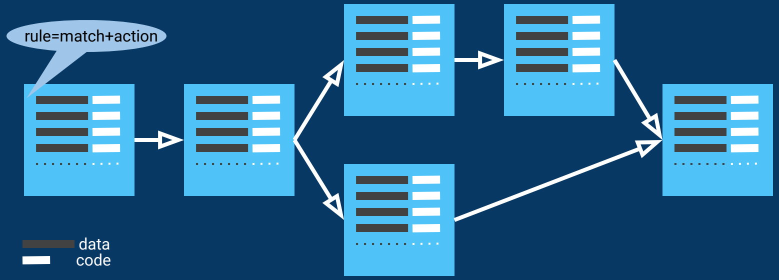
C like DSL for Open vSwitch

Saurabh Shrivastava
@gokodogo

nuagenetworks

Packet Processing Pipeline

- Processing pipeline has several stages
- Each stage is a table of rules
- Each rule has “match” and “actions”
- “data”: the “match” part of a rule and the constants in the “actions” of a rule
- “code”: anything which is not “data”

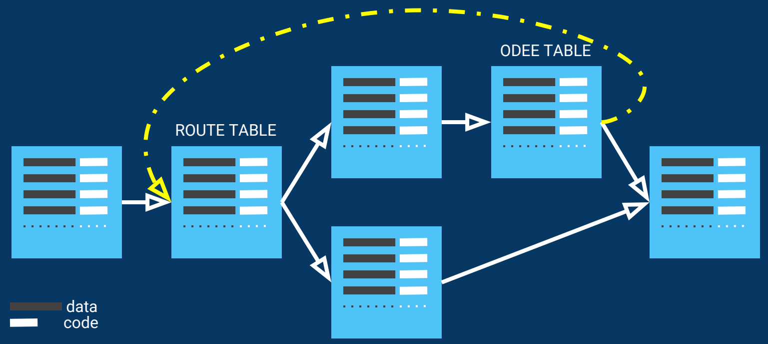


A Packet Processing Pipeline

Code mixed with Data

- Table rules have “code” and “data”
  ⇨ update of “code” can’t happen independent of “data” e.g. bug fixes need only “code” changes

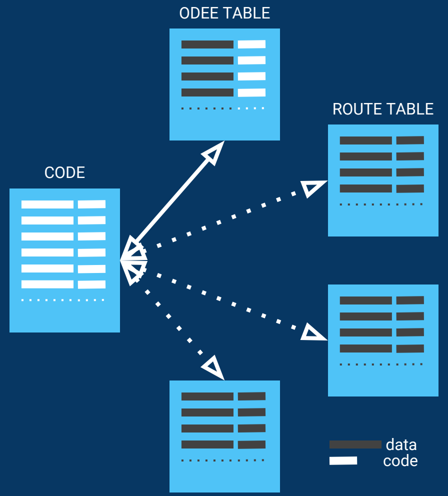
- Table modifies state (due to “code”)
  ⇨ table can’t be reused easily e.g. need for route lookup to be done on source address for RPF checks in addition to lookup done for destination address



A Packet Processing Pipeline with loops

Code separate from Data

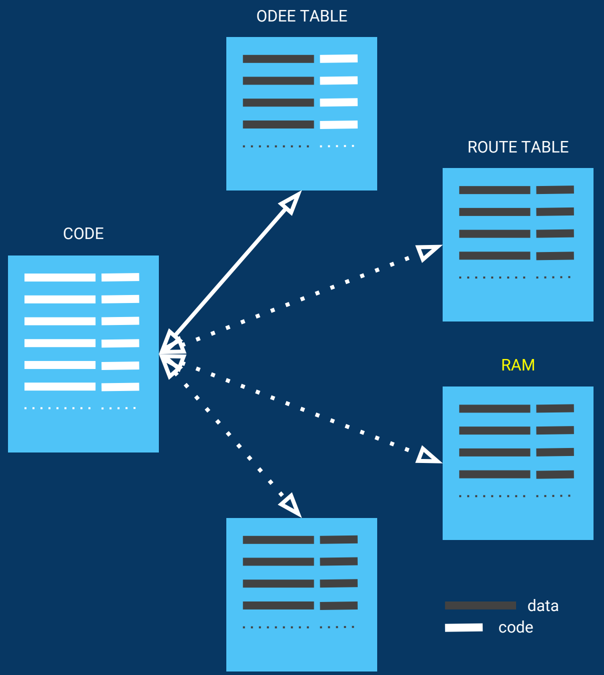
- Table that has only “code”
  ⇨ can be updated without touching forwarding state (“data”)
  ⇨ can achieve 0 downtime!
- Table that has only “data”
  ⇨ no side effects of updating state
  ⇨ table can be reused e.g. ROUTE TABLE



Packet Processing Pipeline implemented by CODE TABLE

A different way to organize tables

- CODE table encodes forwarding logic
- A non-CODE table implements a lookup "function" e.g. ROUTE table
- Code can "call" these "function"s
- RAM table implements a function which takes in a 32b address and returns its 32b contents
- Data structures can be laid down in RAM



Programming model

```
void main (void)
{
    struct context cx = context_lkup (NXM_OF_IN_PORT[], NXM_OF_ETH_SRC[]);
    if (!cx.cx_tenant) { stats_inc (NXM_OF_IN_PORT[], E_NO_TENANT); goto done; }
    u32 tenant_id = cx.cx_tenant->te_tenant_id;
    if (!cx.cx_tenant->te_oper_state_up) { stats_inc (tenant_id, E_OPER_DOWN); goto done; }
    u32 l3_vrf_id = cx.cx_l2subnet->l2_l3vrf->l3_vrf_id;
    u8 ok = acl_lkup (l3_vrf_id); /* rest of match criteria from pkt fields */
```

CONTEXT table lookup

flow fields are “well known” global variables

RAM table lookup

- Flow fields are “well known” global variables e.g. NXM_OF_ETH_SRC[]
- Writing to a global variable updates the corresponding flow field
- Each packet goes through processing starting at “main”
- “main” may call other functions and lookup RAM
- All the writes done to global variables before returning from “main” constitutes the actions to be performed on the packet

Why another programming model?

- Use a model which has hi-fidelity with the real world so that event in the real world can be translated to the model as-is
  - forwarding code doesnt change (CODE)
  - tenant info changes at a slower rate (RAM)
  - forwarding state change at a faster rate (ROUTE)
- (all other reasons why there are several programming languages)

Why a higher level language?

- Coding at higher level of abstraction

  ⇨ no worry about register liveness, function call setup, ...

  ⇨ less lines of code

  ⇨ less things to juggle in mind

  ⇨
  - less bugs
  - higher feature velocity
  - lower barrier to entry to write forwarding code
  - (all other reasons why C is better than assembly)

Why C?

- “closer to the metal” i.e. each C statement translates to few deterministic number of instructions
- Engineers already familiar with C
- Good optimizing compiler available which can optimize use of registers
- Mature static analysis tools available which can tell worst case code path and worst case register usage

Open vSwitch can simulate a stack based processor

- Several match tables

  ⇨ table 0 for CODE, table 1 for RAM, ..
- Several registers

  ⇨ store intermediate state while executing code
- Stack

  ⇨ perform function calls
- goto_table, resubmit

  ⇨ jump to different parts of code
- Atomic transactions for grouping updates

  ⇨
  - atomically update structs in RAM (software transactional memory)
  - atomically update all of CODE and achieve 0 downtime

CODE (table 0)

```
100 void main (void)
110 {
120     struct context cx;
130     cx = context_lkup (NXM_OF_IN_PORT[],
140                         NXM_OF_ETH_SRC[]);
150     if (!cx.cx_tenant) {
160         stats_inc (NXM_OF_IN_PORT[],
170                    E_NO_TENANT);
180     }
```

```
000   priority=0, actions=
        load:130->NXM_NX_REG0[0..31],goto_table:0
130   reg0=130, priority=1, actions=
141   reg0=141, priority=1, actions=
150   reg0=150, priority=2, reg9=0, actions=
150   reg0=150, priority=1, actions=
160   reg0=160, priority=1, actions=
...   ......... .......... ........
```

- Table 0 match criteria is only reg0 which is the “program counter”
- First rule to get executed is 000, a priority 0 rule to jump to the start of “main”
- function call uses one rule to make the call and another one to process result
- “if” uses priority 1 and 2 with same reg0 value
- All other rules are priority 1

Calling a function

```
130 cx = context_lkup (NXM_OF_IN_PORT[],
140                    NXM_OF_ETH_SRC[]);
141
150
```

- Save registers on stack
- Push return address on stack
- Load arguments in registers reg1 onwards
- Jump to table implementing the function

```
130   reg0=130, priority=1, actions=
         push:NXM_NX_REG0[0..31], push:NXM_NX_REG1[0..31], push:NXM_NX_REG2[0..31], # save regs
         push:NXM_NX_REG3[0..31], push:NXM_NX_REG4[0..31], push:NXM_NX_REG5[0..31], # ...
         push:NXM_NX_REG6[0..31], push:NXM_NX_REG7[0..31], push:NXM_NX_REG8[0..31], # ...
         push:NXM_NX_REG9[0..31],                                                   # ...
         load:141->NXM_NX_REG1[0..31], push:NXM_NX_REG1[0..31], # push return address 141
         load:NXM_OF_IN_PORT[0..31]->NXM_NX_REG1[0..31], # load IN_PORT in reg1
         load:NXM_OF_ETH_SRC[0..31]->NXM_NX_REG2[0..31], # load ETH_SRC in reg2
         load:NXM_OF_ETH_SRC[32..47]->NXM_NX_REG3[0..15], # ... and reg3
         goto_table:210                                  # jump to CONTEXT table (table 210)
```

Processing function return value

```
130 cx = context_lkup (NXM_OF_IN_PORT[],
140                    NXM_OF_ETH_SRC[]);
141
150
```

```
struct context {
    struct tenant *cx_tenant;
    struct l2subnet *cx_l2subnet;
};
```

- Pop return value into registers
- Pop saved registers
- Jump to next statement

```
141   reg0=141, priority=1, actions= # process return values from call to CONTEXT table, jump to 150
        pop:NXM_NX_REG9[0..31],                                  # pop cx.cx_tenant into reg9
        pop:NXM_NX_REG8[0..31],                                  # pop cx.cx_l2subnet into reg8
        pop, pop                                                 # pop and discard saved reg9, reg8
        pop:NXM_NX_REG7[0..31], pop:NXM_NX_REG6[0..31],          # pop and restore reg7 and reg6
        pop:NXM_NX_REG5[0..31], pop:NXM_NX_REG4[0..31],          # pop and restore reg5 and reg4
        pop:NXM_NX_REG3[0..31], pop:NXM_NX_REG2[0..31],          # pop and restore reg3 and reg2
        pop:NXM_NX_REG1[0..31], pop:NXM_NX_REG0[0..31],          # pop and restore reg1 and reg0
        load:150->NXM_NX_REG0[0..31], goto_table:0               # jump to next statement (150)
```

Function implementation

```
# <10,00:aa:bb:cc:dd:ee> -> <0x0001a004,0x0008a044>
reg1=0x10, reg2=0x00aabbcc, reg3=0xddee, actions=                # match on arguments
  pop:NXM_NX_REG0[0..31],                                        # load return address
  load:0x0001a004->NXM_NX_REG1[0..31], push NXM_NX_REG1[0..31],  # push cx.cx_l2subnet=0x0001a004
  load:0x0008a044->NXM_NX_REG1[0..31], push NXM_NX_REG1[0..31],  # push cx.cx_tenant=0x0008a044
  goto_table:0                                                   # jump back to caller

# lookup failed
priority=0, actions=
  pop:NXM_NX_REG0[0..31],                                        # load return address
  load:0x0->NXM_NX_REG1[0..31], push NXM_NX_REG1[0..31],         # push cx.cx_l2subnet=0x0
  load:0x0->NXM_NX_REG1[0..31], push NXM_NX_REG1[0..31],         # push cx.cx_tenant=0x0
  goto_table:0                                                   # jump back to caller
```

- Match on function arguments
- Pop return address from stack
- Push return value on stack, jump back to caller

RAM (table 1)

```
reg1=0x00010000, actions=load:0xabcdabcd->NXM_NX_REG1[0..31] # address 0x00010000 => 0xabcdabcd
reg1=0x00010004, actions=load:0xfefefefe->NXM_NX_REG1[0..31] # address 0x00010004 => 0xfefefefe
...
priority=0, actions=exit # address not found => exception
```

- Match on 32b address, load 32b contents of address
- Accessing uninitialized memory causes exit
- Complex data structures can be laid down in memory
- Bundle transactions can be used to update multiple addresses atomically ⇨ explicit synchronization between reader (forwarding code) and writer (controller) not needed - aka “software transactional memory”
- There is no explicit jumping back to caller because caller uses “resubmit” instead of “goto_table”

Pointers

```
200 u32 tenant_id;
210 tenant_id = cx.cx_tenant->te_tenant_id;
```

```
struct tenant {
    u8 te_oper_state_up :1;
    u32 te_tenant_id;
};
```

- If struct starts at 1K boundary and is at most 1K in size, address of a struct member is address of struct with the bottom 10b set as the offset of the member

```
210   reg0=210, priority=1, actions=
        load:NXM_NX_REG9[0..31]->NXM_NX_REG1[0..31], # load cx.cx_tenant in reg1
        load:4->NXM_NX_REG1[0..10],      # load offsetof (struct tenant, te_tenant_id) in bottom 10b
        resubmit (,1)                    # get cx.cx_tenant->te_tenant_id in reg1 by RAM lookup
        load:NXM_NX_REG1[0..31]->NXM_NX_REG8[0..31],   # tenant_id = cx.cx_tenant->te_tenant_id
        load:220->NXM_NX_REG0[0..31], goto_table:0     # jump to next statement (statement 220)
```

“if”

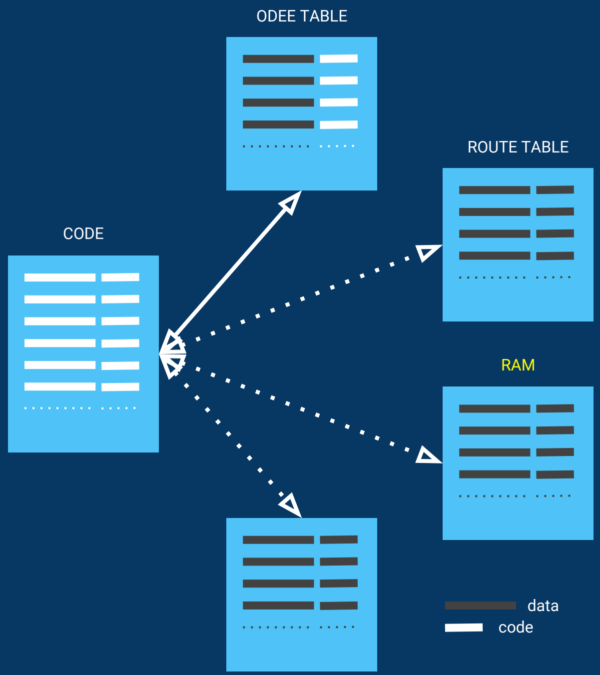
```
150 if (!cx.cx_tenant) {
160       stats_inc (NXM_OF_IN_PORT[],
170                  E_NO_TENANT);
180 }
```

- Two rules with same reg0 value but different priorities
- Higher priority rules matches on the condition being 0 or false
- The two rules jump to different locations

```
150   reg0=150, priority=2, reg9=0,actions=         # if cx.cx_tenant is NULL, jump to 160
         load:160->NXM_NX_REG0[0..31], goto_table:0
150   reg0=150, priority=1,        actions=         # if non NULL, jump to 210
         load:210->NXM_NX_REG0[0..31], goto_table:0
```

Next steps

- Make Open vSwitch a LLVM backend, so that clang C compiler can be used
- Can "asm" can be used to embed OVS instructions in the DSL
- Can service insertion be simulated as a context switch where all state is saved and packet is sent out, state is restored when packet comes back and processing continues from where it had left off



Thank you

nuagenetworks