Erlang NIFs:
Native Implemented Functions

Derek Brown

January 20, 2015

Materials

- Code examples and slides:

  https://github.com/derek121/nif_demo

Topics

- Benefits and dangers
- The minimal, “Hello, Joe” NIF
- Handling basic data structures
- Memory management
- Scheduler issues
- Dirty schedulers

Benefits

- Introduced in R13B03 (November 2009)
- Execute C code in-process with the VM
- Speed
- Easy way to use existing C libraries

Dangers/Issues

- Runs in-process with the VM
- Requires knowledge of/coding in C

A Minimal NIF

Erlang:

- Call to erlang:load_nif/2
- Stub (or alternate) function in the Erlang module

A Minimal NIF

Erlang:

- Call to erlang:load_nif/2
- Stub (or alternate) function in the Erlang module

C:

- Declare the NIF functions
- Implementation with proper signature

Erlang - Load

```
-on_load(init/0).

init() ->
  ok = erlang:load_nif("priv/demo_nif", 0).
```

Erlang - Stub

```
-on_load(init/0).

init() ->
  ok = erlang:load_nif("priv/demo_nif", 0).

hello() ->
  erlang:nif_error(nif_not_loaded).
  %% Or an alternate implementation in
  %% Erlang
```

C - Init

```
#include "erl_nif.h"

ERL_NIF_INIT(demo, nif_funcs,
  load, reload, upgrade, unload)
```

load, reload (*deprecated)*, upgrade, unload:

Pointers to to functions called on those events. May be NULL.

C - Init

```
#include "erl_nif.h"

ERL_NIF_INIT(demo, nif_funcs,
  load, reload, upgrade, unload)

static ErlNifFunc nif_funcs[] = {
  {“hello", 0, hello_nif}
}
```

C - Implementation

```
static ERL_NIF_TERM hello_nif(
  ErlNifEnv* env,
  int argc, const ERL_NIF_TERM argv[])
{
  return enif_make_tuple2(
    env,
    enif_make_atom(env, "ok"),
    enif_make_atom(env, "joe"));
}
```

Build and Run

```
$ rebar compile
```

- Compiles c_src/ .c to priv/
- `port_specs` in rebar.config to define source-to-.so mappings
- `port_env` allows specifying compile/link flags

```
$ rebar shell
```

- Adds ebin/ and deps/*/ebin/ to codepath

Demo Example

https://github.com/derek121/nif_demo

- “minimal” example

Returning Values – Ints, Atoms, Strings

```
ERL_NIF_TERM t;
t = enif_make_int(env, 24);

t = enif_make_atom(env, "hello");

t = enif_make_string(env, "hello”,
  ERL_NIF_LATIN1);
```

Returning Values – Tuples

```
ERL_NIF_TERM enif_make_tuple(ErlNifEnv* env, unsigned cnt, ...)

ERL_NIF_TERM enif_make_tuple1(ErlNifEnv* env, ERL_NIF_TERM e1)
ERL_NIF_TERM enif_make_tuple2(ErlNifEnv* env, ERL_NIF_TERM e1,
ERL_NIF_TERM e2)
ERL_NIF_TERM enif_make_tuple3(ErlNifEnv* env, ERL_NIF_TERM e1,
ERL_NIF_TERM e2, ERL_NIF_TERM e3)
ERL_NIF_TERM enif_make_tuple4(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e4)
ERL_NIF_TERM enif_make_tuple5(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e5)
ERL_NIF_TERM enif_make_tuple6(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e6)
ERL_NIF_TERM enif_make_tuple7(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e7)
ERL_NIF_TERM enif_make_tuple8(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e8)
ERL_NIF_TERM enif_make_tuple9(ErlNifEnv* env, ERL_NIF_TERM e1, ...,
ERL_NIF_TERM e9)
```

Returning Values - Lists

```
ERL_NIF_TERM i1 = enif_make_int(env, 26);
ERL_NIF_TERM i2 = enif_make_int(env, 36);
ERL_NIF_TERM i3 = enif_make_int(env, 46);

ERL_NIF_TERM list = enif_make_list3(env,
i1, i2, i3);
```

Receiving Values- Numeric

```
int val;
enif_get_int(env, argv[0], &val);
```

Receiving Values- Atoms

```
unsigned int len;
enif_get_atom_length(env, argv[0], &len,
  ERL_NIF_LATIN1))
char *buf = (char *)enif_alloc(len + 1);
```

Receiving Values- Atoms

```
unsigned int len;
enif_get_atom_length(env, argv[0], &len,
  ERL_NIF_LATIN1))
char *buf = (char *)enif_alloc(len + 1);

enif_get_atom(env, argv[0], buf, len + 1,
  ERL_NIF_LATIN1);
printf("%s\r\n", buf);

enif_free(buf);
```

Receiving Values- Strings

```
unsigned int len;
enif_get_list_length(env, argv[0], &len);
char *buf = (char *)enif_alloc(len + 1);
```

Receiving Values- Strings

```
unsigned int len;
enif_get_list_length(env, argv[0], &len);
char *buf = (char *)enif_alloc(len + 1);

enif_get_string(env, argv[0], buf, len + 1,
  ERL_NIF_LATIN1);
printf("%s\r\n", buf);

enif_free(buf);
```

Receiving Values- Tuples

```
const ERL_NIF_TERM *tuple;
int arity;
enif_get_tuple(env, argv[0], &arity,
  &tuple);
```

Receiving Values- Tuples

```
const ERL_NIF_TERM *tuple;
int arity;
enif_get_tuple(env, argv[0], &arity,
  &tuple);

for (int i = 0; i < arity; i++) {
  int val;
  enif_get_int(env, tuple[i], &val);
  printf("%d: %d\r\n", i, val);
}
```

Receiving Values- Lists

```
ERL_NIF_TERM head, tail;
int val;
ERL_NIF_TERM list = argv[0];

while (enif_get_list_cell(env, list, &head,
    &tail)) {
  enif_get_int(env, head, &val);
  printf("%d\r\n", val);
  list = tail;
}
```

Receiving Values- Lists

```
ERL_NIF_TERM head, tail;
int val;
ERL_NIF_TERM list = argv[0];

while (enif_get_list_cell(env, list, &head,
    &tail)) {
  enif_get_int(env, head, &val);
  printf("%d\r\n", val);
  list = tail;
}
```

Receiving Values - Lists

```
ERL_NIF_TERM head, tail;
int val;
ERL_NIF_TERM list = argv[0];

while (enif_get_list_cell(env, list, &head,
    &tail)) {
  enif_get_int(env, head, &val);
  printf("%d\r\n", val);
  list = tail;
}
```

Memory Management

- Manually create and destroy
  - E.g., `enif_alloc()`
- Resources
  - Released automatically

Resources

- An opaque pointer passed between C and Erlang
- Deallocated after call to release and all referencing terms have been garbage collected

Resources

- Specified via `init()`

```
ERL_NIF_INIT(demo, nif_funcs,
  load, NULL, NULL, NULL)

int (*load)(
  ErlNifEnv* env,
  void** priv_data,
  ERL_NIF_TERM load_info)
```

Resources - Definition

```
static int load(ErlNifEnv* env, void** priv_data,
ERL_NIF_TERM load_info)
{
  ErlNifResourceType *resource_type =
    enif_open_resource_type(
      env,
      NULL, /* module_str: must be NULL */
      "resource",
      resource_dtor,
      (ErlNifResourceFlags)(ERL_NIF_RT_CREATE),
      NULL /* Used on failure */
  );

  *priv_data = resource_type;
  return 0;
}
```

Resources - Allocating

- `ERL_NIF_TERM enif_make_resource(ErlNifEnv*env, void* obj)`
  - Allocates
- `void enif_release_resource(void* obj)`
  - If called before return, resource will be deallocated once unreferenced on Erlang side
  - Else, may be kept on NIF side to be released later

Resources - Allocating

```
static ERL_NIF_TERM return_resource_nif(
  ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
  ErlNifResourceType* resource_type =
(ErlNifResourceType*)enif_priv_data(env);

  Vals *vals =
    (Vals *)enif_alloc_resource(
      resource_type, sizeof(Vals));
  vals->val1 = 23;
  vals->val2 = 25;

  ERL_NIF_TERM result = enif_make_resource(env, vals);
  enif_release_resource(vals);
  return result;
}
```

Resources - Destructors

- Called when an instance of given resource type is released
- Allows manual cleanup of allocated memory, etc.

```
typedef void ErlNifResourceDtor(
   ErlNifEnv* env, void* obj);
```

Example

- erlsha2

  https://github.com/vinoski/erlsha2.git

Scheduling

- Scheduler per core
- Processes are given a fixed number of reductions before being preempted (2000)
  - E.g., function calls, sending messages, etc.
- NIF operations are not included
- NIFs therefore need to complete within ~1 ms

NIF Scheduling

- `enif_schedule_nif/6`
  - Manually call with smaller tasks
- `enif_consume_timeslice/2`
  - Give scheduler hints on progress
- Example: `bitwise:exor_yield/2`

Dirty Schedulers

- Separated to avoid interfering with regular process scheduling
- Two types: CPU-bound and IO-bound
  - Number of CPU-bound dirty schedulers =< number of normal schedulers
  - Number of IO-bound dirty schedulers not limited
- Set via command line flags to `erl` and via `erlang:system_flag/2`

Run Queues

- Normal schedulers (one per core)
  - Separate run queues for each
- CPU-bound schedulers
  - Share a run queue
- IO-bound schedulers
  - Share a run queue

NIF Dirty Scheduling – Dynamic

- Pass flags to `enif_schedule_nif/6`
  - `ERL_NIF_DIRTY_JOB_CPU_BOUND`
  - `ERL_NIF_DIRTY_JOB_IO_BOUND`
- Allows decision on dirtiness or not to be made per-call

NIF Dirty Scheduling – Static

- Specify in the ErlNifFunc array entry
  - ERL_NIF_DIRTY_JOB_CPU_BOUND
  - ERL_NIF_DIRTY_JOB_IO_BOUND
- E.g.:

```
static ErlNifFunc funcs[] = {
  {"exor_dirty", 2, exor,
   ERL_NIF_DIRTY_JOB_CPU_BOUND}};
```

Long Scheduling

- Receive notification when regular scheduler thread runs past threshhold

```
1> erlang:system_monitor(self(),
     [{long_schedule, 10}]).
{<0.38.0>,[{long_schedule,10}]}
2> spawn(fun() -> demo:delay(1) end).
<0.66.0>
3> flush().
Shell got {monitor,<0.66.0>,long_schedule,
                    [{timeout,1001},
{in,undefined},{out,undefined}]}
ok
```

Example

- bitwise

  https://github.com/vinoski/bitwise

References

API:
http://www.erlang.org/doc/tutorial/nif.html
http://www.erlang.org/doc/man/erl_nif.html

erlsha2:
https://github.com/vinoski/erlsha2

bitwise:
https://github.com/vinoski/bitwise

Steve Vinoski 2014 Codemesh talk on schedulers
http://www.codemesh.io/codemesh2014/steve-vinoski

Thanks

Thanks to Steve Vinoski for details on scheduling and dirty schedulers, and for the creation of the erlsha2 and bitwise projects.

Questions?