Erlang / Elixir Meetup Zürich


NOMOKO


Bare
Metal
Performance

Integration with non-BEAM code

Source: https://potatosalad.io/2017/08/05/latency-of-native-functions-for-erlang-and-elixir

Type	Isolation	Complexity	Latency
Node	Network	Highest	Highest
Port	Process	High	High
Port Driver	Shared	Low	Low
NIF	Shared	Lowest	Lowest

Learn more at: http://erlang.org/doc/tutorial/introduction.html

JASON: An example of fast code on the VM

A JSON parsing library by Michał Muskała

Hex: https://hex.pm/packages/jason
Github: https://github.com/michalmuskala/jason

Protocol based, focused on speed rather than rich feature set
Pretty is not a concern; speed is.

JASON: An example of fast code on the VM

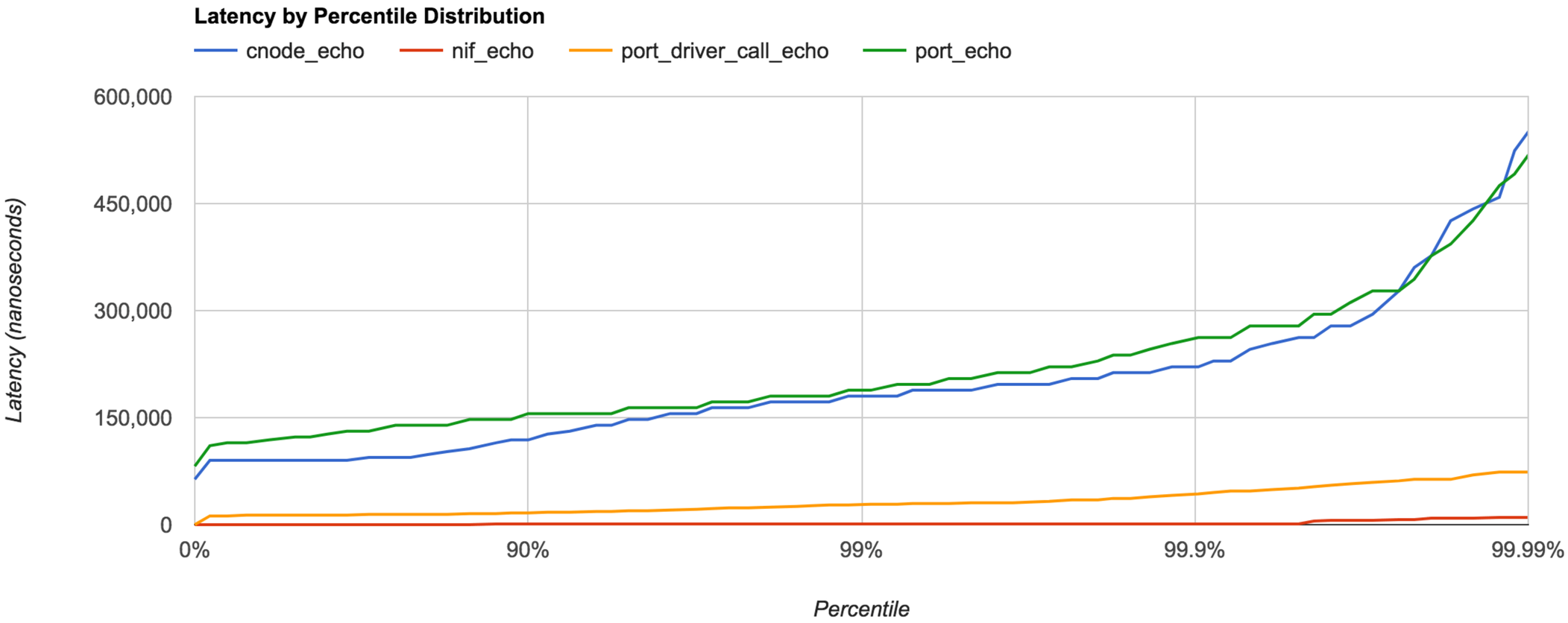
```
defp array(data, original, skip, stack, key_decode, string_decode, value) do
  bytecase data do
    _ in '\s\n\t\r', rest →
      array(rest, original, skip + 1, stack, key_decode, string_decode, value)
    _ in ']', rest →
      [acc | stack] = stack
      value = :lists.reverse(acc, [value])
      continue(rest, original, skip + 1, stack, key_decode, string_decode, value)
    _ in ',', rest →
      [acc | stack] = stack
      value(rest, original, skip + 1, [@array, [value | acc] | stack], key_decode, string_decode)
    _, _rest →
      error(original, skip)
    <<_::bits>> →
      empty_error(original, skip)
  end
end

@compile {:inline, object: 6}

defp object(rest, original, skip, stack, key_decode, string_decode) do
  key(rest, original, skip, [[] | stack], key_decode, string_decode)
end
```

Integration with non-BEAM code

Source: https://potatosalad.io/2017/08/05/latency-of-native-functions-for-erlang-and-elixir



Calling External Processes: Ports

Provide an easy way to start an external native process and connect stdin / stdout

- Looks like a process from the BEAM code side

Easy-peasy:

```
port = Port.open({:spawn, cmd}, [:binary])
Port.command(port, msg)
```

Responses delivered using messages: just use a receive block!

```
{port, {:data, result}}
```

Various libraries on hex.pm that make ports even nicer to use

- Some specific to languages e.g. python, ruby

Natively compiled plugins: Port Drivers

Used for I/O outside of the BEAM

- Historically, anyways: file I/O is moving to BIFs (speeeeed!)

Requires writing a driver in C using the moderately complex port driver API

Run in-process

- Good latency
- Crashes bring down your VM
- Still incurs the overhead of message passing
- Serialization overhead for terms, but usually negligible in the bigger scheme
- Follows the familiar OTP mechanism of processes, so nice for async use

Natively compiled plugins: Port Drivers

```
static ErlDrvEntry latency_driver_entry = {
    latency_drv_init,               /* F_PTR init, called at system startup for statically linked drivers, and after loading for
                                                   dynamically loaded drivers */
    latency_drv_start,              /* F_PTR start, called when open_port/2 is invoked, return value -1 means failure */
    latency_drv_stop,               /* F_PTR stop, called when port is closed, and when the emulator is halted */
    latency_drv_output,             /* F_PTR output, called when we have output from Erlang to the port */
    NULL,                           /* F_PTR ready_input, called when we have input from one of the driver's handles */
    NULL,                           /* F_PTR ready_output, called when output is possible to one of the driver's handles */
    "latency_drv",                  /* char *driver_name, name supplied as command in erlang:open_port/2 */
    latency_drv_finish,             /* F_PTR finish, called before unloading the driver - dynamic drivers only */
    NULL,                           /* void *handle, reserved, used by emulator internally */
    latency_drv_control,            /* F_PTR control, "ioctl" for drivers - invoked by port_control/3 */
    NULL,                           /* F_PTR timeout, handling of time-out in driver */
    latency_drv_outputv,            /* F_PTR outputv, called when we have output from Erlang to the port */
    NULL,                           /* F_PTR ready_async, only for async drivers */
    NULL,                           /* F_PTR flush, called when the port is about to be closed, and there is data in the driver
                                                    queue that must be flushed before 'stop' can be called */
    latency_drv_call,               /* F_PTR call, works mostly like 'control', a synchronous call into the driver */
    NULL,                           /* F_PTR event, called when an event selected by driver_event() has occurred */
    ERL_DRV_EXTENDED_MARKER,        /* int extended marker, should always be set to indicate driver versioning */
    ERL_DRV_EXTENDED_MAJOR_VERSION, /* int major_version, should always be set to this value */
    ERL_DRV_EXTENDED_MINOR_VERSION, /* int minor_version, should always be set to this value */
    LATENCY_DRV_FLAGS,              /* int driver_flags, see documentation */
    NULL,                           /* void *handle2, reserved, used by emulator internally */
    NULL,                           /* F_PTR process_exit, called when a process monitor fires */
    NULL                            /* F_PTR stop_select, called to close an event object */
};
```

Natively compiled plugins: Port Drivers

```
static ErlDrvSSizeT
latency_drv_call(ErlDrvData drv_data,
                 unsigned int command,
                 char *buf, ErlDrvSizeT len,
                 char **rbuf,
                 ErlDrvSizeT rlen,
                 unsigned int *flags)
{
    if (rlen < len) {
        *rbuf = (void *)driver_alloc(len);
    }
    (void)memcpy(*rbuf, buf, len);
    return (ErlDrvSSizeT)(len);
}
```

Native Instruction Functions: NIFs

Pathway for plugins and bindings for the BEAM

Simplest, fastest … least safe

Natively compiled, run in-process, have no process or other OTP integration

Newest of the mechanisms, and looks a lot like many language's bindings mechanisms

Functions get an environment object and an argc/argv pair, return ERL_NIF_TERMs

```
static ERL_NIF_TERM
latency_nif_echo_1(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[])
{
    return argv[0];
}
```

Native Instruction Functions: NIFs

Pathway for plugins and bindings for the BEAM

Simplest, fastest … least safe

Natively compiled, run in-process, have no process or other OTP integration

Newest of the mechanisms, and looks a lot like many language's bindings mechanisms

Functions get an environment object and an argc/argv pair, return ERL_NIF_TERMs

```
static ERL_NIF_TERM
latency_nif_echo_1(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[])
{
    return argv[0];
}
```

Native Instruction Functions: NIFs

Loaded with `erlang:load_nif/2`

Binds the calls in the library to erlang functions named the same:

```
echo(_Term) -> erlang:nif_error({nif_not_loaded, ?MODULE}).
```

Then just call them as you would any any other function!

```
latency_nif:echo(<<"foo">>).
```

Yielding and Dirty NIFs

A NIF running in a normal BEAM scheduler thread that takes more than ~1ms causes chaos

These are called “dirty” NIFs, and there are now schedulers for them

- CPU bound
- I/O bound

Use the `enif_schedule_nif` function

- `ERL_NIF_DIRTY_JOB_CPU_BOUND / ERL_NIF_DIRTY_JOB_IO_BOUND`

```
ERL_NIF_TERM enif_schedule_nif(ErlNifEnv* env, const char* fun_name, int flags,
ERL_NIF_TERM (*fp)(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]), int argc,
const ERL_NIF_TERM argv[])
```

Dirty NIFs

```
static ERL_NIF_TERM
latency_nif_dirty_cpu_echo_1(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[])
{
    return enif_schedule_nif(env, "dirty_cpu_echo",
                             ERL_NIF_DIRTY_JOB_CPU_BOUND,
                             latency_nif_echo_1, argc, argv);
}
```

C++ NIFs with nifpp

Single-header library → https://github.com/goertzenator/nifpp

- Overloaded get()/make() wrappers for the enif_get_xxx()/enif_make_xxx() C API.
- get()/make() support for STL containers tuple, vector, array, list, deque, set, unordered_set, multiset, map, and unordered_map.
- get()/make() support for nested containers.
- A resource pointer type so that any type can be easily used as a NIF resource. Think of it as a std::shared_ptr that the emulator can hold references to.

C++ NIFs with nifpp

```
template<typename TK, typename TV>
nifpp::TERM mapflip_test(ErlNifEnv* env, ERL_NIF_TERM term)
{
    std::map<TK,TV> inmap;
    std::map<TV,TK> outmap;
    get_throws(env, term, inmap);
    for (auto i = inmap.begin(); i =/= inmap.end(); i++)
    {
        outmap[i->second] = i->first;
    }
    return make(env, outmap);
}
```

Protecting the home base: Port + NIF?

What if you have a number of NIFs that you want to run in more complex BEAM code …

… and you don’t want to risk the stability of your VM?

SafeExecEnv to the rescue: starts an external BEAM VM linked to the spawning BEAM in which you can run functions of arbitrary complexity … GenServers, message passing, whatever

SafeExecEnv

```
defmodule MyApp do
  @moduledoc false

  use Application

  def start(_type, _args) do
    children = [SafeExecEnv]
    opts = [strategy: :one_for_one, name: MyApp.Supervisor]
    Supervisor.start_link(children, opts)
  end
End

SafeExecEnv.exec(fn → 2 * 3 end)
SafeExecEnv.exec(module, func, args)

# that's it :)
```

SafeExecEnv

Available on Hex.pm at https://hex.pm/packages/safe_exec_env

Works with releases, unlike OTP's slave module

Feature ideas:

- Multiple external nodes?
- Per-call nodes?
- Async conveniences?


That's all Folks!