PULSE Tutorial

Hans Svensson and Michał Pałka

CHALMERS

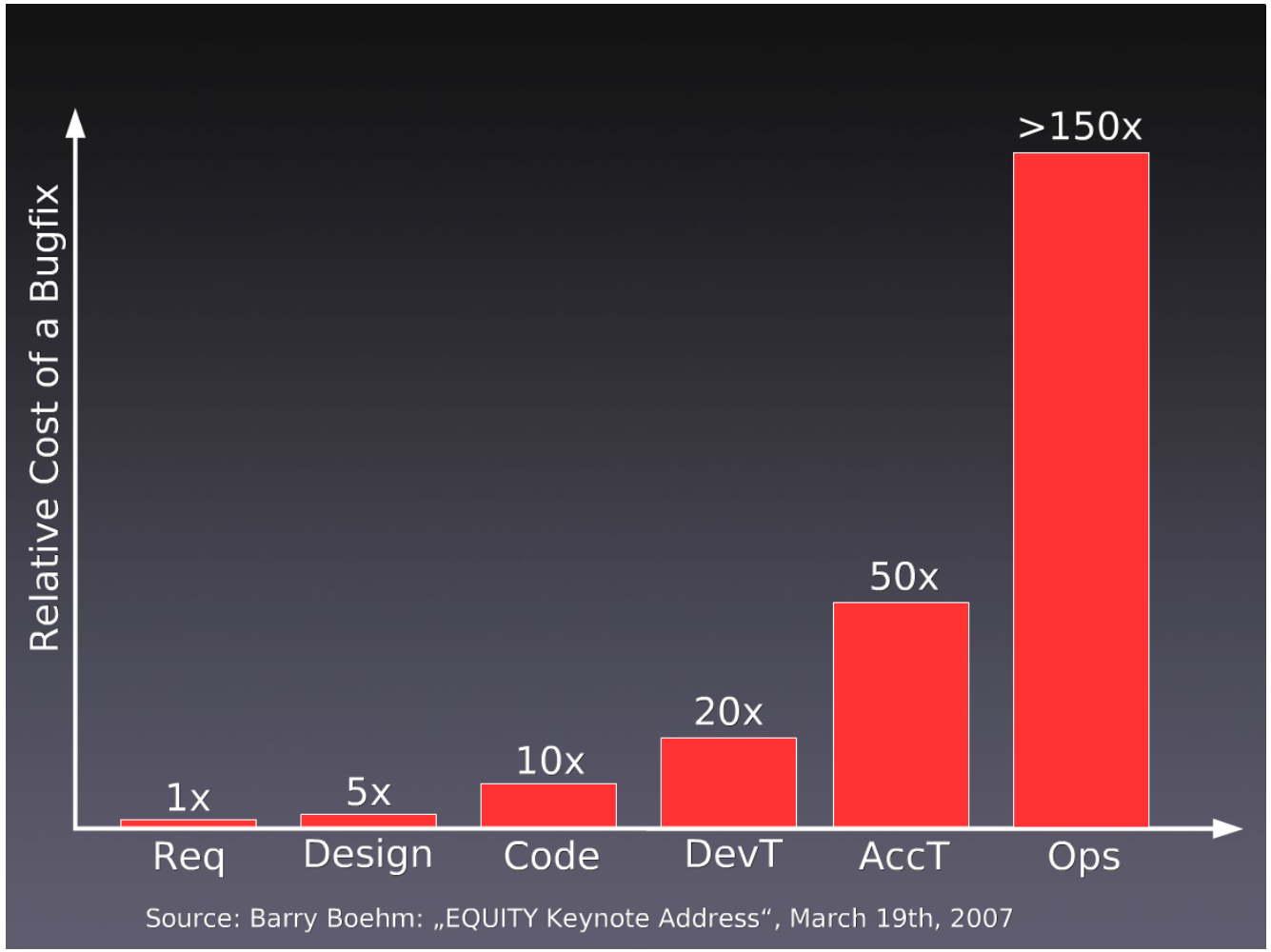
ProTest
property based testing

1.

"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year. We have not been able to track the bug down since the dets files is repaired automatically next time it is opened.“

Tobbe Törnqvist, Klarna, 2007

Cost of bug fixing



Slide: Advanced OOP and Design Patterns, Stefan Priebsch

ProTest
property based testing

Race conditions

- Many found late in the process (system testing, production)
- Often result of design errors
- Very expensive to fix

2.

Introducing example

- Very often we write code like:

```
Items = gather_items(),
lists:foreach(fun(I) -> process_item(I) end, Items)
```

or

```
Items = gather_items(),
Res = lists:map(fun(I) -> process_item(I) end, Items)
```

- Each `process_item(I)` is independent
- Our new 53-core machine has just arrived
- Natural place to parallelize!

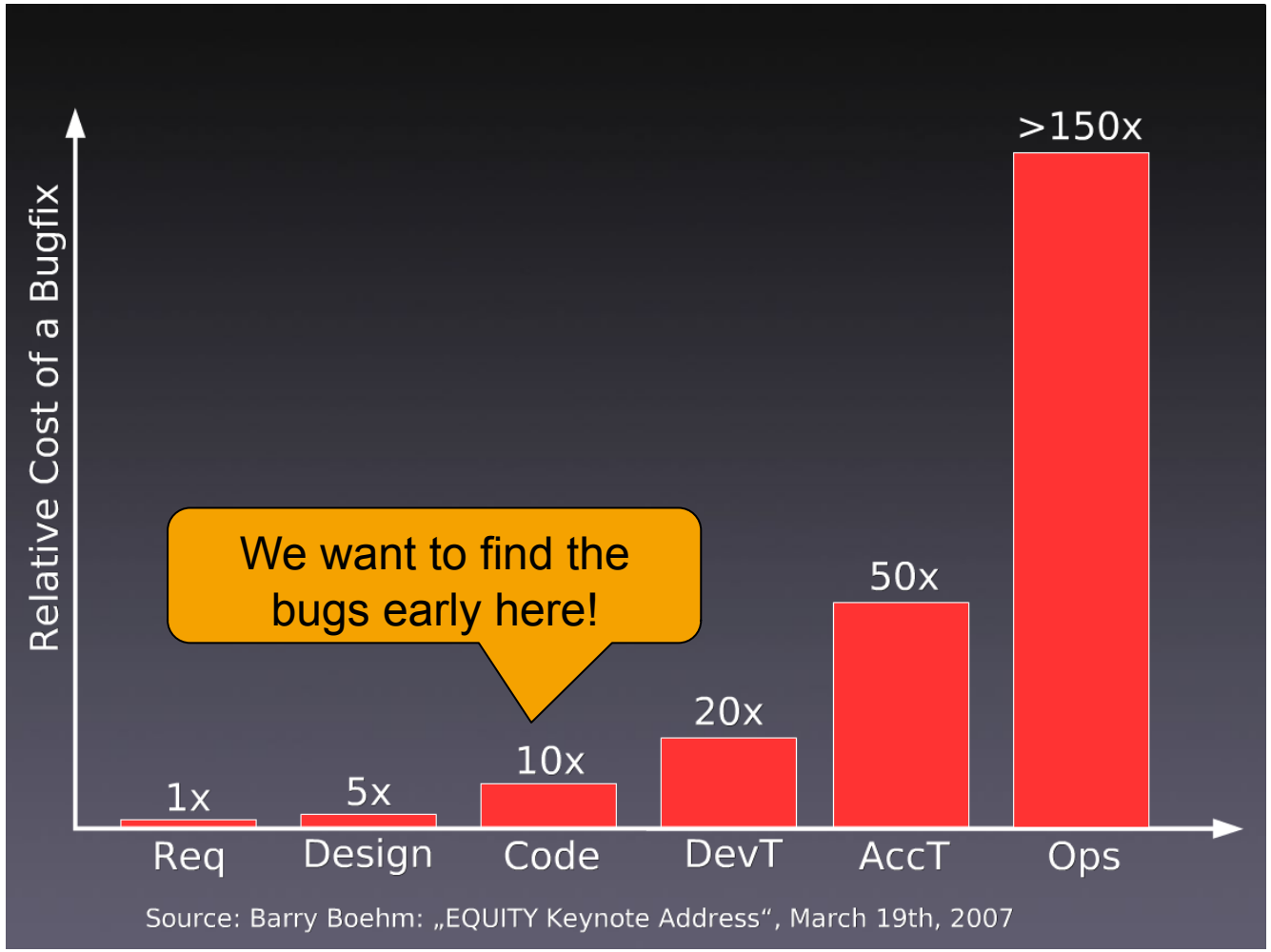
Introducing example

- Replace `map` with parallel map (`pmap`):

```
Items = gather_items(),
Res = pmap(fun(I) -> process_item(I) end, Items)
```

- Unfortunately there is no standard parallel map in Erlang
- How about implementing one!?

Avoid the cost of bugs



Slide: Advanced OOP and Design Patterns, Stefan Priebsch

ProTest
property based testing

QuviQ

- **First write the property!**

```
-include_lib("eqc/include/eqc.hrl").
prop_pmap() ->
  ?FORALL({Fun,Items}, {function1(nat()),list(nat())},
    begin
       Res  = lists:map(Fun,Items),
       PRes = pmap:pmap(Fun,Items),
       Res == PRes
    end).
```

Generates a random function returning natural numbers!

Apply normal map

Apply parallel map

Compare the results

Implementing pmap

- First attempt

```
-module(pmap).

-export([pmap/2]).

pmap(F,Ls) ->
  Self = self(),
  [spawn(fun() -> Self ! F(L) end) || L <- Ls],
  [receive Res -> Res end          || _ <- Ls].
```

Testing with QuickCheck

```
2>eqc:quickcheck(pmap_eqc:prop_pmap()).
.....................................................
.....................................
OK, passed 100 tests
true
```

Good, but let's run some more tests…

```
3>eqc:quickcheck(eqc:numtests(10000,pmap_eqc:prop_pmap())).
.....................................................
.....................................................

...

.......................................
OK, passed 10000 tests
true
```

ProTest
property based testing

- Perfect! Move on to next problem…
- Or wait a second, was our testing really thorough?

- A **concurrent** implementation on a slow single-core laptop!
- Not good enough!

Many race conditions show up only in production; they must be hard to test!

Testing with QuickCheck, 2nd try…

```
Erlang R13B02 (erts-5.7.2) … [smp:2:2]
…
5>eqc:quickcheck(pmap_eqc:prop_pmap()).
.....................................................
.....................................
OK, passed 100 tests
true
```

Still passes, maybe it is actually correct…

```
8>eqc:quickcheck(eqc:numtests(10000,pmap_eqc:prop_pmap())).
.....................................................
.....   ...   ....................Failed! After 841 tests.
{#Fun<eqc_gen.101.34507915>,[30,1,22,3,18,25,22]}
false
```

Ouch!

ProTest
property based testing

Property Driven Development

- We need more information!

?WHENFAIL – to run code when a property fail

- We want to see the values of **Res** and **PRes**.

```
-include_lib("eqc/include/eqc.hrl").
prop_pmap() ->
  ?FORALL({Fun,Items}, {function1(nat()),nat()},
    begin
       Res  = lists:map(Fun,Items),
       PRes = pmap:pmap(Fun,Items),
       ?WHENFAIL(
          io:format("~p /= ~p\n",[Res,PRes]),
          Res == PRes)
    end).
```

ProTest
property based testing

Testing with QuickCheck, 2nd try…

```
8>eqc:quickcheck(eqc:numtests(1000,pmap_eqc:prop_pmap())).
.....................................................................
.....   ...   ..............Failed! After 710 tests.
{#Fun<eqc_gen.101.41379873>,[11,19,14,14,6,32,33,18,26,7]}
[30,13,5,5,0,21,24,29,4,20] /= [30,13,5,5,0,21,24,29,20,4]
Shrinking.(1 times)
{#Fun<eqc_gen.101.41379873>,[11,19,14,14,6,32,33,18]}
[30,13,5,5,0,21,24,29] /= [30,29,24,13,21,5,0,5]
false
```

Observations

- Switching to multi-core (or enabling SMP) makes concurrency bugs more likely to manifest
- We had to run quite a few tests
- Shrinking didn’t work (very well)
  - A small counterexample is often very valuable
  - Shrinking a counterexample is done stepwise
  - Counterexample that ‘happens’ to fail will not shrink well

Erlang scheduling

- The Erlang scheduler is **too deterministic**
  - Small tests
  - Low load on system
  - Deterministic even in multi-core systems
  - Large tests are needed to provoke race conditions
  - Many race conditions may not show up until you deploy your system
- With randomized scheduling
  - Small tests are more likely to provoke race conditions
  - Find concurrency bugs early in development process

3.

PULSE to the rescue

- **PULSE** to the rescue
  - **P** – ProTest
  - **U –** User
  - **L –** Level
  - **S –** Scheduler
  - **E –** for Erlang
- **PULSE** is non-deterministic (random scheduling)
- **PULSE** can re-run a schedule (repeatable tests)

How **PULSE** works

- Controls the concurrency
  - Only one process is executing at a time
- Records all concurrency events
  - Message sending
  - Process spawning
  - Etc…
- **PULSE** can switch to executing another process (simulating context switch) at any time
- We make sure that unlikely scenarios get tested

How to use **PULSE**

- pulse_instrument:
  - Instrumentation of the code at compile time
- Implemented as parse_transform compiler option

- Example:

```
c(example,[{parse_transform,pulse_instrument}]).
```

- Calls to ***spawn***, ***link*** as well as statements ***!*** and ***receive***, etc are replaced by calls handled by **PULSE**

How to use **PULSE**

- Running instrumented code:

```
5> c(pmap,[{parse_transform,pulse_instrument}]).
{ok,pmap}
6> pulse:run(fun() ->
       pmap:pmap(fun(X) -> X + 2 end,[1,2]) end).
** exception exit: {application,pulse_not_running}
       in function pulse:spawn/2
...
```

Application **PULSE** must be running: `pulse:start().`

The **PULSE** application keeps state: last used schedule, random seed, etc, and gives access to event handlers for different kind of output.

How to use **PULSE**

```
8> pulse:start().
Starting eqc version 1.18 …
9> pulse:run(fun() ->
       pmap:pmap(fun(X) -> X + 2 end,[1,2]) end).
[3,4]
scheduling started
root spawns pmap <0.234.0>
root spawns pmap1 <0.235.0>
root blocks
pmap sends 3 to root
pmap terminated normally
root receives 3
...
return value [3,4]
scheduling finished
10>
```

ProTest
property based testing

?PULSE macro

QuickCheck uses ?PULSE macro:

```
?PULSE(
  <Pattern bound to result of E>,
  <Expression E to run in PULSE>,
  <Property using result of E>
)
```

- Normal compilation:

  Run code normally

- Compilation with pulse_instrument, **PULSE** running:

  Run code with **PULSE** scheduler

How to use **PULSE** with **QuickCheck**

- Update property!

```
-include_lib("eqc/include/eqc.hrl").
prop_pmap() ->
  ?FORALL({Fun,Items}, {function1(nat()),nat()},
    begin
       Res  = lists:map(Fun,Items),
       PRes = pmap:pmap(Fun,Items),
       ?WHENFAIL(
         io:format("~p /= ~p\n",[Res,PRes]),
         Res == PRes)
    end).
```

This is what we want to run in PULSE

How to use **PULSE** with **QuickCheck**

- Update property!

PULSE definitions

```
-include_lib("eqc/include/eqc.hrl").
-include_lib("pulse/include/pulse.hrl").
prop_pmap() ->
  ?FORALL({Fun,Items}, {function1(nat()),nat()},
    begin
      PRes = pmap:pmap(Fun,Items),
      ?WHENFAIL(
          io:format("~p /= ~p\n",[Res,PRes]),
          Res == PRes)
    end).
```

PULSE macro

PRes = pmap:pmap(Fun,Items)

ProTest
property based testing

Verbosity in **PULSE**

- Don’t forget the verbosity:
  - **pulse:verbose/1.**

```
24> pulse:verbose([]).
ok
25> pulse:run(fun() ->
       pmap:pmap(fun(X) -> X + 2 end,[1,2]) end).
[3,4]
26> pulse:verbose([all]).
ok
27> pulse:run(fun() ->
       pmap:pmap(fun(X) -> X + 2 end,[1,2]) end).
[3,4]
scheduling started
root spawns pmap <0.234.0>
root spawns pmap1 <0.235.0>
...
```

ProTest
property based testing

Verbosity in **PULSE**

- Verbosity options:
  - **`all`** – All verbosity flags
  - **`send`** – Show sending of messages
  - **`'receive'`** – Show delivery and receiving of messages
  - **`procs`** – Show process events (spawn, link, etc.)
  - **`side_effect`** – Show (user defined) side effects

- Options are similar to ***trace patterns***

How to use **PULSE** with **QuickCheck**

```
32> pulse:verbose([]).
ok
33> eqc:quickcheck(pmap_eqc:prop_pmap()).
.............Failed! After 23 tests.
{#Fun<eqc_gen.101.34457915>,[0,2,0]}
{29191,1432,12821}
[3,1,1] /= [1,3,1]
Shrinking...(3 times)
{#Fun<eqc_gen.101.34457915>,[0,1]}
{29191,1432,12821}
[3,0] /= [0,3]
false
```

- Fewer test cases needed
- Shrinking works (for this example)

ProTest
property based testing

Understanding the counterexample

- What is the error?
- We can use **`pulse:rerun_counterexample/2`** to re-run the counterexample with more verbosity
  - Gets the last counterexample from **`eqc:counterexample/0`**
  - Uses **`eqc:check/2`** to re-run the property

```
35> pulse:rerun_counterexample([all],pmap_eqc:prop_pmap()).
scheduling started
root spawns pmap <0.244.0>
root spawns pmap1 <0.245.0>
root blocks
pmap sends 3 to root
pmap terminated normal
pmap1 sends 0 to root
pmap1 terminated normal
pmap1 delivers 0 to root
root receives 0
root blocks
pmap delivers 3 to root
root receives 3
return value [0,3]
scheduling finished
Failed!
{#Fun<eqc_gen.101.34457915>,[0,1]}
{29197,1532,821}
[3,0] /= [0,3]
false
36>
```

Visualization

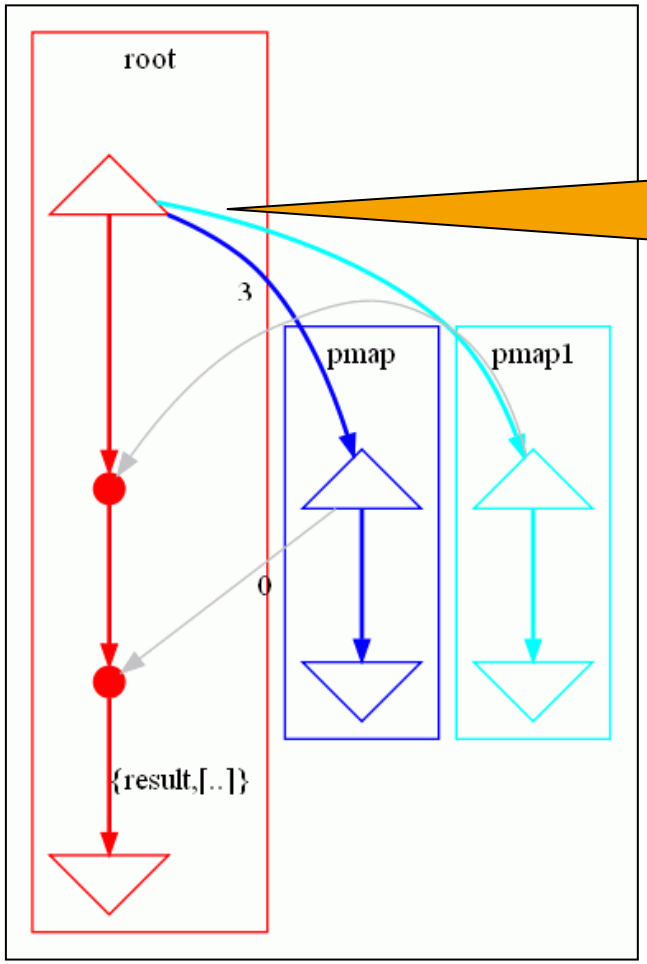
- Another way of understanding an error
- We can visualize the schedule to easier understand it!

- Requires pulse_event_graph to be added as event handler: **`pulse_event_graph:start().`**

```
36> pulse_event_graph:start([]).
ok
37> pulse:rerun_counterexample([],pmap_eqc:prop_pmap()).
pulse_event_graph set verbose to []
pulse_event_terminal set verbose to []
Failed!
...
```

- Every scheduled run now creates a graph.dot file!

Visualization



Work in progress, the only thing seen is the order of the messages.

Requires GraphViz to be installed. In particular the program **dot**
*http://www.**graphviz**.org/*

ProTest
property based testing

pmap 2nd attempt

- We need to ensure the order of the results:

```
-module(pmap).

-export([pmap/2]).

pmap(F,Ls) ->
  Self = self(),
  Pids = [spawn(fun() -> Self ! {self(),F(L)} end)
          || _ <- Ls],
  [receive {Pid,Res} -> Res end
          || Pid <- Pids].
```

Tag the messages with the Pid of the worker process

Use selective receive to fetch the results in order

ProTest
property based testing

Testing the new implementation

```
45>eqc:quickcheck(pmap_eqc:prop_pmap()).
.....................................................
.....................................
OK, passed 100 tests
true
```

Good, but again, let's run some more tests…

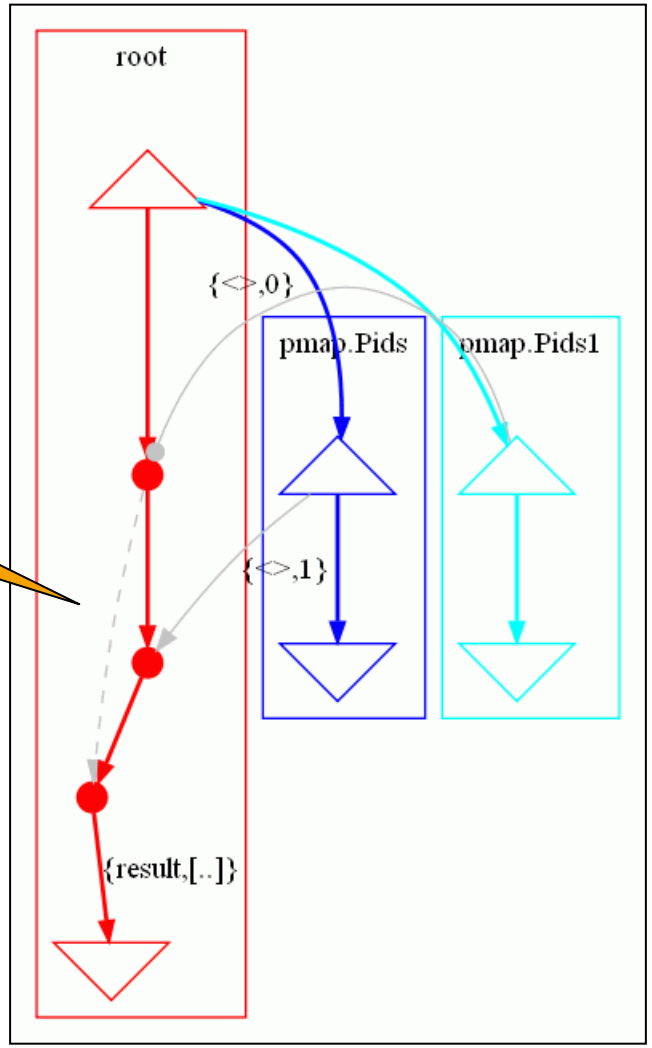
```
48>eqc:quickcheck(eqc:numtests(10000,pmap_eqc:prop_pmap())).
.....................................................
.....................................................
...
.....................................
OK, passed 10000 tests
true
```

Done! Not quite; our implementation doesn't handle errors...

ProTest
property based testing

Visualization – a correct run

The messages are delivered in the wrong order, but **consumed** in the right order



ProTest
property based testing

Short break!

Try it yourselves!

Next: User defined side effects

ProTest
property based testing

4.

Side effects

- Concurrency errors can be caused by modules interacting with other modules
- Example: writefile

```
prop_writefile() ->
  ?FORALL({Text1,Text2},{string(),string()},
    begin
      ok = file:write_file(?TESTFILE,Text1),
      ok = file:write_file(?TESTFILE,Text2),
      {ok,Bin} = file:read_file(?TESTFILE),
      binary_to_list(Bin) == Text2
    end).
```

Sequential writes are obviously safe. How about parallel file writing?

Side effects

- With a simple ?PAR macro we parallelize the writes

```
-define(PAR(E1,E2),
        begin
          spawn(fun() -> E1 end),
          spawn(fun() -> E2 end)
        end).

prop_writefile() ->
  ?FORALL({Text1,Text2},{string(),string()},
    begin
      ?PAR(file:write_file(?TESTFILE,Text1),
           file:write_file(?TESTFILE,Text2)),
      {ok,Bin} = file:read_file(?TESTFILE),
      Res = binary_to_list(Bin),
      Res == Text1 orelse Res == Text2
    end).
```

Write files in parallel

The result should be **either** of the strings

ProTest
property based testing

Example: write_file

```
2> eqc:quickcheck(writefile:prop_writefile()).
.Failed! After 2 tests.
{“e”,”q”}
false
```

Strange! Fails almost immediately, on very short strings.

- Add some more output:

```
...
      {ok,Bin} = file:read_file(?TESTFILE),
      Res = binary_to_list(Bin),
      ?WHENFAIL(
        io:format(“Res: ~\n”,[Res]),
        Res == Text1 orelse Res == Text2)
    end).
```

Example: write_file – more output

```
7> eqc:quickcheck(writefile:prop_writefile()).
Failed! After 1 tests.
{“f”,”e”}
Res: “z”
false
```

???
Where does “z” come from? Maybe we should try **PULSE?**

Example: write_file – **PULSE**

- Add ?PULSE to the property:

```
prop_writefile() ->
  ?FORALL({Text1,Text2},{string(),string()},
    ?PULSE(
      Res,
      begin
        ?PAR(...),
        {ok,Bin} = file:read_file(?TESTFILE),
        binary_to_list(Bin),
      end,
      ?WHENFAIL(io:format("Res: ~\n",[Res]),
                Res == Text1 orelse Res == Text2))).
```

Example: write_file – more output

```
9> pulse:start(),pulse:verbose([all]).
...
10> eqc:quickcheck(writefile:prop_writefile()).
scheduling started
root spawns ‘prop_writefile.Res’ <0.1528.0>
root spawns ‘prop_writefile.Res1’ <0.1529.0>
return value “k”
‘prop_writefile.Res1’ terminated normal
‘prop_writefile.Res’ terminated normal
scheduling finished
Failed! After 1 tests.
{“f”,”e”}
{8534,66433,27482}
Res: “k”
false
```

Doesn’t tell us very much more, we know that write_file is a side-effect, but PULSE does not...

ProTest
property based testing

Example: write_file – **PULSE** behavior

```
11> pulse:rerun_counterexample([all],
                               writefile:prop_writefile()).
scheduling started
...
OK, passed the test.
true
```

???
Now the test passed!

- Important **PULSE** fact:

  PULSE only controls the instrumented program and not the whole environment!

- **PULSE** cannot re-run a schedule (faithfully) when the environment has changed (new files are written etc...)

User defined side-effects

- We want **PULSE** to show an event when we perform a file operation.

All calls to module file are considered side effects:

```
c(writefile,
    [{parse_transform,pulse_instrument},
     {pulse_side_effect,[{file,'_','_'}]}]).
```

Matching module, function, arguments

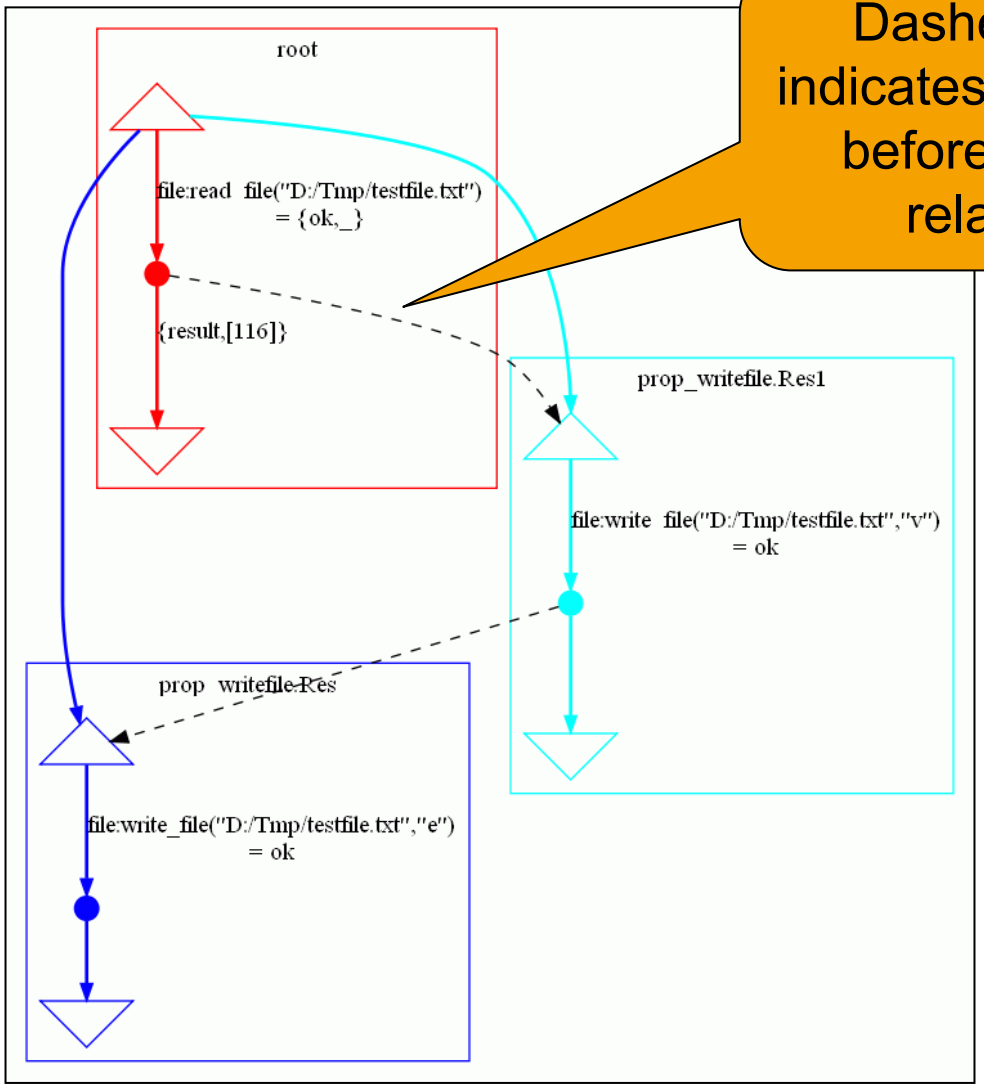
Example: write_file – more output

```
13> eqc:quickcheck(writefile:prop_writefile()).
scheduling started
root spawns ‘prop_writefile.Res’ <0.1832.0>
root spawns ‘prop_writefile.Res1’ <0.1833.0>
root yields
‘prop_writefile.Res1’ yields
‘prop_writefile.Res’ yields
root continues
root side-effect file:read_file(
    “D:/Tmp/testfile.txt”) result in {ok,<<“x”>>}
return value “x”
‘prop_writefile.Res1’ continues
‘prop_writefile.Res1’ side-effect file:write_file(
    “D:/Tmp/testfile.txt”) result in ok
...
```

Aha! We are reading the file before **either** of the writers has written anything!

ProTest
property based testing

Writefile – visualization



Dashed lines indicates 'happens-before' causal relations

ProTest
property based testing

Solution: synchronize

- PAR spawns two processes, but a third process is also running in parallel to them!
- Very common error
- Solution: Synchronize

```
-define(PAR(E1,E2),
  begin
      Self = self(),
      spawn(fun() -> E1, Self!done end),
      spawn(fun() -> E2, Self!done end),
      receive done ->
          receive done -> ok end
      end
  end).
```

ProTest
property based testing

Exercise:
Master-slave workers

Master-slave worker example

- N workers: one master and N-1 slaves
- Process registry is used to identify the master

- Functionality in: **`master.erl`**
- Test case in: **`master_eqc.erl`**

- There is a race condition in the code, which is hard to provoke with a test case
- Hint: use **`pulse_side_effect`**

5.

PULSE – summary

- Race conditions can be very difficult to find in early testing
- Bugs found late are very expensive to fix
- PULSE can be used to find hard-to-find race conditions that would otherwise be hidden until very late
- PULSE is most useful for small but critical modules of an application (steep learning curve)

PULSE – extras and future

- Two useful tricks
- Performance with **PULSE**
- A success story
- Availability of **PULSE**
- The future of **PULSE**

Tips and Tricks

- What to do when shrinking doesn't work?
- Mostly important in larger more complex examples
- Even with pulse the counterexamples can be large

- Visualization is also useful, but graphs quickly get quite large

Tips and Tricks

- **Idea 1:** ?ALWAYS(N,Property)-macro tries the property N times, and fails if any of the tries fails
- **Idea 2:** Try the property many times while shrinking to increase the chance of hitting the bug

```
prop_X() ->
  ?LET(Tries, ?SHRINK(1,[10]),
  ...
    ?ALWAYS(Tries,
      <Property>
    )
  ...
  ).
```

Tries will be 1 during normal testing and 10 during shrinking

PULSE performance

- Comparing performance
- Used parallel map as benchmark
  - Short computations: fib(N) where N = 10-15
  - Long computations: fib(N) where N = 30-35
- Single core:
  - With longer computations **PULSE** is faster!
  - With short computations, communication dominates and **PULSE** is (much) slower
- Multi-core:
  - **PULSE** is always slower, since it only uses one of the cores.

PULSE performance

- Performance is very application dependent
- Communication bound applications could be x100 slower.
- A ‘normal’ distributed application is likely to be x10 slower
  - Due to not using multi-core
  - and slower communication

A success story – **PULSE** used for `proc_reg`

- Real industrial example
- An optimized process registry
- Concurrency errors found by stress testing in 2006 (very large counterexamples)
- Nobody was able to track down the errors, so the component was shelved
- With PULSE we got shorter counterexamples
- With PULSE and the visualizer we could explain the error
- Described in paper at ICFP 2009

PULSE availability

- Two versions:
  - Open source version (BSD license)
    - Developed at Chalmers
    - Work in progress (ProTest)
    - Not very user-friendly
    - No public release yet
  - Commercial version
    - Available as part of Quviq QuickCheck
    - Package **PULSE** in application
    - Integrates QuickCheck and **PULSE**

ProTest – PULSE future plans

- Missing features (multi-node support etc)
- Improve shrinking of traces
- Re-write the core for a more modular design (already started)
- Support for testing timing dependent code (`receive after X -> ...`)
- Package and release open source version

Thank you!

CHALMERS

ProTest
property based testing