. Any higher incoming rate will lead to packet losses due to the inability of the software to keep classify packets fast enough.

Figure 6 shows the throughput achieved by Storm using static thread partitions 2-4-1, 4-2-1, 3-3-1 and dynamic thread assignment, and HyperCuts, where the x-axis represents the seven rulesets R1 through R7, and y-axis is the packet classification throughput achieved by the corresponding ruleset.

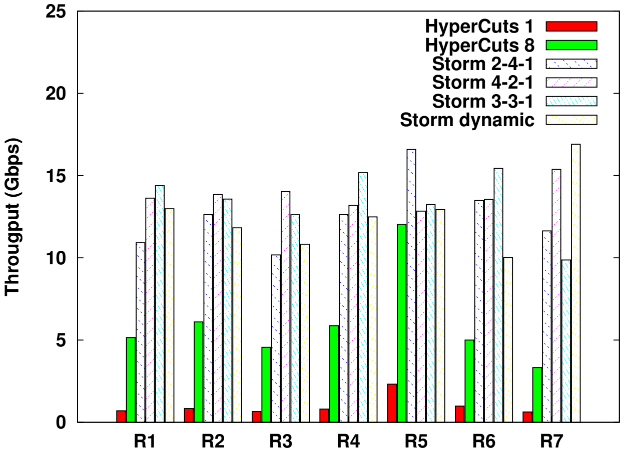

Figure 6: Router throughput and threads partitioning on rulesets R1 through R7.

The throughput gain of HyperCuts-8 over HyperCuts 1 ranges from 5 to 7, roughly proportional to the number of cores (8). This gain comes from data parallelism.

Storm combines the three forms of parallelism, namely task parallelism, data parallelism, and pipeline parallelism. As we can see, the maximum achievable throughput of Storm is generally 2-3 times that of HyperCuts-8. This figure confirms that 2-4-1, 4-2-1 and 3-3-1 are good thread partitions, among which thread partition 4-2-1 gives the best performance overall. The throughput of thread partition 4-2-1 for every ruleset is higher than 12Gbps. For the rule set consisting of more than 9000 rules (R7), Storm achieves a sustained throughput of more than 15Gbps with both static 4-2-1 and dynamic thread assignment.

In the figure, Storm dynamic represents the results from dynamic thread assignment with the lower and upper thresholds set to 0.2 and 0.8 respectively. We can see the throughput of dynamic thread assignment is close to 4-2-1 thread partition. This is because dynamic thread assignment usually switch between 2-4-1 and 4-2-1 at the beginning of each run, and when the system enters a steady state where the hit ratio remains high, it stays in 4-2-1. The actual throughput of dynamic thread assignment is generally a little bit lower than 4-2-1 because of its checking (check the available buffer size in shared queues) and switching overhead.

5.1.1 Understanding the gains of Storm

In this section, we attempt to intuitively understand the gains and performance tradeoffs of Storm, when compared to the alternate and simpler data parallel approach (using HyperCuts-8, which executes an identical instance of HyperCuts in each of the 8 cores of the system).

In HyperCuts-8, the classification time for each packet is about 2000 ns. Given the availability of 8 cores, the average packet classification latency of the system is about 200-300 ns. Hence, if packets arrive at a rate faster than this frequency, then packet losses will happen at the input port of the system. In our experiments, we examine the fastest rate at which HyperCuts can accept incoming packets such that packet losses do not occur.

Storm is a system that uses the rule cache system with the central goal being to identify a small set of rules that are likely to match the largest number of incoming packets. Given a small set of such popular rules, the average time required for a packet to be matched against this ruleset is around 200 ns. Clearly, the rule cache can accept and process packets at a faster rate, given that we can have multiple rule cache threads. For example, 4 rule cache threads will bring the average rule cache lookup delay to 50 (200/4) ns. However, if the packets fail to match against rules in the cache, then they get buffered for classification using the software classifier. So long as the rule cache can classify a significant fraction of incoming packets correctly, the load on the full software classifier would be relatively low leading to no additional packet losses. However, if the success rate of the rule cache falls, then there would be an overload experienced by the full software classifier leading to packet losses. In our experiments, we use an incoming traffic rate that would not cause such packet losses for Storm. As our experiments indicate, Storm can handle an incoming rate which is about 2-3 times that of a data parallel version of HyperCuts (HyperCuts-8).

5.2 Scalability of Storm

We next study the scalability of Storm by varying the number of cores available in the system. Since our system has 8 cores, we used the pthread_setaffinity_np function to bind all threads to a fixed number of cores. We varied the number of cores between 2 and 8. Figure 7 shows the throughput on the seven rulesets using thread partition 1-1-1 with 2 core, 2-2-1 with 4 cores, and 4-4-1 with 8 cores. Figure 7 shows near linear scalability. We expect that this linear growth in performance can continue for a much larger number of cores, using our optimization design approach that selects an efficient partitioning of tasks to threads and cores.

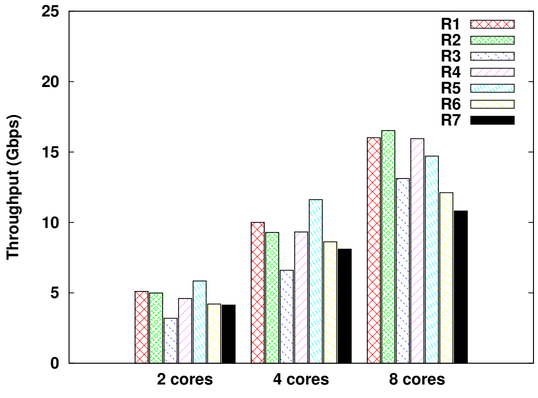

Figure 7: Storm's scalability with number of cores.

5.3 Cache hit ratio

We next study the performance of the rule cache, in particular, the fraction of packets that are classified by the rule cache, which is called the cache hit ratio. To understand how cache hit ratio changes, we show the results from a randomly chosen ruleset (R3). We run the experiment for 7.5 million packets, which takes about 1 second. Figure 8 shows how the hit ratios ramps up. It can be seen that after a short warmup period (about 30 milliseconds) the system arrives at a steady state where the cache hit ratio exceeds 90%. The cache hit ratio constantly improves and stabilizes at 98% soon after.

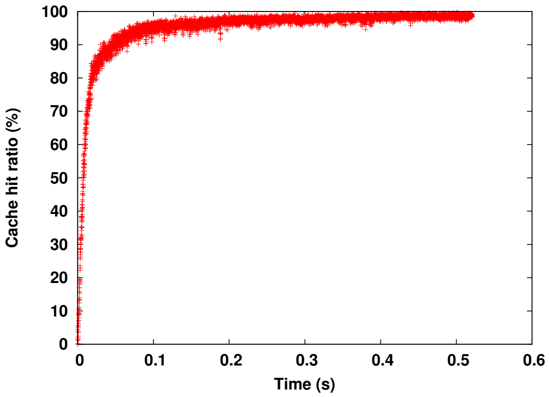

Figure 8: Storm's rule cache hit ratio ramps up quickly (example uses ruleset R3).

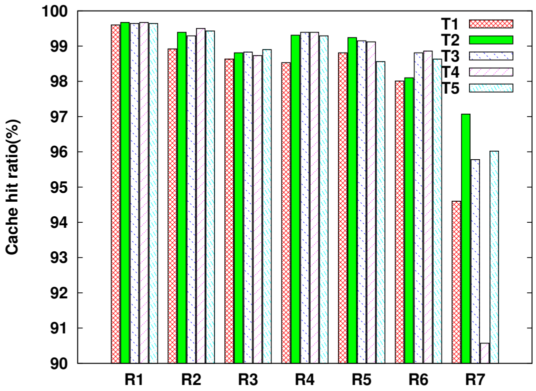

Figure 9: Cumulative rule cache hit ratios of rulesets R1 through R7 with traces T1 through T5 (Y-axis starts at 90%).

To further understand the rule cache hit ratio of each ruleset, we perform an experiment to evaluate the cumulative hit ratios of rulesets R1 through R7 on five different traces each of size about 5 million packets.

Figure 9 shows the cumulative hit ratios of rulesets R1 through R7 on trace files T1, T2, T3, T4 and T5. The rule cache hit ratio shown is the average cache hit ratio in steady state. The thread partition scheme we use is static 4-2-1. The rule cache hit ratio is in excess of 98% for most of the trace and ruleset combinations.

There is also a trend of decreasing cache hit ratio with increase in ruleset sizes. This is expected as the computation time to identify popular rule sets increases with increase in rule set size. As popular rule computation time increases, the cache update frequency goes down, leading to the cached rules being less current. Overall, this leads to a lower cache hit ratio, a factor that contributes to lower achievable throughputs for larger rule sets.

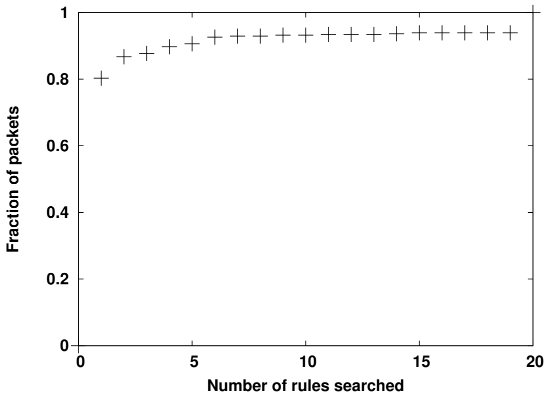

Figure 10: Cumulative distribution of number of entries searched in Storm's rule cache for 1000 randomly sampled packets (using ruleset R2).

5.4 Micro benchmarks

5.4.1 Rule Cache

In this section, we present how we make some design choices on rule cache. Specifically, we answer two questions: how to lookup rules in rule cache (why we use linear search) and what the rule cache size should be.

Rule cache lookup.

We use linear search to search the rules in the rule cache to find a match or a cache miss. As we discuss before, the rules in the rule cache are ordered by its popularity (weight). We run a trace file of 5 million packets on ruleset R2. Figure 10 shows the number of rules searched in the rule cache for 1000 randomly sampled incoming packets. As shown in the figure, for cache hit packets, most searches only take a small number or rules, like 1 or 2. If cache miss happens, all the rules in the cache are searched. Since our cache size is set to 20, we can see in the figure that for a small portion (around 5%) of incoming packets, all the 20 rules are searched. Our experiments show the cache hit ratio is generally above 95%, so this does not happen often. Since the rules are ordered by weights, we found that linear search of the cached rules is sufficient to ensure that only a small number of rules are searched.

Rule cache size.

In our experiments, we choose cache size to be 20. Our experiments show that for a sliding window size of 1024, generally 20-30 evolving rules are formed with incoming packet rate 10Gbps. In our experiments, we only switch the first 20 rules to the rule cache.

Table 5 shows the average classification delay per packet and cache hit ratio with different number of cache sizes on ruleset R3 using thread partition 4-2-1. As the cache sizes getting larger, the average classification delay first decreases with the increase of cache hit ratio, since higher cache hit ratio means more packets take the faster path so that the average delay is smaller. Then as the cache size gets larger (25 and 30), the average delay starts to increase. As we can see, when the cache size is 20, the hit ratio is 96.55% (remaining packets are classified by the full software classifier). Larger size of cache beyond 20 does not give much performance improvement, it rather slows down the classification because more rules need to be checked. Hence, we picked a cache size of 20 for Storm.

Table 5: Performance with different cache sizes on ruleset R3

Cache size	Delay(ns/p)	Hit ratio(%)
10	95.21	93.92
15	93.27	95.31
20	82.01	96.55
25	83.28	95.92
30	96.52	96.52

Table 6: Warmup time (in ms) of the seven rulesets using thread partition 2-4-1 and 4-2-1 for Storm.

Rule set	Storm 2-4-1	Storm 4-2-1
R1	3 ms	5 ms
R2	2 ms	4 ms
R3	10 ms	13 ms
R4	6 ms	12 ms
R5	1 ms	1 ms
R6	8 ms	26 ms
R7	36 ms	79 ms

5.4.2 Warmup behavior

When a Storm system boots up, the rule cache is empty. The system needs some time to construct an initial set of popular rules in rule cache and improve cache hit ratio (as shown in Figure 8). During this time if the incoming traffic rate is greater than the rate at which the software classifier can process packets (say, less than 4 Gbps for R7), the shared queues between rule cache threads and full software classifier threads might encounter some packet loss. After a short period of time, the rule cache hit ratio ramps up, and the system can operate at the much higher incoming traffic rate (around 17 Gbps for R7) without any packet loss. Then the system continues to operate in a steady state after this point with the cache hit ratio staying high. We define this short period of time during which packet losses can occur at the higher speeds, the warmup period.

Table 6 shows that the warmup period for each of the seven rulesets on one of the tracefiles (T1) using thread partitions 2-4-1 and 4-2-1 is no more than 79 milliseconds. The packet classification rate during this warmup period (just after bootup) is limited by the full software classifier's classification rate, and increases to Storm's full classification rate after the short warmup period ends.

6. RELATED WORK

The simplest packet classification algorithm is a linear search through all the rules in a rule set. However, for a large number of rules, this implies a large search time.

The general problem of packet classification on multiple fields was first studied in [16]. In general, there are two main threads of research on packet classification: algorithmic and hardware architectural.

A lot of intelligent algorithmic solutions are proposed to code with packet classification [3, 13, 2, 18, 7, 16, 15], and many original ideas have been investigated for improvement.

Prior work has also proposed software-based classification algorithms, e.g., HiCuts [7], HyperCuts [13], and HyperSplit [12]. Both HiCuts and HyperCuts are decision tree based schemes. Each node in the HyperCuts decision tree represents a multi-dimensional hypercube, while in HiCuts, each node represents a hyperplane. Another algorithm, called HyperSplit [12], is a simple variation of the basic HyperCuts algorithm.

One common feature of algorithms employing the decision tree approach is memory access dependency, that is, the decision tree searches are inherently serial; a matching rule is found by traversing the tree from root to leaf. The serial nature of the decision tree approach precludes fully parallel implementations. The core issue of algorithmic approaches centers on the tradeoff between memory usage and speed.

Our work is complementary to all algorithmic approaches, as we propose a rule cache based system that can use any of these algorithmic approaches as one software component (the full software classifier). In this paper, we utilize HyperCuts as one such algorithmic approach whose throughput was improved by a factor of two to three in our experiments.

Wire speed packet classification has also motivated the development of hardware-based solutions. Ternary Content Addressable Memory (TCAM), the most widely-used packet classification technique, has been studied and employed in industry [6, 5, 1]. TCAMs perform a parallel single-clock-cycle search for an incoming packet in all stored rules. TCAMs do suffer from four primary deficiencies: (1) high cost per bit relative to other memory technologies, (2) storage inefficiency, (3) high power consumption, and (4) limited scalability to long inputs. To compress space in TCAM, [8] proposes a new approaches to range reencoding by taking into account classifier semantics. Some new architectural research combines intelligent algorithms and novel architectures to eliminate many of the unfavorable characteristics of current TCAMs [14]. However, all such solutions still depend on specialized hardware, which is avoided by Storm. The key difference between Storm and such prior approaches is its software-only nature that can be mapped into any multi-core platform available today.

7. CONCLUSION

We present a detailed design, implementation, and measurement study of a system, called Storm, for packet classification in multi-core platforms. Our system does not propose a new packet classification algorithm. Its design is based on the common notion that a small subset of rules are likely to be more popular than others. Hence, if we can identify these popular rules and match all incoming traffic against them, we can get significant speed-up in classification performance. When packets fail to be matched against the cached popular rules, full software classification is required, for which any existing packet classification algorithm can be used. Storm uses a combination of task, data, and pipeline parallelism to significantly outperform a naive data parallel approach (using multiple instances of a single packet classification algorithm, one on each core). Further, our intuition and initial experiments indicate that the proposed approach can scale with increase in number of cores in the system. We believe that such an approach can find direct use in emerging router systems that are based on desktop-based components, e.g., RouteBricks.

8. ACKNOWLEDGMENTS

The authors would like to thank our shepherd Neil Spring and other anonymous reviewers for their comments and suggestions which helped bring this paper to its final form. Yadi Ma and Suman Banerjee were partially supported by the US NSF through awards CNS-0916955, CNS-0855201, CNS-0751127, CNS-0627589, CNS-0627102, and CNS-0747177.

9. REFERENCES

[1] A.J.McAulay and P. Francis. Fast routing table lookup using cams. In *IEEE INFOCOM*, 1993.

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

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

[4] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, and K. Fall. Routebricks: Exploiting parallelism to scale software routers. In *SOSP*, 2009.

[5] 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*, pages 253–264, June 2007.

[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] P. Gupta and N. Mckeown. Packet classification using hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, January 2000.

[8] C. R. Meiners, A. X. Liu, and E. Torng. Topological transformation approaches to optimizing tcam-based packet classification systems. In *ACM SIGMETRICS*, pages 73–84, 2009.

[9] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *SOSP*, pages 217–231, 1999.

[10] C. Networks. Octeon network service processors. http://www.cavium.com/octeon_software_develop_kit.html.

[11] M. H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. *Journal of Algorithms*, 21:629–656, 1994.

[12] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *IEEE INFOCOM*, 2009.

[13] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, pages 213–224, August 2003.

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

[15] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM SIGCOMM*, pages 135–146, 1999.

[16] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. Fast and scalable layer four switching. In *ACM SIGCOMM*, 1998.

[17] D. Taylor and J. Turner. Classbench: A packet classification benchmark. http://www.arl.wustl.edu/∼ det3/ClassBench/index.htm.

[18] T. Y. Woo. A modular approach to packet classification: algorithms and results. In *IEEE INFOCOM*, pages 1213–1222, 2000.