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 preventions 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

```
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, addresss, 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

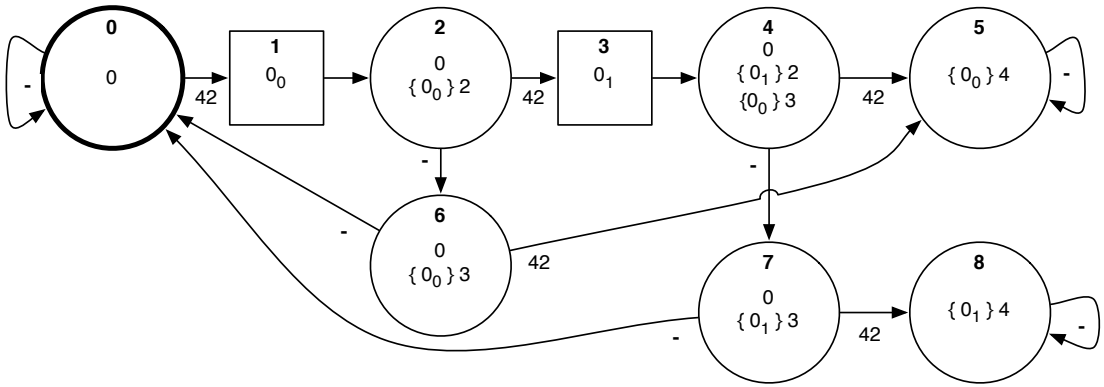
- Deterministic Finite Automaton for matching multiple patterns

  state types `inspection, memory inspection, jump, tag, accept`

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

```
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

```
__declspec(shared gp_reg)
__declspec(sram)
__declspec(shared scratch)

__declspec(dram_read_reg)
```

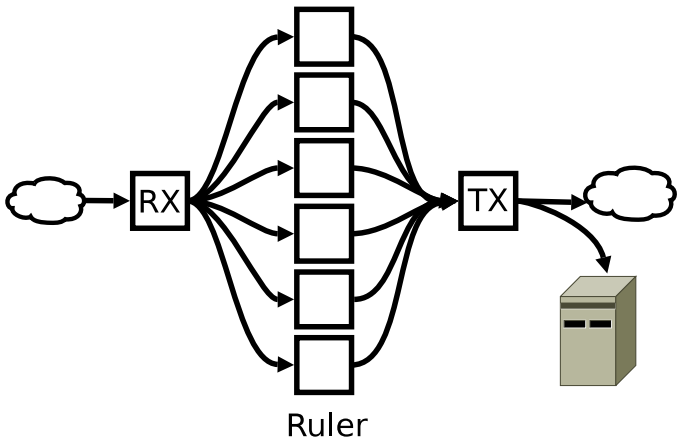
IXP2xxx

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

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 $\Rightarrow$ need optimization

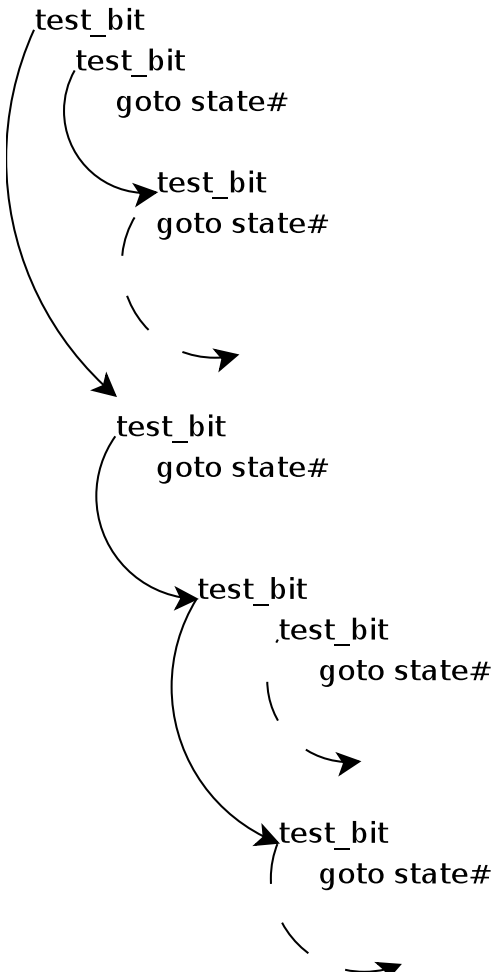
Reading the next byte from the input

- 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#]
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

```
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#]
```

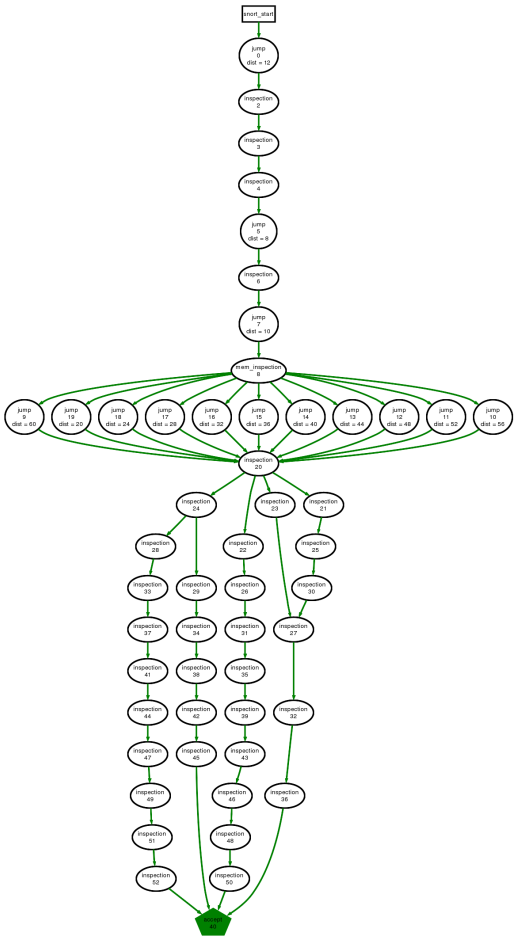
Binary tree

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

Executed vs. interpreted states

Instruction store is limited ⇒ executed and interpreted states

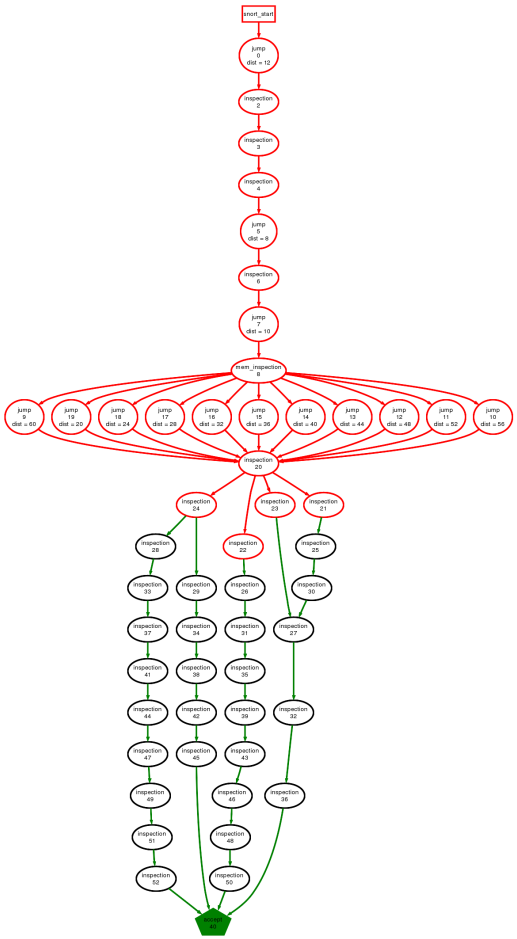


Simplified DFA, loop edges are missing

- Number of states may explode exponentialy
- 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

Executed vs. interpreted states

Instruction store is limited ⇒ executed and interpreted states

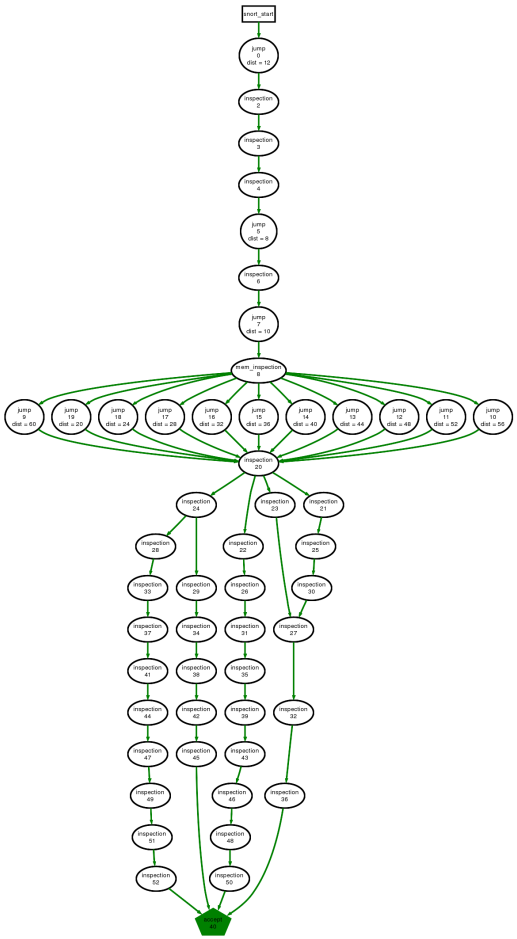


Simplified DFA, loop edges are missing

- Number of states may explode exponentialy
- 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

Executed vs. interpreted states

Instruction store is limited ⇒ executed and interpreted states



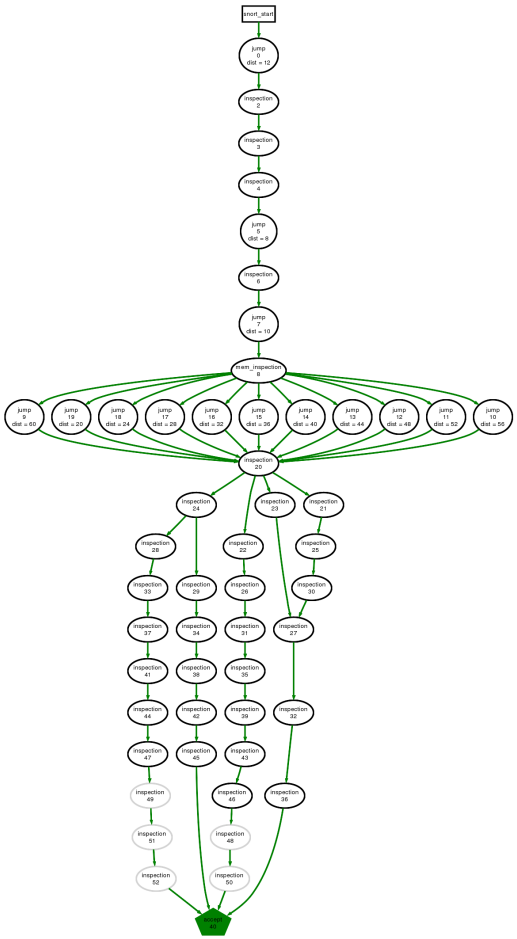
Simplified DFA, loop edges are missing

- Number of states may explode exponentialy
- 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

Executed vs. interpreted states

Instruction store is limited ⇒ executed and interpreted states



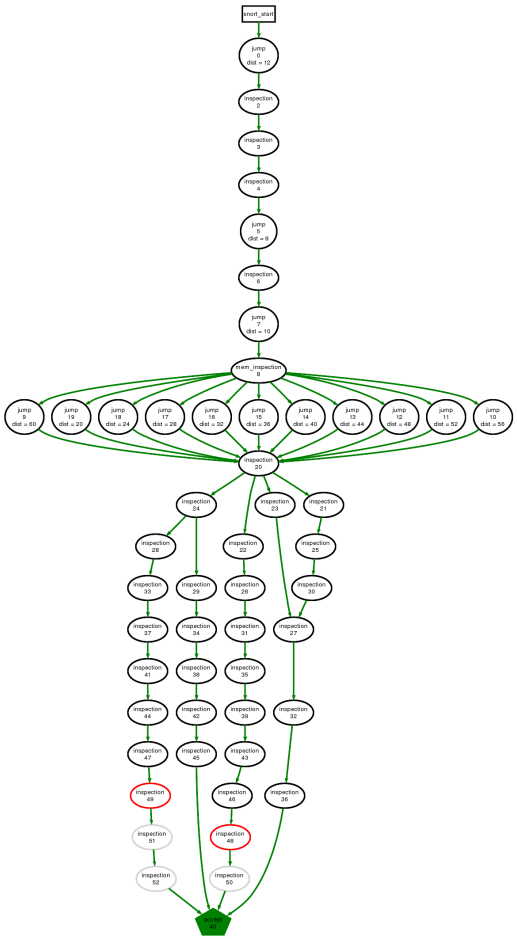
Simplified DFA, loop edges are missing

- Number of states may explode exponentialy
- 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

Executed vs. interpreted states

Instruction store is limited ⇒ executed and interpreted states



Simplified DFA, loop edges are missing

- Number of states may explode exponentialy
- 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**× more cycles per byte

Memory size

Instruction store	4k instructions	up to ∼200 states
SRAM	up to 32MB	up to ∼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

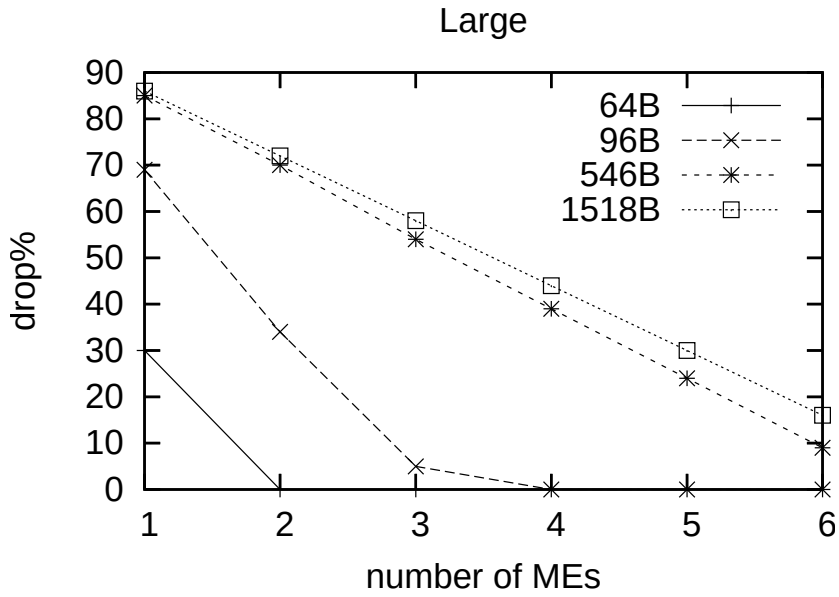
Benchmark filters

filter	states	instructions	insns/state	interpreted states
anon	19	641	30.05	
anonhdr	19	641	30.05	
backdoor	2441	46041	18.83	2147
large	2327	19216	8.23	2141
payload	24	400	13.75	
null	1	145	6.00	

Pattern-matching performance

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

Percentage of dropped (not processed) packets



Rewriting performance

Synthetic traffic

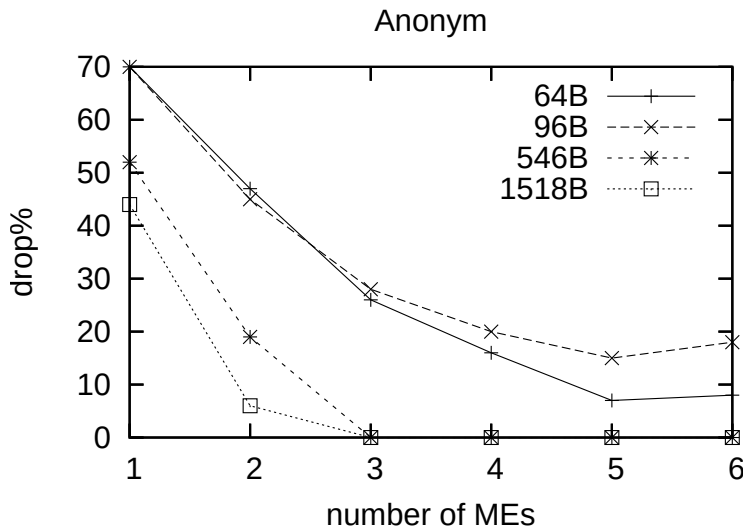
	packet size 64						packet size 96						packet size 546						packet size 1518					
	531.9 Mbit/s						751.2 Mbit/s						962.1 Mbit/s						990.8 Mbit/s					
# of MEs	**1**	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**	**5**	**6**
anon	70	47	26	16	7	8	70	45	28	20	15	18	52	19	0	0	0	0	44	6	0	0	0	0
anonhdr	55	18	3	0	0	0	52	17	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Real traffic

av. pkt size 305.0

829.0 Mbit/s

# of MEs	**1**	**2**	**3**	**4**	**5**	**6**
anonym	78	37	2	0	0	0
anonymhdr	3	0	0	0	0	0



Percentage of dropped (not processed) packets

Summary

- 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