# Pores in LING (experimental)

Maxim Kharchenko, Cloudozer LLP

08/17/2014

## 1 Summary

1. A driver reimplemented in Erlang requires 3x less code. This make the driver code more maintainable and the whole virtual machine more stable.

2. Performance hit from the reimplementation is small and there may be significant performance gain for highly asynchronous scenarios.

## 2 Overview

Currently LING (and BEAM) has two kinds of active entities – processes and ports (or outlets). LING uses word 'outlet' when BEAM uses 'port'. These are synonyms. Both processes and outlets can receive event notificiations. Both maintain an intricate state including references to other processes and outlets.

The simplified primary loop of the LING scheduler is shown below.

```
scheduler_next()
{
    // The current proccess runs out of reductions or hits 'receive'.
    // Let us choose another process to run.

    // 1. Put the current proccess on the appropriate queue.
    park_process(current);

    // 2. Choose the next runnable process.
try_again:
    next = select_runnable();

    // 3. Process pending events.
    events_do_pending();

    // 4. If there is a runnable process then continue with it as a current process.
    if (next != 0)
        return next;
```

```
        // 5. There is no runnable processes - yield control to Xen.
        wait_for_events();
        goto try_again;
    }
```

For processes, events are represented as messages. The process handles them with typically one receive statement. Events targeting outlets take all kind of forms and are handled by separate functions. Process logic is written in Erlang, outlets are implemented in C. This created a large 'gray zone' in the implementation of Erlang, a zone of fragile and rigid code, source of bugs and nightmares for maintainers.

Is it possible to implement outlet logic in Erlang too and follow the same pattern when all events are messages and callbacks are neatly packaged into receive statements?

The document describes 'pores' – a much simpler replacement for outlets. It also describes the reimplmentation of certain kind of outlet as a pore and measures the impact on the code base and performance.

## 3 Pores

A 'pore' is a stripped-down passive version of an outlet. Certain data structures cannot map to Erlang types. The typical example is a memory page shared between Xen domains. It must remain at a fixed page-aligned location. An Erlang binary is not enough here. Pores encapsulate such data structures.

Similarly to an outlet, a pore has an owner process. The owner process of a pore never changes. In addition it is not possible to 'link' to or 'register' a pore. `open_port()`, `port_control()`, `port_command()` do not work on pores.

Generally it much easier to 'close' a pore when its owner process dies or using `pore:close()`. A pore is passive, it does not maintains references to other data structures or promises that a certain process will eventually receive a message from it. Closing a pore mostly boils down to releasing its memory.

## 4 Pore vs. outlet

LING has two separate implementations of Erlang interface to Xenstore - a Xenstore pore and Xenstore outlet. The table below compares the code footprint of these implementations.

| File | Language | Outlet | Pore |
|---|---|---|---|
| xenstore.erl | Erlang | 222 | 257 |
| xenstore.c | C | 119 | - |
| ol_xstore.c | C | 100 | - |
| bif_pore.c | C | - | 86 |

The pore requires 16% more Erlang code and 61% less C code when compared to an outlet. Note that C code is where rigidity and bugs are and its reduction is highly desirable.

To test relative performance of two implementations a simple code starts 'N' writers. Each writer issues 1,000 write requests to Xenstore. The parameter 'S' is the size of the value written on each request. The performance data is summarised in the table below.

| N | S | Outlet, ms | Pore, ms | Diff |
|---|---|---|---|---|
| 1 | 1 | 72 | 74 | -3.0% |
| 1 | 100 | 87 | 97 | -12.3% |
| 3 | 1 | 161 | 135 | 16.5% |
| 3 | 100 | 191 | 147 | 23.1% |
| 10 | 1 | 538 | 434 | 19.3% |
| 10 | 100 | 632 | 444 | 29.8% |
| 25 | 1 | 1,494 | 1,060 | 29.1% |
| 25 | 100 | 1,835 | 1,072 | 41.6% |

The maximum rate achieved by the Xen pore is 23.5k writes per second, the old outlet peaks at 18.5k. The pore is slightly slower for one writer because it is implemented mostly in Erlang. It is faster than the outlet when there are more then one writer because C implementation uses 'busy loops' that block the CPU. In theory, it possible to avoid busy loops in C code and improve the asynchronous behaviour of the outlet. This is likely to increase its code (and complexity) considerably.

## 5 Conclusions

1. Mostly Erlang implementation of a driver based on pores is slightly ($\sim$10%) slower than the traditional C-based approach.
2. The pore-based drivers require $\sim$50% less C code and 10–20% more Erlang code.
3. Pore-based drivers may exhibit much better asynchronous behaviour.
4. The drawback of the drivers implemented in Erlang is that they may overflow memory quickly. Hints to the garbage collector or periodic restarts of the driver process should help.