ERICSSON

VM features in OTP 20

Kenneth Lundin, Erlang/OTP, Ericsson
Erlang User Conference, Stockholm 2017

VM features in OTP 20

- enif_select
- Dirty schedulers...
- I/O scalability
- Constant data not copied in messages

enif_select
Intro about NIFs

- NIFs are **N**ative **I**mplemented **F**unctions.
- Shared library, dynamically linked into the VM
- For Integration with existing C-libraries and for time critical tasks
- ENIF API functions give the interface towards VM internals

enif_select
Intro about NIFs

- NIFs belong to an Erlang module
- a module can contain both erlang functions and NIFs
- The NIF library must be explicitly loaded by Erlang code in the same module.
- All NIFs of a module must have an Erlang implementation as well.

enif_select
Intro about NIFs

- The ENIF API is continuously enriched
- **enif_select**, an interesting newcomer
- A light version of native processes
  - setup waiting for external events (on an FD)
  - Message to an Erlang process when an event occurs

enif_select

- Can replace the driver concept
- Plan to rewrite the inet (tcp/udp/sctp) driver as NIFs
- Plan to rewrite the file driver as NIFs (sendfile using enif_select), using dirty schedulers instead of async threads
- Compared to drivers, more code can be in Erlang
- Potential to give better performance

enif_select *from the manual*

ErlNifEnv* **env**	All terms of type ERL_NIF_TERM belong to an environment of type ErlNifEnv. All API functions that read or write terms has env as the first function argument.
ErlNifEvent **event**,	the event object. On Unix systems an FD that select/poll can use
enum ErlNifSelectFlags **mode**,	type of events to wait for ● ERL_NIF_SELECT_READ, ERL_NIF_SELECT_WRITE or both ● ERL_NIF_SELECT_STOP ● A notification message is sent to the process identified by **pid**: **{select, Obj, Ref, ready_input | ready_output}** ● The notifications are one-shot only.
void* **obj**,	A resource object obtained from enif_alloc_resource. The purpose of the resource objects is as a container of the event object to manage its state and lifetime. A handle is received in the notification message as **Obj**.
const ErlNifPid* **pid**	**pid** may be NULL to indicate the calling process.
ERL_NIF_TERM **ref**	a reference or the atom undefined. Passed as **Ref** in the notifications

enif_select, *example*

Erlang

```
recv_do(Rsrc, Length, Timeout) ->
    Ref = make_ref(),
    case recv_try_nif(Rsrc, Length, Ref) of
        Bin when is_binary(Bin) ->
            {ok, Bin};
        eagain ->
            receive
                {select, Rsrc, Ref, ready_input} ->
                    recv_do(Rsrc, Length, Timeout)
            after Timeout ->
                    {error, timeout}
            end;

        {error, _}=Err ->
            Err
    end.
```

C

```
/* recv_try(Sock, Length, Ref) */
static ERL_NIF_TERM recv_try_nif(ErlNifEnv* env, int argc,
const ERL_NIF_TERM argv[])
{
...
   got = read(conn->sock,
              conn->read_bin.data + conn->read_bin.size -
conn->read_capacity,
              length ? length : conn->read_capacity);
   if (got >= length) {
...
   else if (errno != EAGAIN && errno != EWOULDBLOCK) {
       res = enif_make_tuple2(env, atom_error,
enif_make_int(env,errno));
       goto done;
   }
   conn->read_waits++;
/* errno is EAGAIN or EWOULBLOCK */
   rv =
enif_select(env, conn->sock, ERL_NIF_SELECT_READ,
conn, NULL, argv[2]);
   ASSERT(!(rv & ERL_NIF_SELECT_ERROR));
   return = atom_eagain;
}
```

Dirty schedulers

- Has been around as experimental since OTP 17.0
- Named “Dirty” because they run potentially “unclean” jobs (don’t return timely (<~1 ms) to the scheduler)
- “Dirty” schedulers are separate threads which only runs potential “dirty” jobs.
- All “normal” schedulers must always be responsive because of the:
  - lock free functionality
  - load balancing between schedulers
  - soft real time characteristics
  - ...

Dirty schedulers

- A large effort in the beginning by Steve Vinoski (OTP 17)
- Now all loose ends are put together
- Activated as default in OTP 20

“Dirty” schedulers
How it works

- Two types of dirty schedulers,
  - **CPU** (cpu bound jobs)
  - **IO** (waiting for IO events)
- A potentially blocking or long running NIF, can be run on a dirty scheduler by:
  - invoking **enif_schedule_nif**
  - setting flags in the **ErlNifFunc** entry.
- Automatically used for long GCs

“Dirty” Schedulers

CPU bound “dirty” schedulers

Normal Sch 1

Sch *N*

run queue

IO bound “dirty” schedulers

1 - 1024, default 10

Async thread pool, default 10

Dirty schedulers
in OTP 20

- Defaults:
  - One dirty `CPU` scheduler per `normal` scheduler
  - 10 dirty IO schedulers
- Configurable:
  - 1 <= CPU <= normal schedulers
  - CPU online/offline can be configured in runtime
  - IO can be configured from 1 to 1024
- “dirty” IO threads intended to replace the async thread pool used by the file driver
- dirty IO not used by OTP today

Dirty schedulers
in OTP 20 continued

- code purging not blocked by a process that is stuck on a dirty scheduler.
- In addition to “dirty” NIFs also support for “dirty” BIFs and “dirty” GC
- All GCs that potentially will take a long time are now run on dirty CPU schedulers if enabled.
  - heap size and heap fragments > 1 Mb on a 64 bit (512 kb on 32 bit)
  -

Dirty schedulers
in OTP 20 continued

- `erlang:statistics/1` inspecting scheduler and run queue states has been changed due to the dirty scheduler support
- configurable stack size for dirty schedulers

Enhanced I/O scalability

- How it works today
- First attempt to parallelize
- Second attempt (ongoing)
- Plan for introduction

Enhanced I/O scalability
How it works today

- The Erlang VM waits for external events on a FD and using poll/select
- Poll/select works with a “poll-set” = the FDs to wait for
- Update of the poll-set is done from any scheduler and requires a lock
- the lock will be a bottleneck if there are many parallel IO activities (many processes sending and receiving from sockets)

Enhanced I/O scalability
How it works today

- Internal APIs which updates the poll-set are `ENIF_select` and `driver_select`.

- Example:

```
inet:setopts(Socket,[{active,once}])
```

will result in a `driver_select`

Enhanced I/O scalability

today, (OTP 19)

Sch 1

Sch *N*

run queue

Total_Reduction_Count
4 * 4000

check_io

File descriptior
to pid/port
mapping

0	1	2	3	4	5	6	7	…	*MaxFD*
pid/ port	pid/ port	pid/ port	pid/ port	pid/ port	pid/ port	pid/ port	pid/ port	…	pid/ port

Poll set

Enhanced I/O scalability
How it works today continued

- A scheduler will check_io when:
  - there are no more jobs (empty run_queue)
  - based on a total reduction counter for all schedulers 4*MaxRedPerProcess (4000 today)
- Check_io
  - map from FD to pid or port (update port/pid value)
  - check result of poll
  - On Linux ppoll or epoll is used (use epoll with +k true|false) where false is default
  - if any FD has triggered an event then lookup corresponding port or pid and send port_signal (async) or erlang message (enif_select)

Enhanced I/O scalability

new, first attempt

Sch 1

Sch *N*

run queue

Total_Reduction_Count
4 * 4000

check_io

File descriptior
to pid/port
mapping

0	1	2	3	4	5	6	7	*MaxFD*
pid/ port	pid/ port	pid/ port	pid/ port	pid/ port	pid/ port	pid/ port	pid/ port	pid/ port

Poll set 1

Poll set *N*

Enhanced I/O scalability
new , first attempt

- Check_io works the same except that there are several poll sets, chosen based on scheduler id where the port/pid is handled.
- Pros
  - This reduces the contention and parallel IO performance is very good
- Cons
  - A system with a lot of messages between processes and not so much IO get worse performance
- Conclusion
  - Not acceptable to degrade performance for systems with low IO intensity
-

Enhanced I/O scalability Benchmarks

Avg Latency vs Num Clients

Latency in us

10000
8000
6000
4000
2000
0

kp multipoll, 10000 req/s
kp baseline 10000 req/s
kp multipoll, 50000 req/s
kp baseline 50000 req/s
kp multipoll, 75000 req/s
kp baseline 75000 req/s

0
50000
100000
150000
200000
250000
300000
350000
400000

Background clients

Enhanced I/O scalability Benchmarks

Requests per second

kp baseline
kp multiplot
nkp baseline
nkp multiplot

Requests per second achieved

0, 20000, 40000, 60000, 80000, 100000, 120000

Requests per second goal

0, 20000, 40000, 60000, 80000, 100000, 120000, 140000

100 clients

Enhanced I/O scalability
Benchmarks

mnesia mnesia_tpcb tpcb_conflict_ramcopies

Stones

11K
10K
9K
8K
7K
6K
5K
4K
3K

5

09Apr 16Apr 23Apr 30Apr 07May 14May 21May

Date

Enhanced I/O scalability
new , first attempt

Analyzis of the degraded performance

- With parallel poll-sets the schedulers:
  - all sleep in a call to poll
- current solution:
  - all sleep in a futex (except one)

- It is more expensive to wake up from a poll than from a futex
- The work done before and after using poll is also costly compared to the use of a futex

Enhanced I/O scalability

new, second attempt

Enhanced I/O scalability
new , second attempt

- Introduce separate IO thread(s) in addition to the scheduler threads
- IO threads doing all the wait in epoll_wait
- Check_io changed to work with only one kernel poll set (synchronization handled by Linux kernel)
- the poll_set updated with epoll_ctl directly to the OS-kernel (Linux)
- possibly multiple IO thread(s) calls epoll_waity with the poll_set
- Scheduler threads wait on a futex to be waken up by an IO thread or other scheduler

Enhanced I/O scalability
new , second attempt

- Pros
  - This reduces the contention and parallel IO performance is very good
  - **All schedulers wait on a futex which is good**
- Cons
  - A single IO thread might become a bottle neck, but the solution can be expanded to having several IO threads.
  - Update of the single pollset via kernel might cause contention, but can be worked around by having several poll sets.

Enhanced I/O scalability plan for introduction

- Unfortunately we did not make it for OTP 20
- Put on master when OTP 20 is released
- Possibly Introduce as experimental with alternative build via configure in OTP 20.X or as a separate branch based on OTP 20

Constant data not copied

- What is “literal” data
- Compiler handles “literal” data
- Internal representation of constant data
- Not copied in internal messages
- Still copied to ets tables
- when code is purged the literals should be deleted!!!!!!

Constant data not copied example

lit.erl

```
-module(lit).
-compile(export_all).

value1() ->
    {“abc”, ”xyz”, “fge”}.

value2(A,B,C) ->
    {A, B, C}.
```

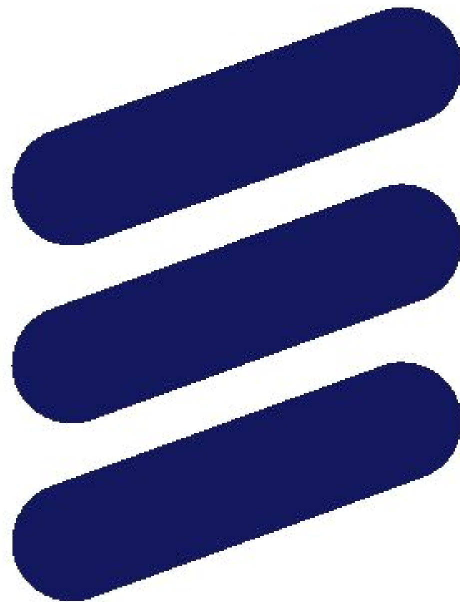
```
> erlc -S lit.erl

# lit.S is created
```

lit.S

```
{function, value1, 0, 2}.
  {label,1}.
    {line,[{location,"lit.erl",5}]}.
    {func_info,{atom,lit},{atom,value1},0}.
  {label,2}.
    {move,{literal,{"abc","xyz","fge"}},{x,0}}.
    return.

{function, value2, 3, 4}.
  {label,3}.
    {line,[{location,"lit.erl",7}]}.
    {func_info,{atom,lit},{atom,value2},3}.
  {label,4}.
    {test_heap,4,3}.
    {put_tuple,3,{x,3}}.
    {put,{x,0}}.
    {put,{x,1}}.
    {put,{x,2}}.
    {move,{x,3},{x,0}}.
    return.
```



ERICSSON