

printf debugging, without the printf

mats cronqvist

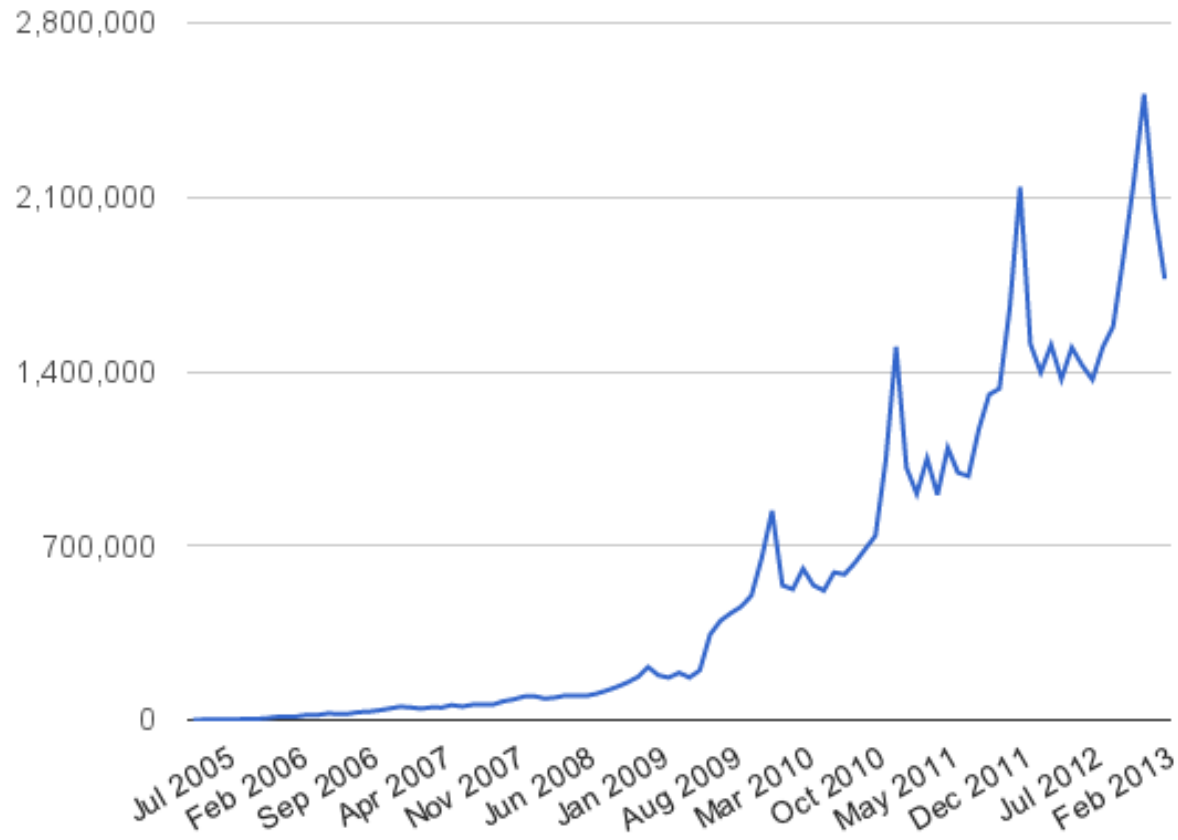
June 2014

id

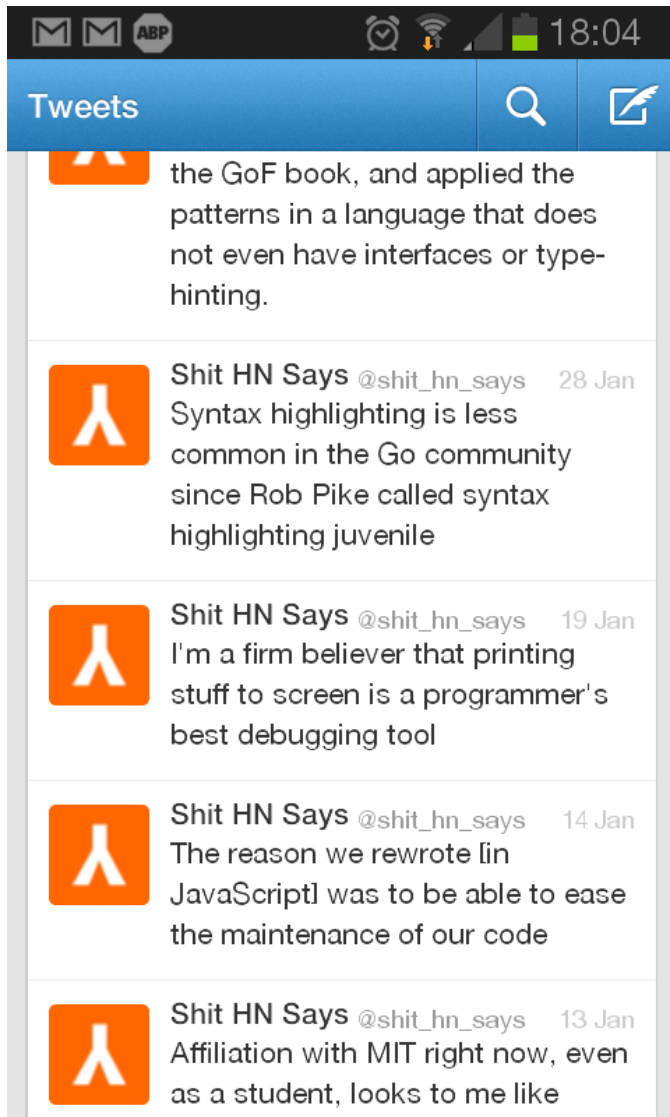
- Ph.D. in experimental Nuclear Physics
 - hardware
 - online software
 - offline software
- 10 years @ [ericsson](#)
 - building
 - testing
 - troubleshooting
 - supporting (4 years in Budapest)
- 5 years @ [klarna](#)
 - developer
 - architect



purchases per month



inspiration



"printing stuff
to screen is a
programmer's
best
debugging
tool"

sad but true

- code inspection
 - no context
- crash dumps/postmortem,
 - probably wrong context
- logging
 - swamped with trivia, wrong place
- debugger
 - changes timing, non-production only

printf it is

- the good
 - the right context
 - the right place
 - filter out uninteresting stuff
 - minimal impact on timing (not small)
- the bad
 - edit/recompile
 - tricky to undo
 - tricky in production (especially in an embedded system)

the silver bullet

- tracing
 - the right context
 - the right place
 - filter out uninteresting stuff
 - minimal impact on timing (not small)
 - no editing
 - no recompile
 - easy undo

erlang:trace/3

- Tracing is turned on/off by `erlang:trace/3`
- It is turned on per process
- Different trace types
 - message passing (send/receive)
 - garbage collection
 - scheduler
 - call tracing
- Each triggered trace condition generates
 - a message sent to the tracer process
 - writing of a trace event to file

trace BIF in action

```
29> Shell=self().
```

```
30> F = fun()->erlang:trace(Shell,true,[running]),  
             receive after 3000->  
                 Shell!process_info(self(),messages)  
             end  
end.
```

```
31> spawn(F).
```

```
32> flush().
```

```
Shell got {messages,[{trace,<0.34.0>,in,{io,execute_request,2}},  
                    {trace,<0.34.0>,out,{io,wait_io_mon_reply,2}},  
                    {trace,<0.34.0>,in,{io,wait_io_mon_reply,2}},  
                    {trace,<0.34.0>,out,{io,wait_io_mon_reply,2}},  
                    {trace,<0.34.0>,in,{io,wait_io_mon_reply,2}},  
                    {trace,<0.34.0>,out,{shell,eval_loop,3}}]}
```

but I wanted printf!

To emulate printf, we need

- a way to insert instrumentation code
- into an arbitrary function
- without recompiling

How hard could it be?

erlang:trace_pattern/3

Loads instrumentation code into a function

The target function is specified as `{M, F, A}`

Wildcards are allowed; `{lists, '_', '_'}`

The instrumentation code is a match spec.

match specs

A small, and pretty cryptic, language.

Expressed as an Erlang term.

Each match spec has

- a Head
 - like a function head in Erlang
- a Condition
 - like a guard in Erlang
- a Body
 - like a function body in Erlang

did I say cryptic?

- MatchExpression ::= [MatchFunction, ...]
- MatchFunction ::= { MatchHead, MatchConditions, MatchBody }
- MatchHead ::= MatchVariable | '_' | [MatchHeadPart, ...]
- MatchHeadPart ::= term() | MatchVariable | '_'
- MatchVariable ::= '\$<number>'
- MatchConditions ::= [MatchCondition, ...] | []
- MatchCondition ::= { GuardFunction } | { GuardFunction, ConditionExpression, ... }
- BoolFunction ::= is_atom | is_float | is_integer | is_list | is_number | is_pid | is_port | is_reference | is_tuple | is_binary | is_function | is_record | is_seq_trace | 'and' | 'or' | 'not' | 'xor' | andalso | orelse
- ConditionExpression ::= ExprMatchVariable | { GuardFunction } | { GuardFunction, ConditionExpression, ... } | TermConstruct
- ExprMatchVariable ::= MatchVariable (bound in the MatchHead) | '\$_' | '\$\$'
- TermConstruct = {{}} | {{ ConditionExpression, ... }} | [] | [ConditionExpression, ...] | NonCompositeTerm | Constant
- NonCompositeTerm ::= term() (not list or tuple)
- Constant ::= {const, term()}
- GuardFunction ::= BoolFunction | abs | element | hd | length | node | round | size | tl | trunc | '+' | '-' | '*' | 'div' | 'rem' | 'band' | 'bor' | 'bxor' | 'bnot' | 'bsl' | 'bsr' | '>' | '>=' | '<' | '<=' | '=:=' | '===' | '=/=' | '/=' | self | get_tcw
- MatchBody ::= [ActionTerm]
- ActionTerm ::= ConditionExpression | ActionCall
- ActionCall ::= {ActionFunction} | {ActionFunction, ActionTerm, ...}
- ActionFunction ::= set_seq_token | get_seq_token | message | return_trace | exception_trace | process_dump | enable_trace | disable_trace | trace | display | caller | set_tcw | silent

example, then

Match a function with **two arguments that are equal**.
The **head** of the arguments should be **less than 3**.

```
[ { [ '$1', '$1' ], [ { '<', {hd, '$1'} }, 3 ] }, [ ] ]
```

example in action

```
22> Shell = self().
```

```
23> MFA = {lists,append,'_'}. 
```

```
24> MatchSpec = [{{['$1','$1'],[{'<',{hd,'$1'},3]}],[]}] .
```

```
25> F = fun()->
```

```
    erlang:trace_pattern({'_','_','_'},false,[local]),
```

```
    erlang:trace(Shell,true,[call]),
```

```
    erlang:trace_pattern(MFA,MatchSpec,[local]),
```

```
    receive after 3000->
```

```
        Shell!process_info(self(),messages)
```

```
    end
```

```
end.
```

example works!

```
26> spawn(F).
```

```
27> lists:append([5],[5]).
```

```
28> lists:append([2],[2]).
```

```
29> flush().
```

```
Shell got {messages,[{trace,<0.34.0>,call,{lists,append,[[2],  
[2]]}}]}
```


how does it work?

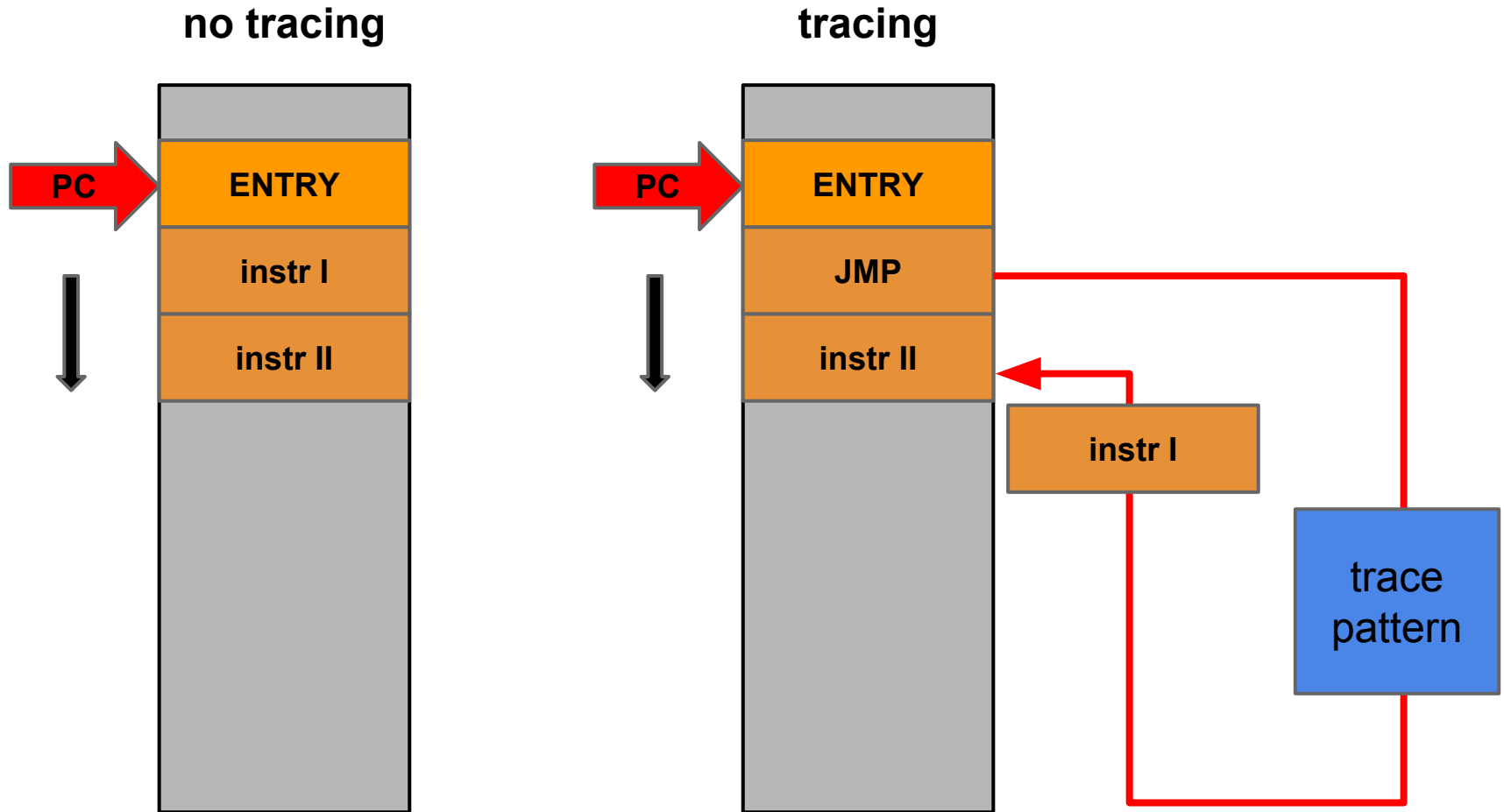
When call tracing is turned on, a JMP is inserted in place of the first instruction of the function(s) specified in the trace pattern.

At the JMP target the VM runs the MFA of the called function against the match spec, possibly binding match spec variables.

If they match, the match spec Body is run.

Before control is returned to the normal program counter, the original first instruction is run.

eh, what?



a war story

- Embedded system running OTP R5.
- Live in Denmark.
- There was no way to log into the CPU.
- There was no way to load new code.
- There was no usable debugging tool.
- You could physically connect a terminal.

The node got overloaded after 90 days.

A tech travelled there to reboot every 89 days.

...tracing...

- Wrote a one-liner...
- ...that ran a one-minute trace, wrote to a file.
- Sent it to the Danish tech by mail...
- ...who ran it by pasting it into a shell...
- ...before and after the reboot...
- ...and emailed the files to me (base-64 encoded)

...saves the day!

Wrote a comparison profiler.

Compare the average execution time for each function, before and after the reboot.

```
ets:lookup/2
```

was 100 times slower before the reboot.

the answer

The hash function was broken for bignums.

another war story



sitrep

- live network in Germany
- switch in Frankfurt was misbehaving
- millions of connected Germans
- ~10,000 ongoing calls
- no appropriate debugging support
- shell access

the oneliner

```
Pi = fun(P) when pid(P) -> case process_info(P, registered_name) of [] -> case process_info(P, initial_call) of {_,
{proc_lib,init_p,5}}-> proc_lib:translate_initial_call(P); {_,MFA} -> MFA; undefined ->unknown end; {_,Nam} -> Nam;
undefined -> unknown end; (P) whenport(P) -> {name,N} = erlang:port_info(P,name), [Hd|_] =string:tokens(N," "), Tl
=lists:reverse(hd(string:tokens(lists:reverse(Hd),"/")),list_to_atom(Tl); (R) when atom(R) -> R; ({R,Node}) when
atom(R),Node == node() -> R; ({R, Node}) when atom(R), atom(Node) -> {R,Node} end, Ts = fun(Nw) -> {_,{H,M,S}}
=calendar:now_to_local_time(Nw), {H,M,S,element(3,Nw)} end, Munge =fun(I) -> case string:str(I, "Return addr") of 0 -
> casestring:str(I, "cp = ") of 0 -> []; _ -> [_, C|_] =string:tokens(I,"()+"), list_to_atom(C) end; _ -> case
string:str(I,"erminate process normal") of 0 -> [_, C|_] =string:tokens(I,"()+"), list_to_atom(C); _ -> [] end end
end, Stack= fun(Bin) -> L = string:tokens( binary_to_list( Bin),"\\n"),{stack,list:flatten(lists:map(Munge,L))} end,
Prc = fun(all) ->all; (Pd) when pid(Pd) -> Pd; ({pid,P1,P2}) when integer(P1),integer(P2) -> c:pid(0,P1,P2); (Reg)
when atom(Reg) -> case whereis(Reg) of undefined -> exit({rdbg, no_such_process, Reg}); Pid when pid(Pid) -> Pid end
end, MsF = fun(stack, [{Head,Cond,Body}]) -> [{Head,Cond, [{message,{process_dump}}|Body]}; (return, [{Head,Cond,
Body}]) -> [{Head,Cond, [{return_trace}|Body]}; (Head, [{_,Cond,Body}]) when tuple(Head) -> [{Head,Cond,Body}]; (X,_)
->exit({rdbg,bad_match_spec,X}) end, Ms = fun(Mss) -> lists:foldl(MsF, [{'_',[[]]}, Mss) end, ChkTP = fun({M,F})
when atom(M), atom(F),M/'_' , F/'_' -> {{M,F,'_'}, [], [global]}; ({M,F,MS}) when atom(M),atom(F), M/'_' , F/'_' ->
{{M,F,'_'},Ms(MS), [global]}; ({M,F,MS,local}) when atom(M), atom(F), M/'_' , F/'_' ->{{M,F,'_'},Ms(MS), [local]};
({M,F,MS,global}) when atom(M), atom(F),M/'_' , F/'_' -> {{M,F,'_'},Ms(MS), [global]}; (X) ->exit({rdbg,
unrec_trace_pattern,X}) end, ChkTPs = fun(TPs) when list(TPs) -> lists:map(ChkTP,TPs); (TP) -> [ChkTP(TP)] end,
SetTPs =fun({MFA,MS,Fs}) -> erlang:trace_pattern(MFA,MS,Fs) end, DoInitFun =fun(Time) -> erlang:register(rdbg,
self()),erlang:start_timer(Time,self(), {die}),erlang:trace_pattern({'_','_','_'}, false,[local]),erlang:
trace_pattern({'_','_','_'},false,[global]) end, InitFun =fun(Time,all,send) -> exit({rdbg, too_many_processes});
(Time,all,'receive') -> exit({rdbg,too_many_processes}); (Time,P,send) -> DoInitFun(Time), erlang:trace(Prc(P),true,
[send,timestamp]); (Time,P,'receive') ->DoInitFun(Time), erlang:trace(Prc(P),true, ['receive', timestamp]);(Time,P,
TPs) -> CTPs = ChkTPs(TPs), DoInitFun(Time),erlang:trace(Prc(P),true, [call,timestamp]), lists:foreach(SetTPs,CTPs)
end, LoopFun = fun(G,N,Out) when N < 1 -> erlang:trace(all, false,[call,send,'receive']), erlang:trace_pattern
({'_','_','_'},false,[local]), erlang:trace_pattern( {'_','_','_'},f, false,[global]), io:fwrite("***rdbg, ~w msgs
**~n", [length(Out)]), io:fwrite("~p~n", [lists:reverse(Out)]), io:fwrite("~p~n", process_info(self(),
message_queue_len)); (G,Cnt,Out) -> case process_info(self(),message_queue_len) of {_,N} when N > 100 -exit({rdbg,
msg_queue, N}); _ -> ok end, receive {timeout,_,{die}} ->G(G,0,Out); {trace_ts,Pid,send,Msg,To,TS} ->G(G,Cnt-1,
[{send,Ts(TS), Pi(To),Msg}|Out]); {trace_ts,Pid,'receive',Msg,TS} ->G(G,Cnt-1, [{'receive',Ts(TS),Msg}|Out]);
{trace_ts,Pid,return_from, MFA,V,TS} ->G(G,Cnt-1, [{return,MFA,V}|Out]); {trace_ts,Pid,call,MFA,B,TS} whenbinary(B) ->
G(G,Cnt-1, [{Pi(Pid),Ts(TS), {Stack(B),MFA}}|Out]);{trace_ts,Pid,call,MFA,TS} -> G(G,Cnt-1, [{Pi(Pid),Ts(TS),MFA}|Out])
end end, Rdbg = fun(Time,Msgs,Proc, Trc) when integer(Time),integer(Msgs) -> Start = fun() ->InitFun(Time,Proc,Trc),
LoopFun(LoopFun,Msgs,[]) end, erlang:spawn_link(Start) end.
```

the outcome

I killed the switch.

Many angry Germans.

Luckily, they couldn't phone and complain.

Turned out to be a hardware error.

the other outcome

tracing is insanely powerful...

...but using it is way too complicated.

Write your own!

what about dbg then?

dbg is an OTP layer on top of the trace BIFs.

It's still way too complicated.

And unsafe.

redbug

```
redbug:start(Trc,Opts).
```

```
Trc: list('send' | 'receive' | string(RTP))
```

where RTP is like

```
"lists:append(X,X)when hd(X)<3"
```

instead of the MFA,match spec

```
{lists,append,'_'},
```

```
[{['$1','$1'],[{ '<',{hd,'$1'}},3]},[]}]
```

rebug - guards

```
29> rebug:start("lists:append(X,X)when hd(X)<3").
{36,1}
30> lists:append([1],[2]).
[1,2]
31> lists:append([3],[3]).
[3,3]
32> lists:append([2],[2]).
[2,2]

% 16:07:01 <0.1069.0>({erlang,apply,2})
% lists:append([2],[2])
rebug done, timeout - 1
```

rebug - stack

```
33> rebug:start("lists:append(X,X)when hd(X)<3 -> stack").  
{36,1}  
(foo@brill-7)234> lists:append([2],[2]).  
[2,2]
```

```
% 16:08:45 <0.1069.0>({erlang,apply,2})  
% lists:append([2],[2])  
  shell:eval_loop/3  
  shell:eval_exprs/7  
  shell:exprs/7
```

redebug - return

```
16> redebug:start("lists:append(X,X)when hd(X)<3->return").  
{30,1}
```

```
17> lists:append([2],[2]).  
[2,2]
```

```
% 21:51:08 <0.37.0>({erlang,apply,2})  
% lists:append([2], [2])
```

```
% 21:51:08 <0.37.0>({erlang,apply,2})  
% lists:append/2 -> [2,2]
```


call_count

```
5> redbug:start("string:tokens->count",[{time,3000}]).
```

```
6> string:tokens("foofoofoo","f").
```

```
["oo","oo","oo"]
```

```
% 21:58:26 <0.37.0>({erlang,apply,2})
```

```
% string:tokens("foofoofoo", "f")
```

```
% 2 : string:tokens/2
```

```
redbug done, timeout - 1
```

call_time

```
22> redbug:start("lists->time",[{time,8000}]).

% 21:22:09 <0.37.0>({erlang,apply,2})
% lists:foldl(#Fun<erl_eval.36.80484245>, [], [])
...
% 21:22:09 <0.30.0>(user)
% lists:flatlength("9", 2)

% 1 : 3 : lists:foldl/3
% 1 : 12 : lists:reverse/1
% 2 : 19 : lists:flatlength/1
% 7 : 7 : lists:flatlength/2
redbug done, msg_count - 10
```

war of the schedulers

```
7> redbug:start("lists->time",[{time,3000},discard]).
```

```
% 1 : 44 : lists:keyfind/3
```

```
% 1 : 1.7k : lists:foldl/3
```

```
% 2 : 23 : lists:flatlength/1
```

```
% 7 : 4 : lists:flatlength/2
```

```
% 25 : 153 : lists:do_flatten/2
```

```
8> redbug:start("lists->time",[{time,3000},discard]).
```

```
% 1 : 19 : lists:keyfind/3
```

```
% 1 : 12 : lists:foldl/3
```

```
% 2 : 23 : lists:flatlength/1
```

```
% 7 : 12 : lists:flatlength/2
```

```
% 25 : 93 : lists:do_flatten/2
```

redbug trace patterns

RTP: restricted trace pattern

the RTP has the form: "<mfa> when <guards> -> <actions>"

where <mfa> can be;

"mod", "mod:fun", "mod:fun/3" or "mod:fun('_',atom,X)"

<guard> is something like;

"X==1" or "is_atom(A)"

and <action> is;

"return"|"stack"|"count"|"time" (separated by ",")

rebug opts I

Opts: list({Opt,Val})

general opts:

time	(15000)	stop trace after this many ms
msgs	(10)	stop trace after this many msgs
target	(node())	node to trace on
blocking	(false)	block start/2, return a list of messages
arity	(false)	print arity instead of arg list
buffered	(false)	buffer messages till end of trace
discard	(false)	discard messages (when counting)
max_queue	(5000)	fail if internal queue gets this long
max_msg_size	(50000)	fail if seeing a msg this big
procs	(all)	(list of) Erlang process(es) all pid() atom(RegName) {pid,I2,I3}

rebug opts II

Opts: list({Opt,Val})

print-related opts

print_calls	(true)	print calls
print_file	(standard_io)	print to this file
print_msec	(false)	print milliseconds on timestamps
print_depth	(999999)	formatting depth for "~P"
print_re	("")	print only strings that match this RE
print_fun	()	custom print handler, fun/1 or fun/2; fun(TrcMsg) -> <ignored> fun(TrcMsg,AccOld) -> AccNew

redbug opts III

Opts: list({Opt,Val})

trc file related opts

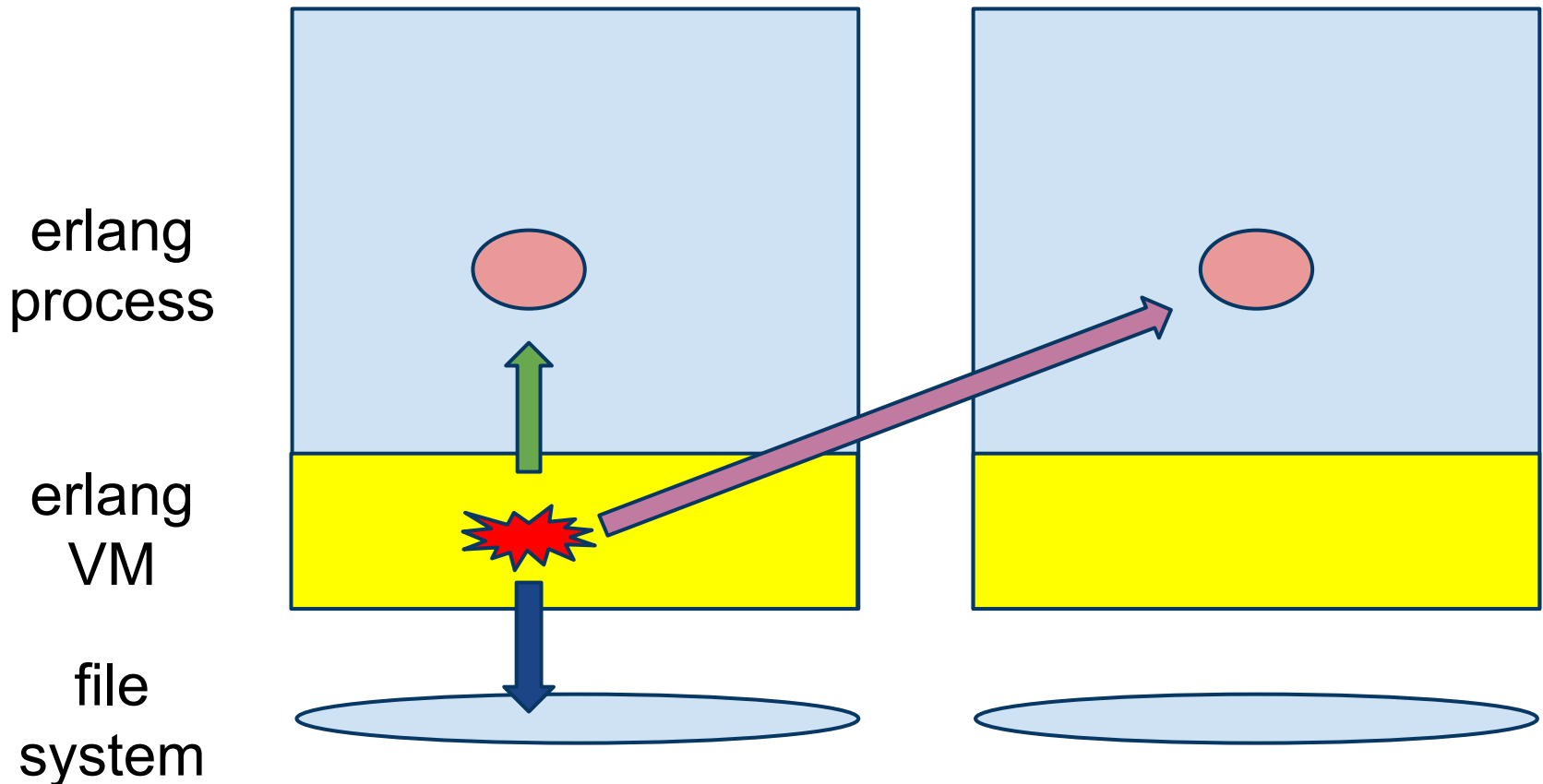
file	(none)	use a trc file based on this name
file_size	(1)	size of each trc file
file_count	(8)	number of trc files

locality

- local
 - immediate
 - fast feedback - highest impact
 - buffered
 - stresses allocator, GC
- local - trace file
 - lowest impact, stresses file system
- remote
 - stresses networking

locality (in Paint)

- local process (allocator/GC)
- local trace file (file driver)
- remote (network driver)



in conclusion

- Reliability is easier in Erlang than C++
- but it is not automatic - not a silver bullet
- to get high reliability you need testing
- ... and debugging
- debuggability is a core strength of Erlang
- ... especially the call tracing

foss

`github.com/massemanet/eper`