The latency cost of Erlang primitives

Maxim Kharchenko, Cloudozer LLP

18/01/2014

1 Summary

1. Analysis of latency costs of Erlang primitives confirms the approach for the faster LINC backend: use dynamically-recompiled rule matching function, avoid ETS tables and message passing.

2. The new LINC backend should have an average processing delay of 25μs. The maximum delay is 150μs.

2 Overview

Each piece of network equipment introduces a delay on the path of a network frame. The delay related to processing of the frame inside the box is referred to as 'processing delay' in the literature[1].

This document shows how adding Erlang constructs to the fast path of the software switch affects the processing delay. The latency cost of an Erlang construct is the increase in the processing delay associated with the use of the construct.

3 Baseline delay

The latency cost is the increase of the processing delay. If no processing takes place in the switch there is still a delay associated with Erlang ports, scheduling, etc. This baseline delay is established by nullx experiment. All intervals are constructed for 95% confidence level.

$$Baseline\ delay = 4.71 \pm 0.66 \mu s$$

The possibility to lower the baseline delay is small. The baseline delay is the low bound of the latency we can ever achieve in the software switch based on LING VM.

4 Latency cost

To measure the latency cost of an Erlang primitive it was added to the fast path of the nullx forwarder. The processing delay was measured by the instrumentation of the LING VM. The baseline delay was subtracted to calculate the latency cost. See Table 1 below.

[1] globecom2004.pdf

Table 1: Latency cost

Construct	LC, us	Note
pkt:decapsulate()	4.1	
pkt:encapsulate()	27.0	
Message passing	3.4	barebone
Message passing	8.0	gen_server
ETS lookup()	2.9	100 records
ETS lookup()	4.7	10,000 records
ETS update_counter	3.6	
ETS tab2list()	32.2	100 records
Pattern matching	0.3	10 rules (UPENN CIDRs)
Pattern matching	4.0	911 rules (CHINANET CIDRs)

5 Observations

1. pkt:encapsulate() is very slow. The trace shows that the function uses setelement() many (six) times. Each time the record gets copied.

2. ets:tab2list() is slow. This is expected. The function is more suitable for a database backup, not for the fast path of the software switch.

3. The pattern matching is quite efficient. A realistic routing over almost a thousand rules adds only 4μs.

6 Setting targets

The following line of reasoning is not strictly rooted in evidence. Yet it is better to have targets set using the gut feeling then to proceed without.

The nullx forwarder shows the throughput of 2.75Gbit/s. Most probably the thoughtput in this case is capped by latencies of other elements on the path of packets, such as Linux bridges. These latencies are out of scope of the current effort.

The hardware routers have processing latencies starting at 10μs. The faster hardware may use ASICs or FPGAs. The boxes that do deep packet inspection may have a processing delay of 1000μs.

The low bound of the processing delay we can achieve is established earlier (chapter 3). We cannot go lower than 4.71μs. There is also a data point for unsatisfactorily long delay — 204.2μs.

Given the above the following target processing delays suggest themselves:

$$Average\ target\ delay = 25\mu s$$

$$Max\ target\ delay = 150\mu s$$

The faster backend we are going to build should never process a network frame longer then the max target delay. The idea of configurations that can be used to verify the target delays are as follows:

1. Average target delay: 4 ports, 20 rules

2. Max target delay: 32 ports, 200 rules

7 Appendix

7.1 The fast path as implemented

The processing delay of the LINC switch in the simplest confguration is as follows:

$$LINC\ delay = 204.2 \pm 7.5\mu s$$

In the basic configuration of the LINC switch the following sequence of events happens for each network frame:

1. A frame retrieved from the mailbox using `{Port,{data,Frame}}` pattern.
2. Enter handle_frame().
3. Check if `no_recv` flag is present. This requires 2 calls to lists:member().
4. Decode the frame using linc_us4_packet:binary_to_record(). The Ethernet type, the source and destination MAC addresses get extracted using pkt:decapsulate().
5. rx counters updated by calling ets:update_counter().
6. `no_packet_in` option checked. This requires upto two calls to lists:member().
7. The call to linc_us4_routing:spawn_route(LincPkt) spawns a new process that executes proc_lib:spawn_link(). Everything below happens on the new spawned thread.
8. proc_lib:spawn_link() performs a few unwieldy preparatory steps in addition to calling erlang:spawn_link().
9. Enter linc_us4_rounting:route().
10. A flow table retrieved using tab2list(). All rules get copied from ETS space to Erlang space. This is like making a database dump for each query. BTW, tab2list() is a wrapper over ets:match_object() function.
11. The packet gets matched against all entries of the flow table. The match_flow_entry() is simple.
12. More checks on the contents of the packet performed by pkt_fields_match_flow_fields() function. The function applies a matching fun to all flow fields.
13. The match fun applies yet another matching to all packet fields. Even in the most trivial case the matching takes about two dozen function invocations per flow table entry.
14. The second flow table entry matches and ETS table flow_table_counters gets updated.
15. linc_us4_instructions:apply() convert instructions from the flow table into actions. The actions are applied by linc_us4_actions:apply_list().
16. In our case the (only) action is to forward the frame to another port. linc_us4_actions:apply_list() interprets the action and calls linc_us4_port:send().

17. The matching clause of linc_us4_port:send() retrieves pid of the process associated with the port. This does two lookups to ETS tables.
18. The packets sent to the process using gen_server:cast(). The trace shows four layers of function calls before the message is actually sent and four more until the message reaches handle_cast().
19. The frame repackaged using pkt:encapsulate(). pkt:encapsulate() recalculates the ip checksum. The function calls erlang:setelement() 6 times.
20. Yet another update of the counters in ETS tables. This time it is tx counters.
21. The frame swallowed by the port. Done.