Ruler: High-Speed Packet Matching and Rewriting on Network Processors

Tomáš Hrubý   Kees van Reeuwijk   Herbert Bos

Vrije Universiteit, Amsterdam
World45 Ltd.

ANCS 2007
Why packet pattern matching?

Protocol header inspection
- IP forwarding
- Content based routing and load-balancing
- Bandwidth throttling, etc.

Deep packet inspection
- Required by intrusion detection and prevention systems (IDPS)
- Inspecting IP and TCP layer headers is not sufficient
- The payload contains malicious data
Why packet rewriting?

Anonymization
- We need to store traffic traces
- Network users are afraid of misuse of their data and identity
- ISPs want to protect their customers

Data reduction
- The amount of data in the Internet is huge
- Applications need only data of their interest
- The data reduction must be online!
The **Ruler** goals

- a system for packet classification based on regular expressions
- a system for packet rewriting
- a system deployable on the network edge
- a system easily portable to other architectures

*Ruler* provides all of these!
The **Ruler** program

```plaintext
filter udp
  header: (byte#12 0x800~2 byte#9 17 byte#2)
  address: (192 168 1 byte)
  tail: *
    =>
  header 0#4 tail;
```

- A program (**filter**) is made up of a set of **rules**
- Each rule has the form `pattern => action;`
- Each rule has an **action part**
  - accept `<number>`
  - reject
  - rewrite pattern (e.g., `header 0#4 tail`)
- Labels (e.g., `header`, `address`, `tail`) refer to sub-patterns
The **Ruler** templates

- Often used patterns can be defined as templates

  ```
  pattern Ethernet :
      (dst:byte#6 src:byte#6 proto:byte#2)
  ```

- Templates can use other templates for more specific patterns

  ```
  pattern Ethernet_IPv4 :
      Ethernet with [proto=0x0800~2]
  ```

  ```
  filter ether
      e:Ethernet_IPv4 t:* => e with [src=0#6] t;
  ```

- **Ruler** program can include files with templates

  ```
  include "layouts.rli"
  ```
Parallel pattern matching

- Deterministic Finite Automaton for matching multiple patterns
  state types inspection, memory inspection, jump, tag, accept

- **Ruler** remembers position of sub-patterns - Tagged DFA (**TDFA**)

```plaintext
filter byte42
  * 42 b:(byte 42) * => b;
```

Position of label `b` is determined only at runtime
DFA contains *tag states* to record the position in a tag-table
Why is it so difficult to use NPUs?

Parallelism

It is difficult to think *parallel* and NPUs employ various parallelism techniques: multiple execution units or threads, pipelines.

Poor code portability

- Various C dialects
- Too many features to exploit

IXP2xxx

- Hierarchy of *asynchronous* memories (Scratch, SRAM, DRAM)
- Many cores with hardware multi-threading (micro-engines - ME)
- Special instructions, atomic memory operations, queues, etc.

__declspec(shared gp_reg)
__declspec(sram)
__declspec(shared scratch)
__declspec(dram_read_reg)
Why use NPUs?

- Running on bare-metal with minimal overhead
- Embedded in routers, switches and smart NICs
- Worst case guarantees
  - number of available cycles
  - exact memory latency
  - no speculative execution or caching
- Hardware acceleration
  - PHY integrated into the chip
  - hashing units
  - crypto units
  - CAM
  - fast queues
Ruler on the IXP2xxx

- Dedicated RX and TX engines
- All other engines execute up to 8 Ruler threads
- Only one thread per ME is polling on the RX queue to reduce memory load and execution resources
- Each thread processes independently a single packet
- Only RX and TX queues synchronize the threads
Inspection states

Inspection states are the most often executed ⇒ need optimization

Reading the next byte from the input

- No DRAM latency due to prefetching
- Faster reading from positions known in compile time (headers)
- Skipping bytes of no interest

Multi-way branch

- Select the transition to the next state
- Has the most impact on the performance
- The default branch is the one taken most frequently
- We have two implementations:
  - Naive
  - Binary tree with default branch promotion
Binary tree switch statements

**Binary tree**

- Test multiple values by checking single bits, one at a time
  - '0' ... '9' < 64
  - 'a' ... 'z' 'A' ... 'Z' < 128
- We select the bit that puts most of the default values in one subtree
- Testing a bit takes 1 cycle
- The "jump" branch takes 3 extra cycles
- We make fall-through branch the subtree with more defaults
- It is a heuristic
Naive vs. binary tree switch statements

**Naive**

```
alu[--, act_char, -, 47]
blt[STATE_20#]
alu[--, act_char, -, 120]
bge[STATE_20#]

br=byte[act_char, 0, 47, STATE_24#]
br=byte[act_char, 0, 110, STATE_26#]
br=byte[act_char, 0, 112, STATE_23#]
br=byte[act_char, 0, 115, STATE_33#]
br=byte[act_char, 0, 117, STATE_22#]
br=byte[act_char, 0, 119, STATE_21#]

br[STATE_20#]
```

**Binary tree**

```
alu[-, act_char, -, 47]
blt[STATE_20#]
alu[--, act_char, -, 120]

br_bclr[act_char, 5, STATE_20#]
br_bclr[act_char, 0, BIT_BIN_33_31#]
br_bset[act_char, 2, BIT_BIN_33_32#]
br[STATE_20#]

BIT_BIN_33_32#:
br_bclr[act_char, 1, BIT_BIN_33_33#]
br_bset[act_char, 3, BIT_BIN_33_34#]
br_bset[act_char, 4, BIT_BIN_33_35#]
br[STATE_20#]

BIT_BIN_33_35#:
...
```

- Default branch is taken after 2 cycles in contrast to 10 if bit 5 is not set
- Measured up to 10% overall speedup
Naive vs. binary tree switch statements

**Naive**

\[
\begin{align*}
\text{alu} & \text{[--}, \text{act}_\text{char}, -, 47]\text{]} \\
\text{blt} & \text{[STATE}_20\text{]} \\
\text{alu} & \text{[--}, \text{act}_\text{char}, -, 120]\text{]} \\
\text{bge} & \text{[STATE}_20\text{]} \\
\text{br} & \text{byte[act}_\text{char}, 0, 47, \text{STATE}_24\text{]} \\
\text{br} & \text{byte[act}_\text{char}, 0, 110, \text{STATE}_26\text{]} \\
\text{br} & \text{byte[act}_\text{char}, 0, 112, \text{STATE}_23\text{]} \\
\text{br} & \text{byte[act}_\text{char}, 0, 115, \text{STATE}_33\text{]} \\
\text{br} & \text{byte[act}_\text{char}, 0, 117, \text{STATE}_22\text{]} \\
\text{br} & \text{byte[act}_\text{char}, 0, 119, \text{STATE}_21\text{]} \\
\text{br} & \text{[STATE}_20\text{]} \\
\end{align*}
\]

**Binary tree**

\[
\begin{align*}
\text{alu} & \text{[--}, \text{act}_\text{char}, -, 47]\text{]} \\
\text{blt} & \text{[STATE}_20\text{]} \\
\text{alu} & \text{[--}, \text{act}_\text{char}, -, 120]\text{]} \\
\text{br}\_\text{bclr}[\text{act}_\text{char}, 5, \text{STATE}_20\text{]} \\
\text{br}\_\text{bclr}[\text{act}_\text{char}, 0, \text{BIT}_\text{BIN}_33\text{31}\text{]} \\
\text{br}\_\text{bset}[\text{act}_\text{char}, 2, \text{BIT}_\text{BIN}_33\text{32}\text{]} \\
\text{br} & \text{[STATE}_20\text{]} \\
\text{BIT}_\text{BIN}_33\text{32}\text{:} \\
\text{br}\_\text{bclr}[\text{act}_\text{char}, 1, \text{BIT}_\text{BIN}_33\text{33}\text{]} \\
\text{br}\_\text{bset}[\text{act}_\text{char}, 3, \text{BIT}_\text{BIN}_33\text{34}\text{]} \\
\text{br}\_\text{bset}[\text{act}_\text{char}, 4, \text{BIT}_\text{BIN}_33\text{35}\text{]} \\
\text{br} & \text{[STATE}_20\text{]} \\
\text{BIT}_\text{BIN}_33\text{35}\text{:} \\
\vdots
\end{align*}
\]

- Default branch is taken after 2 cycles in contrast to 10 if bit 5 is not set
- Measured up to 10% overall speedup
Executed vs. interpreted states

Instruction store is limited $\Rightarrow$ executed and interpreted states

- Number of states may explode exponentially
- Experiments show that hot states are few and they are close to the initial state
- We move distant states to off-chip memory
- We also move states that are too expensive
- The code must include stubs to start the interpreter that reads transitions from a table in SRAM
- The iteration stops once the code fits in the instruction store

Simplified DFA, loop edges are missing
Executed vs. interpreted states

Instruction store is limited $\Rightarrow$ executed and interpreted states

- Number of states may explode exponentially
- Experiments show that hot states are few and they are close to the initial state
- We move distant states to off-chip memory
- We also move states that are too expensive
- The code must include stubs to start the interpreter that reads transitions from a table in SRAM
- The iteration stops once the code fits in the instruction store
The implementation

**Executed vs. interpreted states**

Instruction store is limited ⇒ executed and interpreted states

- Number of states may explode exponentially
- Experiments show that hot states are few and they are close to the initial state

- We move distant states to off-chip memory
- We also move states that are too expensive
- The code must include stubs to start the interpreter that reads transitions from a table in SRAM
- The iteration stops once the code fits in the instruction store

*Simplified DFA, loop edges are missing*
Executed vs. interpreted states

Instruction store is limited $\Rightarrow$ executed and interpreted states

- Number of states may explode exponentially
- Experiments show that hot states are few and they are close to the initial state
- We move distant states to off-chip memory
- We also move states that are too expensive
- The code must include stubs to start the interpreter that reads transitions from a table in SRAM
- The iteration stops once the code fits in the instruction store

Simplified DFA, loop edges are missing
Executed vs. interpreted states

Instruction store is limited ⇒ executed and interpreted states

- Number of states may explode exponentially
- Experiments show that hot states are few and they are close to the initial state
- We move distant states to off-chip memory
- We also move states that are too expensive
- The code must include stubs to start the interpreter that reads transitions from a table in SRAM
- The iteration stops once the code fits in the instruction store
Limits of the IXP2400

Clock cycles

- **29** to **36** cycles per byte (1518 to 64 bytes ethernet frames)
- Interpreted inspection states consume at most **35** cycles per byte
- IXP28xx has about $5.4 \times$ more cycles per byte

Memory size

- **Instruction store**
  - 4k instructions up to $\sim 200$ states
- **SRAM**
  - up to 32MB up to $\sim 64k$ states

Rewriting

- Expensive unaligned access to DRAM
- Fast but tiny local memory for constructing packets
- Only a single thread per ME can do rewriting
<table>
<thead>
<tr>
<th>filter</th>
<th>states</th>
<th>instructions</th>
<th>insns/state</th>
<th>interpreted states</th>
</tr>
</thead>
<tbody>
<tr>
<td>anon</td>
<td>19</td>
<td>641</td>
<td>30.05</td>
<td></td>
</tr>
<tr>
<td>anonhdr</td>
<td>19</td>
<td>641</td>
<td>30.05</td>
<td></td>
</tr>
<tr>
<td>backdoor</td>
<td>2441</td>
<td>46041</td>
<td>18.83</td>
<td>2147</td>
</tr>
<tr>
<td>large</td>
<td>2327</td>
<td>19216</td>
<td>8.23</td>
<td>2141</td>
</tr>
<tr>
<td>payload</td>
<td>24</td>
<td>400</td>
<td>13.75</td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>1</td>
<td>145</td>
<td>6.00</td>
<td></td>
</tr>
</tbody>
</table>
# Pattern-matching performance

<table>
<thead>
<tr>
<th>packet size 64</th>
<th>packet size 96</th>
<th>packet size 546</th>
<th>packet size 1518</th>
</tr>
</thead>
<tbody>
<tr>
<td>531.9 Mbit/s</td>
<td>751.2 Mbit/s</td>
<td>962.1 Mbit/s</td>
<td>990.8 Mbit/s</td>
</tr>
<tr>
<td># of MEs</td>
<td># of MEs</td>
<td># of MEs</td>
<td># of MEs</td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
<td>1 2 3 4 5 6</td>
<td>1 2 3 4 5 6</td>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td>backdoor</td>
<td>backdoor</td>
<td>backdoor</td>
<td>backdoor</td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>large</td>
<td>large</td>
<td>large</td>
<td>large</td>
</tr>
<tr>
<td>30 0 0 0 0 0</td>
<td>69 34 5 0 0 0</td>
<td>85 70 54 39 24 9</td>
<td>86 72 58 44 30 16</td>
</tr>
<tr>
<td>payload</td>
<td>payload</td>
<td>payload</td>
<td>payload</td>
</tr>
<tr>
<td>3 0 0 0 0 0</td>
<td>50 0 0 0 0 0</td>
<td>71 43 14 0 0 0</td>
<td>73 46 20 0 0 0</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Percentage of dropped (not processed) packets

![Graph showing the percentage of dropped packets vs. number of MEs for different packet sizes.](image)
Rewriting performance

Synthetic traffic

<table>
<thead>
<tr>
<th>Packet Size</th>
<th>64</th>
<th>96</th>
<th>546</th>
<th>1518</th>
</tr>
</thead>
<tbody>
<tr>
<td># of MEs</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>anon</td>
<td>70</td>
<td>47</td>
<td>26</td>
<td>16</td>
</tr>
<tr>
<td>anonhdr</td>
<td>55</td>
<td>18</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Real traffic

<table>
<thead>
<tr>
<th>Packet Size</th>
<th>305.0</th>
</tr>
</thead>
<tbody>
<tr>
<td># of MEs</td>
<td>1 2 3</td>
</tr>
<tr>
<td>anonym</td>
<td>78 37</td>
</tr>
<tr>
<td>anonymhdr</td>
<td>3 0 0</td>
</tr>
</tbody>
</table>

Percentage of dropped (not processed) packets
We developed **Ruler** - a language and compiler

**Ruler** supports a wide range of architectures including NPUs, FPGAs and standard general-purpose CPUs

**Ruler** offers pattern matching and packet *rewriting*

**Ruler** makes programming NPUs simple

**Ruler** is directly portable to current and upcoming multi-core chips e.g., Niagara1 and Niagara2

We evaluated **Ruler** on real hardware using Intel IXP 2400

Sponsors: EU FP6 Lobster project, Intel IXA University Program
Thank you for your attention

Questions ...

Sponsors : EU FP6 Lobster project, Intel IXA University Program