Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers

Yadi Ma  Suman Banerjee
Shan Lu  Cristian Estan*

Presenter: Chang Chen
Mar 19, 2014
Question

- If we were to implement classification using state-of-art desktops, given that it may find applications in router design, 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?
Outline

- Introduction
- Overview of Storm
- Storm Design
  - Thread Assignment
  - Rule Cache Updater
  - Micro Benchmarks
- Experimental Results
**Introduction**

- **Storm**: a software-based solution to the multi-dimensional packet classification problem.

- A new software system which
  - takes existing classification algorithms;
  - utilizes a common idea of caching;
  - partitions critical tasks into multiple threads that effectively leverage desktop platforms to meet various computation and memory access needs.
Prior Work


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

- A parallelized software system using multi-core desktop platforms.
- Uses a combination of task, data and pipeline parallelism.

Figure 2: Architecture of Storm.

NSLab, RIIT, Tsinghua Univ
Overview of Storm

- Task Parallelism
  - Rule cache threads
    - Match incoming packets against cached rules.
  - Full software classifier threads
    - Use the entire rule set to carry out classification (with HyperCuts algorithm).
  - Rule cache updater threads
    - Continuously sample incoming traffic, identify the evolving set of popular rules and update the rule caches.
Overview of Storm

- **Data Parallelism**
  - By creating multiple instances of each thread and allowing them to operate on different packets.

- **Pipeline Parallelism**
  - Multiple tasks executed in a specific pre-defined order for each incoming packet.
  - Following a producer-consumer pattern between the two classification stages.
Overview of Storm

- Main contributions of Storm:
  - A practical, multi-threaded, software-only packet classification system.
  - Design of dynamic balancing of computation resources to tasks.
Outline

- Introduction
- Overview of Storm

Storm Design
- Thread Assignment
- Rule Cache Updater
- Micro Benchmarks

- Experimental Results
Thread Assignment

- Three different tasks threads assignment
  - Rule cache lookup
  - Full software classification
  - Rule cache updating

- Theoretical model

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

- $T$: throughput
- $d_1$: average delay of rule cache lookup
- $d_2$: average delay of full software classification
- $t_1$: number of rule cache threads
- $t_2$: number of full software classifier threads
- $r$: average rule cache hit ratio
- $C$: const (queuing delay, synchronization overhead, etc.)
Thread Assignment

- **Theoretical model**

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

- $d_1$: average delay of rule cache lookup
- $d_2$: average delay of full software classification
- $t_1$: number of rule cache threads
- $t_2$: number of full software classifier threads
- $N$: $t_1 + t_2$
- $r$: average rule cache hit ratio

- Use only one thread to carry out the sampling and rule cache updating task.
Thread Assignment

- Static thread assignment (on 8-core machines)

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

- Control total number of thread around 8
- $d_1$: about 200 ns (rule cache lookup)
- $d_2$: about 2000 ns (full classification using HyperCuts)
- $r$: around 0.95
- minimize synchronization overhead

- Potential thread partitions: 3-3-1, 4-2-1, 2-4-1
Thread Assignment

- Dynamic thread assignment (on 8-core machines)
  - Initially create 4 rule cache threads and 4 full software classification threads.
  - Define the total buffer size in between to be $B$; define two thresholds, $r_1$ and $r_2$, where $0 < r_1 < r_2 < 1$.
  - Depending on the available buffer size, switch between 4-2-1 (if $< r_1 \times B$) and 2-4-1 ($> r_2 \times B$).
Rule Cache Updater

- A rule cache updater thread is responsible for periodically sampling the incoming packets and updating the rules in the rule cache.

- Each entry in the rule cache stores an evolving rule.
Rule Cache Updater

Table 3: A simple ruleset on 2 fields.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Field\textsubscript{1}</th>
<th>Field\textsubscript{2}</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>1-9</td>
<td>4-10</td>
<td>action\textsubscript{0}</td>
</tr>
<tr>
<td>R1</td>
<td>7-14</td>
<td>3-8</td>
<td>action\textsubscript{1}</td>
</tr>
<tr>
<td>R2</td>
<td>3-11</td>
<td>1-6</td>
<td>action\textsubscript{2}</td>
</tr>
<tr>
<td>R3</td>
<td>0-15</td>
<td>0-11</td>
<td>action\textsubscript{3}</td>
</tr>
</tbody>
</table>

Figure 4: Constructing evolving rules, rule priority: 
R0 > R1 > R2 > R3, \( action_0 \neq action_1 \neq action_2 \neq action_3 \).
Rule Cache Updater

- Sliding window
  - A FIFO queue that stores recently sampled packets.

- Evolving rule list
  - A list of proposed evolving rules waiting to be checked for conflicts and to be transferred into rule cache;
  - Each evolving rule includes a weight field;
Rule Cache Updater

- Five properties of evolving rules:
  - Represents a d-dimensional hypercube.
  - Be associated with a single action consistent with the original ruleset.
  - Each sample packet in the sliding window is assigned to one evolving rule that matches it.
  - Evolving rules either have the same action or are non-overlapping.
  - Lies entirely inside one of the rules in the original ruleset.
Rule Cache Updater

- Check conflicts
  - Start with root
  - Runs recursively on overlapping child node until a leaf node is identified
  - Check each rule in the leaf node is a match or a conflict

```
Algorithm 1 CheckConflict(Cnode, ExpandedRule, MatchID, ConflictID)
1: if Cnode is a leaf node then
2:   for each rule r in the rule list of Cnode do
3:     if r.ID > min(MatchID, ConflictID) then
4:       return
5:   end if
6:   if r overlaps with ExpandedRule then
7:     if r.action \neq ExpandedRule.action then
8:       ConflictID = r.ID
9:     return
10:   end if
11:   if ExpandedRule lies entirely inside r then
12:     MatchID = r.ID
13:   return
14: end if
15: end if
16: end for
17: end if
18: if Cnode is not a leaf node then
19:   for each child c of Cnode that overlaps with ExpandedRule do
20:     CheckConflict(c, ExpandedRule, MatchID, ConflictID);
21: end for
22: end if
```
Rule Cache Updater

- Check conflicts

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>F₁</th>
<th>F₂</th>
<th>F₃</th>
<th>F₄</th>
<th>F₅</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₀</td>
<td>000*</td>
<td>111*</td>
<td>10</td>
<td>*</td>
<td>UDP</td>
<td>action₀</td>
</tr>
<tr>
<td>R₁</td>
<td>000*</td>
<td>10*</td>
<td>01</td>
<td>10</td>
<td>TCP</td>
<td>action₁</td>
</tr>
<tr>
<td>R₂</td>
<td>000*</td>
<td>01*</td>
<td>*</td>
<td>11</td>
<td>TCP</td>
<td>action₀</td>
</tr>
<tr>
<td>R₃</td>
<td>0*</td>
<td>1*</td>
<td>*</td>
<td>01</td>
<td>UDP</td>
<td>action₂</td>
</tr>
<tr>
<td>R₄</td>
<td>0*</td>
<td>0*</td>
<td>10</td>
<td>*</td>
<td>UDP</td>
<td>action₁</td>
</tr>
<tr>
<td>R₅</td>
<td>000*</td>
<td>0*</td>
<td>*</td>
<td>01</td>
<td>UDP</td>
<td>action₁</td>
</tr>
<tr>
<td>R₆</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>UDP</td>
<td>action₃</td>
</tr>
<tr>
<td>R₇</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>TCP</td>
<td>action₄</td>
</tr>
</tbody>
</table>

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

- How to lookup rules in rule cache?
  - Use linear search to search rules in cache to find a match or cache miss.

- What the rule cache size should be?

<table>
<thead>
<tr>
<th>Cache size</th>
<th>Delay(ns/p)</th>
<th>Hit ratio(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>95.21</td>
<td>93.92</td>
</tr>
<tr>
<td>15</td>
<td>93.27</td>
<td>95.31</td>
</tr>
<tr>
<td><strong>20</strong></td>
<td><strong>82.01</strong></td>
<td><strong>96.55</strong></td>
</tr>
<tr>
<td>25</td>
<td>83.28</td>
<td>95.92</td>
</tr>
<tr>
<td>30</td>
<td>96.52</td>
<td>96.52</td>
</tr>
</tbody>
</table>
Experimental Results

- Rulesets

<table>
<thead>
<tr>
<th>Rule set</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>real</td>
<td>460</td>
</tr>
<tr>
<td>R2</td>
<td>real</td>
<td>711</td>
</tr>
<tr>
<td>R3</td>
<td>real</td>
<td>852</td>
</tr>
<tr>
<td>R4</td>
<td>real</td>
<td>1036</td>
</tr>
<tr>
<td>R5</td>
<td>real</td>
<td>1802</td>
</tr>
<tr>
<td>R6</td>
<td>synthetic</td>
<td>4415</td>
</tr>
<tr>
<td>R7</td>
<td>synthetic</td>
<td>9603</td>
</tr>
</tbody>
</table>

- 5 tuples; action is either permit or deny.

- Traces
  - Five traces each contain about 7M packets (packet size: 128 bytes).
  - Different incoming traffic rates.
Experimental Results

- Maximum achievable throughput
Experimental Results

- Scalability with number of cores
Experimental Results

- Cache hit ratio

![Graph showing cache hit ratio over time](image1)

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

![Bar chart showing cumulative cache hit ratios](image2)

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

- Micro benchmarks

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

- Micro benchmarks

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

<table>
<thead>
<tr>
<th>Rule set</th>
<th>Storm 2-4-1</th>
<th>Storm 4-2-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>3 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td>R2</td>
<td>2 ms</td>
<td>4 ms</td>
</tr>
<tr>
<td>R3</td>
<td>10 ms</td>
<td>13 ms</td>
</tr>
<tr>
<td>R4</td>
<td>6 ms</td>
<td>12 ms</td>
</tr>
<tr>
<td>R5</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>R6</td>
<td>8 ms</td>
<td>26 ms</td>
</tr>
<tr>
<td>R7</td>
<td>36 ms</td>
<td>79 ms</td>
</tr>
</tbody>
</table>
Conclusions

- **Storm**: a software-based solution to the multi-dimensional packet classification problem.

- A new software system which
  - takes existing classification algorithms;
  - utilizes a common idea of caching;
  - partitions critical tasks into multiple threads that effectively leverage desktop platforms to meet various computation and memory access needs.
Thank You!