LINC fast path and OpenFlow spec

Maxim Kharchenko, Cloudozer LLP

04/01/2014

1 Summary

- The suggested approach is suitable for reimplementation of the fast path of the LINC switch.
- ISSUE: If a lot of packets are routed to the controller (packet-in messages) the performance will degrade as the fast path will effectively include the management stack.
- The management of counters and packet encapsulation can be made much faster using (non-portable) changes to the Erlang VM.

2 Overview

The document discusses reimplementation of the “fast path” of the LINC switch, the portion of the switch critical for its throughput. The OpenFlow Switch Specification 1.4 (1) and the LINC switch sources form basis for the discussion. The primary goal is to confirm that all functions on the fast path can be reimplemented using the suggested approach, to estimate the expected processing delay, to pinpoint tradeoffs and potential problem areas.

3 The Approach

The suggested approach for the fast path reimplementation:

- Map flow tables to functions (not ETS tables);
- Use Erlang pattern matching for matching packets against flow entries;
- Generate/compile/reload matching functions when flow tables change;
- Minimize constructs that use process heap;
- Avoid garbage collection;
- Reuse existing code of the LINC switch, if possible.

The target processing delay of the enhanced LINC switch is 25μs (150μs max).

4 Erlang functions

Erlang functions can take no more than 255 arguments. Both LING and BEAM has 255 registers. Arguments are passed to functions in registers and thus a function can have only up to this many arguments. This constraints what OpenFlow entities can potentially be mapped to function arguments. For instance, if ports are mapped to arguments then there cannot be more than 255 ports.

5 Flows tables are functions

Example:

```
%%
%% Hdr1, Hdr2, Hdr3 are protocol headers
%% P1, P2, P3 are OpenFlow ports
%%
flow0(<<0,0,0,_/binary>> = Hdr1, Hdr2, Hdr3, Payload, P1, P2, P3) ->
    drop;
flow0(<<0,0,1,_/binary>> = Hdr1, Hdr2, Hdr3, Payload, P1, P2, P3) ->
    send(..., P1);
flow0(<<0,0,2,_/binary>> = Hdr1, Hdr2, Hdr3, Payload, P1, P2, P3) ->
    flow1(..., P2);
flow0(_, _, _, _, _, _, _) ->
    table_miss.
```

`flow0()` and other matching functions must be regenerated/compiled/reloaded when the flow table changes.

6 OpenFlow ports

The OpenFlow ports should be mapped to ordinary Erlang ports, if possible. Alternatively, OpenFlow ports can be represented as Erlang processes. There is no need to use gen_server processes for this.

The LOCAL reserved port creates a difficulty. This port lets the exchange of in-band OpenFlow messages. These messages can modifiy flows and thus invoke the lengthy regeneration of matching functions. The solution is to perform the regeneration asynchronously. This means that certain number of packets will follow the earlier rules until the regeneration completes.

The CONTROLLER reserved port is a nuisance too. The packet-in messages effectively include the management stack of the switch into the fast path. The only consolation is that packet-in messages are relatively rare.

7 Counters

Counters that are pervasive in the OpenFlow switch create certain difficulty for an efficient implementation. The fastest implementation of a counter is this:

```
flow0(Counter,...) ->
    ...
    flow0(Counter +1,...);
...
```

That is, a counter variable is kept in a register. The problem with this implementation is twofold. Firstly, the counters are 32-bit (or 64-bit) and they wrap to zero on overflow. Small integers in LING range from -536870912 to 536870911 (29 bits). When they go outside this range they become bignums and bignums take space on the heap and are orders of magnitude slower. This can be tackled by using two registers for a counter:

```
flow0(Counter0, Counter1,...) ->
    carry(Counter0+1, Counter1,...);
...

carry(0, 16#100,...) ->
    carry(0, 0,...);
carry(16#1000000, Counter1,...) ->
    carry(0, Counter1+1,...);
carry(Counter0, Counter1,...) ->
    flow0(Counter0, Counter1,...).
```

Counter0 keeps 24 bits of the counter, Counter1 keeps remaining 8 bits. A 64-bit counter would require 3 variables/registers.

Yet another issue with the approach is that there are just too many counters to put them all into registers, especially when 2 or 3 are needed for each counter. The solution is to keep the counters in a dictionary (not an ETS table). This is much slower and grows heap on each update of the counter. Probably, the best is the hybrdid approach when counters that update most often are kept in registers while the rest live in a dictionary or dictionaries.

To squeeze even more performance from the switch counters can be added to LING as an extension. This functionality is needed by many applications, yet it does not map well to existing Erlang primitives. The counters interface can be modelled after the timers interface:

```
erlang:new_counter(Bits)
erlang:increment_counter(CntrRef)
erlang:read_counter(CntRef)
erlang:release_counter(CntRef)
```

OpenFlow designers too think that counters sometimes affect performance:

New flow-mod flags have been added to disable packet and byte counters on a per-flow basis. Disabling such counters may improve flow handling performance in the switch. (1, p.198)

8 Timeouts

Efficient implementation of timeouts requires a thorough thought. A ***hard_timeout*** of a flow entry can be implemented by calculating the time until the closest such timeout and using the timeout clause:

```
loop(...) ->
    receive
    {Port,{data,Packet}} ->
        ...
        flow0(...);
    after T ->
        ...
        %% remove expired flow entries
        loop(...)
    end.
```

The ***idle_timeout*** is trickier. Such timeout should restart each time a flow entry activates. The efficient implementation of such timeouts should be based on a periodic operation that sweeps away expired idle entries. Such operation can run every 0.5s.

9 Matching fields

The specification (1, p.55) lists 42 possible match fields. Most of these fields belong to respective protocol headers. The most efficient matching can be achieved by splitting the incoming packet into a sequence of headers and passing these headers as binaries to the matching function that implements a flow table. The clauses of the functions (or flow entries) will contain binary patterns select a particular bit range of a header and bind it to a value. For instance,

```
flow0(<<0,1,2,3,4,5,_/binary>> = _EtherHdr, _IpHdr, _TcpHdr, Data,...) ->
    ... OXM_OF_ETH_SRC is 00:01:02:03:04:05
flow0(_EtherHdr, <<_:16/binary,1,2,3,4,_/binary>> = _IpHdr, _TcpHdr, Data,...) ->
    ... OXM_OF_IPV4_DST is 1.2.3.4
```

A packet may have multiple headers of the same type but the matching always applies to the outermost one.

This approach works neatly for masked fields too. Most often the mask will contain a single range of consecutive bits, such as `0xfffc0000` — top 14 bits. The Erlang binary matching pattern will represent such masked match efficiently.

10 Dynamically-generated functions

The modules generated dynamically, then compiled and (re)loaded is the key mechanism of the proposed approach for the fast path reimplementation. The generation/compilation/loading cycle is relatively slow when compared to an update to an ETS table. Yet the rate of flow entry updates is much-much lower than the rate of the data flow. We should take our time to digest the flow table updates to the maximum extent to make the "fast" path as fast as possible. The smallest possible portion of the system should be regenerated. If possible, the regeneration should affect only functions mapped to flow tables.

If implemented as follows the code will switch to the new code automatically:

```
loop(...) ->
    receive
    {Port,{data,Data}} ->
        flow0:incoming(Port, Data),
        loop(...)
    end.
...

-module(flow0).

incoming(Port, Data,...) ->
    {Hdr1,Hdr2,Hdr3} = decapsulate(Data),
    match(Hdr1, Hdr2, Hdr3, Port,...).

match(<<1,2,3,0,0,0,_/binary>>, Hdr2, Hdr2, Port,...) ->
    drop;
...
```

The `flow0:incoming()` function is needed because the signature of the match function can change.

11 Instructions/actions

The instruction and actions are described by the OpenFlow specification do not contain any stumbling blocks. Most of the existing code can be reused.

12 Meters/Queues

The best way to implement meters and queues is map them to Erlang processes. Their implementation should not use dynamic generation of modules. Most probably, the existing code can be reused.

13 Packet encapsulation

Earlier study showed that the packet encapsulation had high latency cost. One of the causes is a poor current implementation that disregards the Erlang behaviour when concatenating binaries. In addition, the calculation of checksums for all protocol headers does add up.

The fastest implementation of the packet encapsulation is still to be determined.

14 Garbage collection

The standard Erlang garbage collection may introduce undesirable "hiccups" in the data flow or lead to memory overflow. The memory overflow is likely to happen because heuristics the Erlang VM uses to decide when to collect garbage are not particularly suitable for the fast-paced software switch. The band-aid solution is to run `erlang:garbage_collect()` manually after processing a given number of packets. This does not fix the "hiccups" though.

A better solution is to periodically restart the fast path process and start from the clean slate. It is important to drain the mailbox of the old process properly:

```
    NewPid = spawn(...),
    ...
    erlang:port_connect(P1, NewPid),
    erlang:port_connect(P2, NewPid),
    erlang:port_connect(P3, NewPid),
    ...
    drain_packets(NewPid)
    ...

drain_packets(NewPid) ->
    receive
    {_Port,{data,_Data}} =X ->
        NewPid ! X,
        drain_packets(NewPid)
    after 0 ->
        ok
    end.
```

`drain_packets()` reinserts the packets received by the old process while the substitute process has been being created. Afterwards we can destroy the old process. This is much faster than GC.

The only drawback of the approach is that the order of packets changes. If needed, the `drain_packets()` can be rewritten to preserve the ordering.

Bibliography

[1] OpenFlow Switch Specification 1.4 (https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf) 1, 4