Useless performance optimisations on the BEAM

for fun and… fun?

WAT

The “ref trick” doesn’t always work

```
do_call(Process, Label, Request, Timeout) ->
 Mref = erlang:monitor(process, Process),
 erlang:send(Process, {Label, {self(), Mref}, Request}, [noconnect]),
 receive
   {Mref, Reply} ->
     erlang:demonitor(Mref, [flush]),
     {ok, Reply};
   {'DOWN', Mref, _, _, noconnection} ->
     Node = get_node(Process),
     exit({nodedown, Node});
   {'DOWN', Mref, _, _, Reason} ->
     exit(Reason)
 after Timeout ->
     erlang:demonitor(Mref, [flush]),
     exit(timeout)
 end.
```

```
wait_for_two_servers(Ref, Pid1, Pid2) ->
 Mon1 = monitor(process, Pid1),
 Mon2 = monitor(process, Pid2),
 receive
   {ok, Ref, Res} ->
     demonitor(Pid1, [flush]),
     demonitor(Pid2, [flush]),
     {ok, Res};
   {'DOWN', Mon1, _, _, _} ->
     demonitor(Mon2, [flush]),
     error;
   {'DOWN', Mon2, _, _, _} ->
     demonitor(Mon1, [flush]),
     error
 end.
```

OTP-14505 (from OTP 21.0 Release Notes)

In code such as `example({ok, Val}) -> {ok, Val}.` a tuple would be built. The compiler will now automatically rewrite the code to `example({ok,Val}=Tuple) -> Tuple.` which will reduce code size, execution time, and remove GC pressure.

Can a human beat the compiler?

About me

Dániel Szoboszlay

- github.com/dszoboszlay
- Klarna.

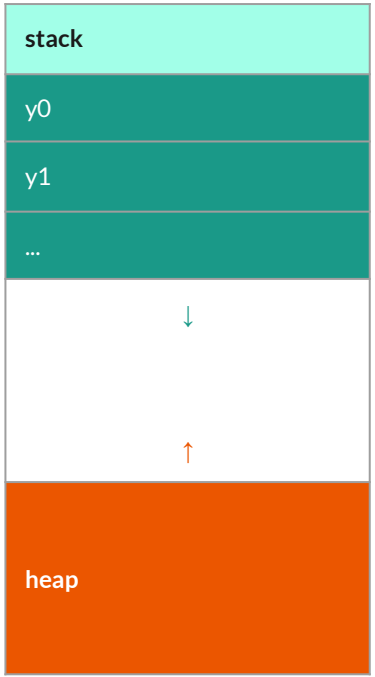
I couldn't give this talk without:

- Björn Gustavsson's posts about SSA on blog.erlang.org
- Happi's The BEAM Book

Useless performance optimisations on the BEAM

The BEAM VM

- A register machine
  - Registers (x0, x1, …): storing 1 word each
  - Stack (y0, y1, … & CP): storing 1 word each
  - Heap: for boxed data types
- Data types
  - Immediates: size = 1 word
    (small integers, atoms, pids…)
  - Boxed types: size > 1 word
    (list cells, tuples, maps, big integers…)



The BEAM VM

Call conventions

- Arguments are in x0, x1, ...
- Return value is in x0
- All other registers are destroyed by the call (save the data to the stack)
- BIFs don't follow these call conventions
  - Use any register for args and return value
  - Won't destroy other registers
  - No reduction counting, no tracing

Keep track the number of live registers for the GC

- Extra argument for ops that may trigger GC
  - Heap/stack allocation
  - Function calls (may be scheduled out)
  - BIF-s that may allocate heap
  - BIF-s that are interruptible
  - Message sending

The instruction set

- 168 opcodes
- It's Erlang all the way down
- Most common ones (OTP 22.0 rc1, Mnesia, 78 different opcodes, 62k instructions in total):
  - move: 34.2%
  - call_ext: 6.5%
  - get_tuple_element: 5.7%
  - put_tuple2: 4.6%
  - test_heap: 4.3%

Hello World in BEAM assembly

```
hello_world() ->
 io:put_chars(<<"hello world!\n">>).
```

```
{function, hello_world, 0, 2}.
 {label,1}.
   {func_info,{atom,demo},{atom,hello_world},0}.
 {label,2}.
   {move,{literal,<<"hello world!\n">>},{x,0}}.
   {call_ext_only,1,{extfunc,io,put_chars,1}}.
```

- Sequence of Erlang terms
- Very verbose format.
- function: name, arity and entry point
- label: target for jumps
- call_ext_only: tail-recursive external call
- func_info: dual purpose instruction
  - Generate a function clause error
  - Must stand before the entry point of the function, so local function calls can be deciphered

Useless performance optimisations on the BEAM

Case study: adding the ref trick

Erlc's version

```
reftrick(Pid1, Pid2) ->
 Mon1 = monitor(process, Pid1),
 Mon2 = monitor(process, Pid2),
 Pid1 ! {request, Mon1, self()},
 Pid2 ! {request, Mon1, self()},
 receive
   {response, Mon1, Val} ->
     {ok, Val};
   {'DOWN', Mon1, process, Pid1, Reason} ->
     {error, Reason};
   {'DOWN', Mon2, process, Pid2, Reason} ->
     {error, Reason}
 end.
```

```
{function, reftrick, 2, 2}.
 {label,1}.
   {func_info,{atom,demo},{atom,reftrick},2}.
 {label,2}.
   {allocate_zero,4,2}.
   {move,{x,1},{y,2}}.
   {move,{x,0},{y,3}}.
   {move,{x,0},{x,1}}.
   {move,{atom,process},{x,0}}.
   {call_ext,2,{extfunc,erlang,monitor,2}}.
   {move,{x,0},{y,1}}.
   {move,{y,2},{x,1}}.
   {move,{atom,process},{x,0}}.
   {call_ext,2,{extfunc,erlang,monitor,2}}.
   {test_heap,4,1}.
   {bif,self,{f,0},[],{x,1}}.
   {put_tuple2,{x,1},{list,[{atom,request},{y,1},{x,1}]}}.
   {move,{x,0},{y,0}}.
   {move,{y,3},{x,0}}.
   send.
   {test_heap,4,0}.
   {bif,self,{f,0},[],{x,0}}.
   {put_tuple2,{x,1},{list,[{atom,request},{y,1},{x,0}]}}.
   {move,{y,2},{x,0}}.
   send.
```

Erlc's version

```
reftrick(Pid1, Pid2) ->
 Mon1 = monitor(process, Pid1),
 Mon2 = monitor(process, Pid2),
 Pid1 ! {request, Mon1, self()},
 Pid2 ! {request, Mon1, self()},
 receive
   {response, Mon1, Val} ->
     {ok, Val};
   {'DOWN', Mon1, process, Pid1, Reason} ->
     {error, Reason};
   {'DOWN', Mon2, process, Pid2, Reason} ->
     {error, Reason}
 end.
```

```
{label,3}.
  {loop_rec,{f,9},{x,0}}.
  {test,is_tuple,{f,8},[{x,0}]}.
  {select_tuple_arity,{x,0},{f,8},{list,[5,{f,5},3,{f,4}]}}.
{label,4}.
  {get_tuple_element,{x,0},0,{x,1}}.
  {get_tuple_element,{x,0},1,{x,2}}.
  {test,is_eq_exact,{f,8},[{x,1},{atom,response}]}.
  {test,is_eq_exact,{f,8},[{x,2},{y,1}]}.
  {test_heap,3,1}.
  {get_tuple_element,{x,0},2,{x,0}}.
  remove_message.
  {put_tuple2,{x,0},{list,[{atom,ok},{x,0}]}}.
  {deallocate,4}.
  return.
{label,5}.
  {get_tuple_element,{x,0},0,{x,1}}.
  {get_tuple_element,{x,0},1,{x,2}}.
  {get_tuple_element,{x,0},2,{x,3}}.
  {get_tuple_element,{x,0},3,{x,4}}.
  {get_tuple_element,{x,0},4,{x,0}}.
  {test,is_eq_exact,{f,8},[{x,1},{atom,'DOWN'}]}.
  {test,is_eq_exact,{f,8},[{x,3},{atom,process}]}.
  {test,is_eq_exact,{f,6},[{x,2},{y,1}]}.
  {test,is_ne_exact,{f,7},[{x,4},{y,3}]}.
{label,6}.
  {test,is_eq_exact,{f,8},[{x,2},{y,0}]}.
  {test,is_eq_exact,{f,8},[{x,4},{y,2}]}.
{label,7}.
  {test_heap,3,1}.
  remove_message.
  {put_tuple2,{x,0},{list,[{atom,error},{x,0}]}}.
  {deallocate,4}.
  return.
{label,8}.
  {loop_rec_end,{f,3}}.
{label,9}.
  {wait,{f,3}}.
```

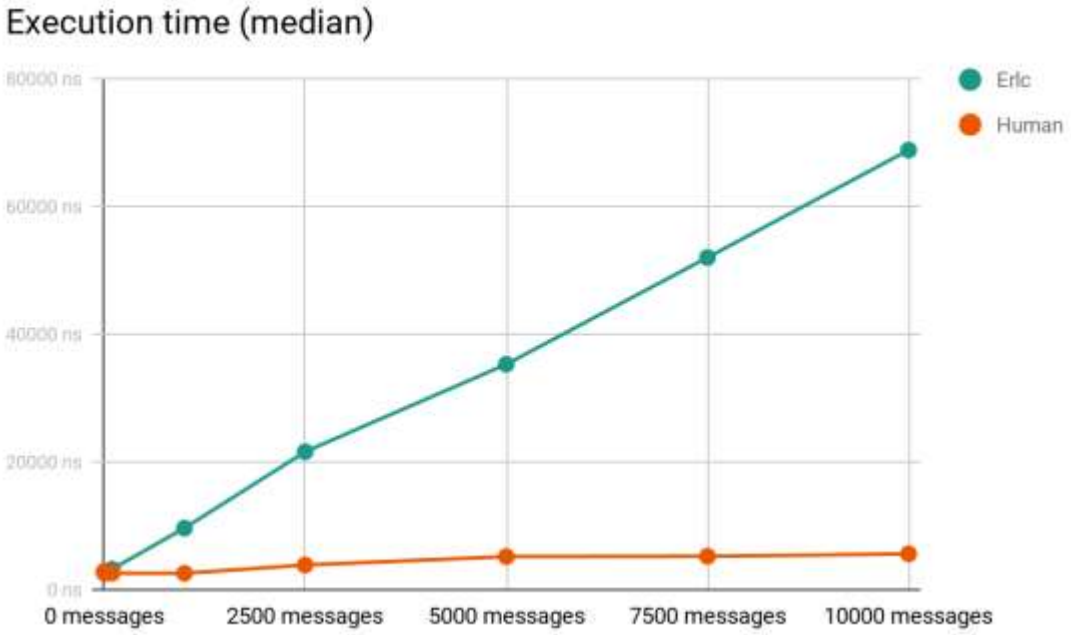
Manual optimisation

```
  {move,{x,0},{x,1}}.
  {move,{atom,process},{x,0}}.
  {recv_mark,{f,3}}.
  {call_ext,2,{extfunc,erlang,monitor,2}}.
  %% ...
  {recv_set,{f,3}}.
{label,3}.
  {loop_rec,{f,9},{x,0}}.
```

```
  {test_heap,4,1}.
  {bif,self,{f,0},[],{x,1}}.
  {put_tuple2,{x,1},{list,[{atom,request},{y,1},{x,1}]}}.
  {move,{x,0},{y,0}}.
  {move,{y,3},{x,0}}.
  send.
  {test_heap,4,0}.
  {bif,self,{f,0},[],{x,0}}.
  {put_tuple2,{x,1},{list,[{atom,request},{y,1},{x,0}]}}.
  {move,{y,2},{x,0}}.
  send.
```

```
{label,5}.
  {get_tuple_element,{x,0},0,{x,1}}.
  {get_tuple_element,{x,0},1,{x,2}}.
  {get_tuple_element,{x,0},2,{x,3}}.
  {get_tuple_element,{x,0},3,{x,4}}.
  {get_tuple_element,{x,0},4,{x,0}}.
  {test,is_eq_exact,{f,8},[{x,1},{atom,'DOWN'}]}.
  {test,is_eq_exact,{f,8},[{x,3},{atom,process}]}.
  {test,is_eq_exact,{f,6},[{x,2},{y,1}]}. %% =:= Pid1
  {test,is_ne_exact,{f,7},[{x,4},{y,3}]}. %% =/= Mon1
{label,6}.
  {test,is_eq_exact,{f,8},[{x,2},{y,0}]}. %% =:= Pid2
  {test,is_eq_exact,{f,8},[{x,4},{y,2}]}. %% =:= Mon2
```

Results


Execution time (median)
80000 ns
60000 ns
40000 ns
20000 ns
0 ns
Erlc
Human
0 messages
2500 messages
5000 messages
7500 messages
10000 messages

Case study: redundant tests on recursion

Erlc's version

```
gcd(A, B) when A < B ->
 gcd(B, A);
gcd(A, 0) when A >= 0 ->
 A;
gcd(A, B) when A >= 0, B >= 0 ->
 gcd(B, A rem B).
```

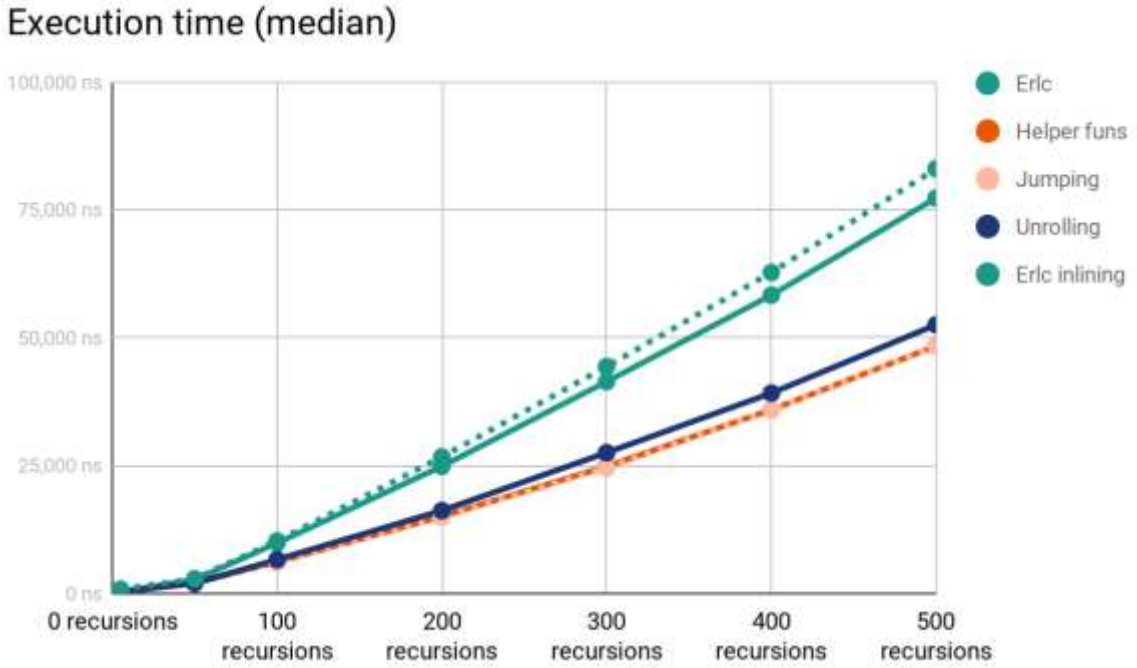
```
{function, gcd, 2, 2}.
 {label,1}.
   {func_info,{atom,demo},{atom,gcd},2}.
 {label,2}.
   {test,is_lt,{f,3},[{x,0},{x,1}]}.
   {move,{x,1},{x,2}}.
   {move,{x,0},{x,1}}.
   {move,{x,2},{x,0}}.
   {call_only,2,{f,2}}.
 {label,3}.
   {test,is_eq_exact,{f,4},[{x,1},{integer,0}]}.
   {test,is_ge,{f,4},[{x,0},{integer,0}]}.
   return.
 {label,4}.
   {test,is_ge,{f,1},[{x,0},{integer,0}]}.
   {test,is_ge,{f,1},[{x,1},{integer,0}]}.
   {gc_bif,'rem',{f,0},2,[{x,0},{x,1}],{x,0}}.
   {move,{x,1},{x,2}}.
   {move,{x,0},{x,1}}.
   {move,{x,2},{x,0}}.
   {call_only,2,{f,2}}.
```

Manual optimisations

- Example on right won't compile
- Workarounds:
  - Split it into multiple functions
  - Use `jump` instead of call_only
  - Use loop (recursion) unrolling
    (e.g. 10 unrolls eliminate 90% of the cost)

```
{function, gcd, 2, 2}.
 {label,1}.
   {func_info,{atom,demo},{atom,gcd},2}.
 {label,2}.
   {test,is_lt,{f,4},[{x,0},{x,1}]}.
   {move,{x,1},{x,2}}.
   {move,{x,0},{x,1}}.
   {move,{x,2},{x,0}}.
   {call_only,2,{f,4}}.
 {label,3}.
   {func_info,{atom,demo},{atom,gcd},2}.
 {label,4}.
   {test,is_ge,{f,1},[{x,0},{integer,0}]}.
   {test,is_ge,{f,1},[{x,1},{integer,0}]}.
   {jump,{f,6}}.
 {label,5}.
   {func_info,{atom,demo},{atom,gcd},2}.
 {label,6}.
   {test,is_eq_exact,{f,7},[{x,1},{integer,0}]}.
   return.
 {label,7}.
   {gc_bif,'rem',{f,0},2,[{x,0},{x,1}],{x,2}}.
   {move,{x,1},{x,0}}.
   {move,{x,2},{x,1}}.
   {call_only,2,{f,6}}.
```

Results


Execution time (median)
100,000 ns
75,000 ns
50,000 ns
25,000 ns
0 ns
0 recursions
100 recursions
200 recursions
300 recursions
400 recursions
500 recursions
Erlc
Helper funs
Jumping
Unrolling
Erlc inlining

Assorted ideas

What else to hack or optimise?

Things that didn't work out:

- Destructively updating terms
- Using custom call conventions
- Jumping backwards on OTP 21

Things to improve in the compiler:

- Detect creation of identical terms
- The same guard is evaluated on each clause
- Return value of a BIF is moved immediately
- `fun Name/Arity` defines a new wrapper
- Code factoring

Useless performance optimisations on the BEAM

Writing BEAM assembly code is impractical

- Cannot inline assembly code into Erlang modules
- Unfriendly assembler
  - Verbose syntax
  - Lots of boilerplate
  - Numeric labels
  - No error messages: erlc just crashes with a cryptic stacktrace
- Just use a NIF instead!

What are the benefits?

- Knowing the limits of your compiler may help
  - Don’t have unrealistic expectations (e.g. ref trick)
  - Adopt a coding style that can be optimised (e.g. assign terms that you plan to reuse to variables)
- Improve the compiler
  - Exciting challenge
  - There are low hanging fruits
  - Relatively flexible interface (even adding new opcodes is possible)

Contribute to the compiler!
The community needs you.

Questions?