Erlang Programming for Multi-core

Ulf Wiger
Ericsson AB
ulf@wiger.net

http://www.protest-project.eu/

ERICSSON
TAKING YOU FORWARD

The purpose of this presentation is to give a hands-on tutorial on Erlang programming for multi-core.

Since providing the audience with a multi-core computer each seems problematic, and since the idea of Erlang programming for multi-core is that your programs should run unchanged with good characteristics, a regular tutorial seems impractical.

Instead, this is a fairly detailed briefing with some code examples intended to illustrate which things one needs to consider, followed by a presentation of an interesting technique for testing and debugging programs on multi-core.

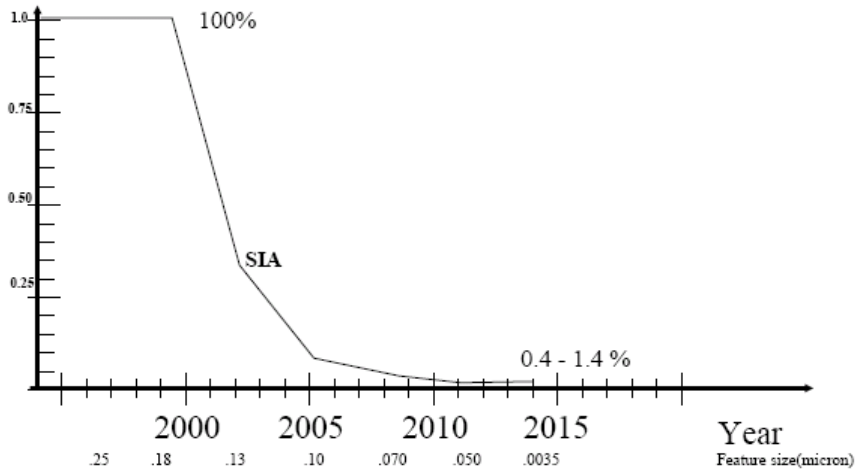
The presentation is part of the EU-sponsored research project ProTest. The intention is to make the QuickCheck functionality shown in this presentation (or something similar) commercially available.

Outline

- Background
- Industrial example
- Programming examples
- Some benchmarks
- Testing and debugging

Enter the age of multicore

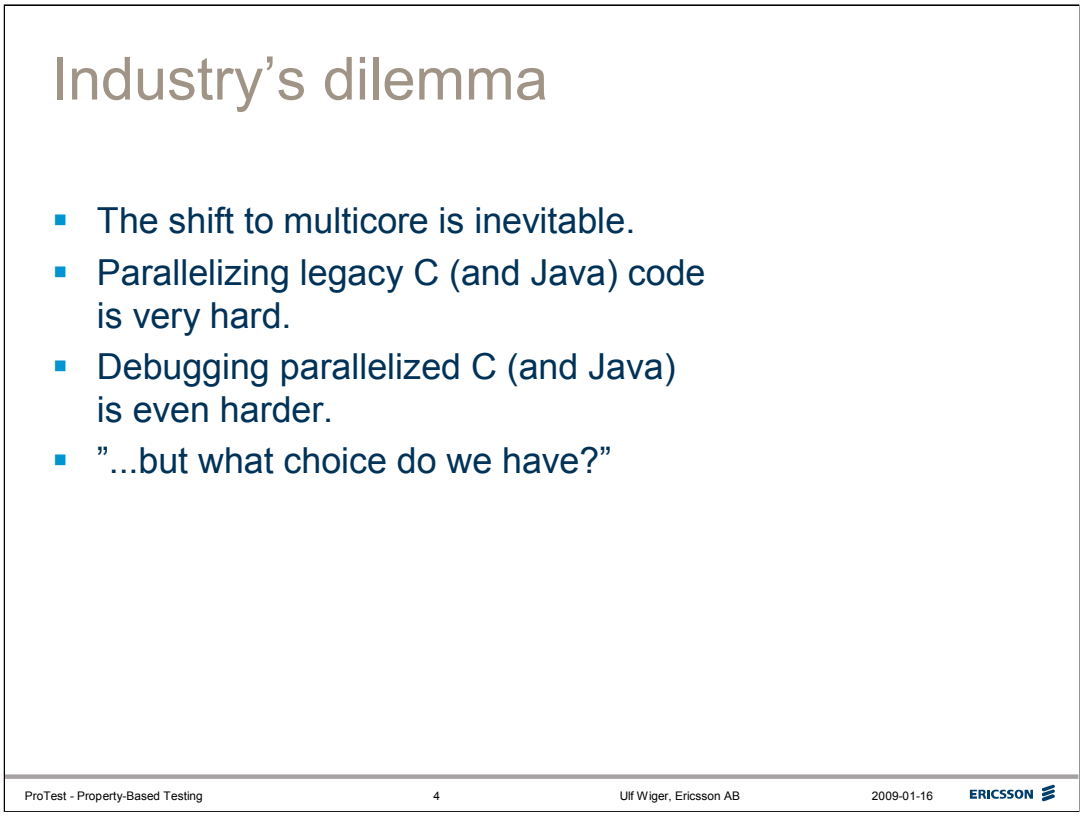
Fraction of chip reachable within one clock cycle



[source] Erik Hagersten (http://www.sics.se/files/projects/multicore/day2007/ErikHintro.pdf)

The picture is borrowed from dr. Joe Armstrong, who borrowed it from prof. Erik Hagersten.

Presumably, the people attending this workshop are aware of the reasons behing the multi-core trend, and no time will be spent delving into that here.



This is my own executive summary of the discussions so far on multi-core, at least in industry.

A possible result of this could be that the trend towards multi-core is slowed, since most legacy software is unable to take advantage of more than 2 or 4 cores.

I've often heard the comment "what choice do we have?", by people who seem either unaware of the existence of alternative technologies, or refuse to regard them as commercially viable. Of course, one should not underestimate the risks (both technical and commercial) of rewriting an established product from scratch, so in many cases, this may well be the truth – at least in the short term.

Erlang and multicore

- Erlang was designed for
  - Share-nothing concurrency
  - Asynchronous message passing
  - Distribution transparency
  - Fault-tolerance (”robustness in the presence of SW errors”)
- No mutexes, no transaction memory
  - Great for multicore – for some types of problem
  - Inadequate for some others
  - Erlang’s domain focus strongly affects how it approaches multicore
- But the Erlang VM of course uses shared memory and POSIX threads...

In this presentation we will focus on Erlang and multi-core.

Erlang has many traits that make it nearly ideal for multi-core, at least for a certain class of problems.

Erlang was originally designed for telecommunications software, where distributed programming has been the norm ever since the 1980s. Telecoms software also has a great deal of natural concurrency. Erlang processes were designed to have the right kind of concurrency for (the logical representation of) a telephone call. In fact, one early prototype used Parlog, which was deemed to be much *too parallel*; and the concurrency couldn’t be controlled sufficiently for the problem domain.

It follows that Erlang’s process model is too heavyweight for some forms of parallelism (e.g. the ones for which Parlog would be ”just right”).

Erlang has no shared memory. The main reason for this was reliability (one process cannot corrupt the memory of another), but also because distribution transparency was desired, and processes cannot share memory across a network link.

Fault tolerance was also an important requirement. The philosophy was that errors cannot be entirely eliminated, so the software must be able to withstand errors even in the field.

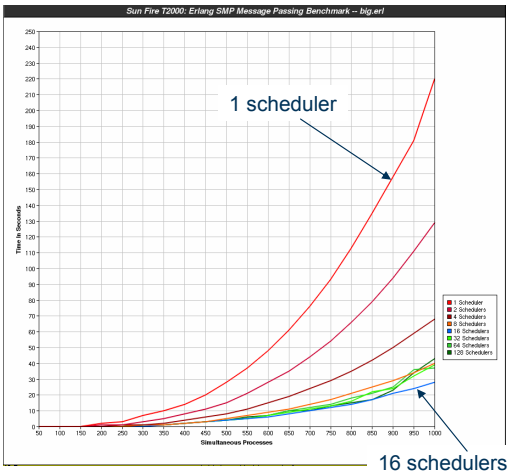
The fault-tolerance and distribution transparency requirements led naturally to an asynchronous message-passing model.

When looking at the challenges for multi-core chip designers, it seems that Erlang’s asynchronous message-passing share-nothing model is quite attractive, especially for many-core (>> 4 cores). However, mainstream chips are naturally focused mainly on performing well for legacy software, which largely is either single-threaded, or relies on a POSIX shared-memory thread model. Thus, neither the CPUs nor the common operating systems can be expected to offer instruction sets and frameworks for message-passing concurrency at the chip level.

Erlang on multicore

- SMP prototype ‘97,
  First OTP release May ‘06.
- Mid -06 we ran a benchmark mimicking call handling (axdmark) on the (experimental) SMP emulator. Observed speedup/core: 0.95
- First Ericsson product (TGC) released on SMP Erlang in Q207.

”Big bang” benchmark on Sunfire T2000



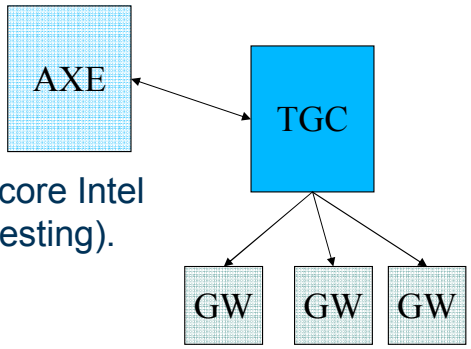
http://www.franklinmint.fm/blog/archives/000792.html

The first SMP experiments with Erlang were made in 1997, as a MSc thesis project by Pekka Hedqvist (supervised by Tony Rogvall). It was a success inasmuch as it showed that normal Erlang programs can scale well with the help of SMP, but as the products using Erlang had no room (literally) for the rather bulky SMP systems of that time, the work wasn’t continued.

As the multi-core trend started picking up a little, the work was revived. Tony Rogvall (then at Synapse) assisted in the beginning, and the first SMP-capable Erlang/OTP version was released in May 2006. It was considered experimental. Ericsson ran some benchmarks and ported a commercial product to it (the main initial challenge was that this required moving from a PowerPC architecture to AMD64). These experiments on ”real software” (including linked-in drivers etc.) led to several improvements to the VM, and in the second quarter of 2007, we were able to release our first commercial product using SMP Erlang.

Case study: The Ericsson TGC

- **Telephony Gateway Controller**
- Mediates between legacy telephony and multimedia networks.
- Hugely complex state machines + massive concurrency.
- Developed in Erlang.
- Multicore version shipped to customer Q207.
- Porting from 1-core PPC to 2-core Intel took < 1 man-year (including testing).



The first commercial product using SMP Erlang (as far as we know) was the Ericsson Telephony Gateway Controller. This product mediates between legacy telephony networks and IP telephony, and contains some frightningly complex state machines and massive concurrency. It is required to have at least 99.999% availability (< 6 min/year downtime, including maintenance and upgrades.) The work to port to SMP and verify the TGC on dual-core AMD64 boards was less than one man-year in total (not counting the amount of work put in by the Erlang/OTP team, of course.) For a product of this complexity, this effort is almost negligible.

TGC Results (top)

```
Tasks:  50 total,   2 running,  48 sleeping,   0 stopped,   0 zombie
Cpu0  : 62.5% us,  3.7% sy,  0.0% ni, 32.4% id,  0.0% wa,  0.0% hi,  1.3% si
Cpu1  : 36.1% us,  2.7% sy,  0.0% ni, 60.9% id,  0.0% wa,  0.0% hi,  0.3% si
Mem:   4092764k total,   459352k used,  3633412k free,     8196k buffers
Swap:        0k total,        0k used,        0k free,   215796k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1975 homer     25   0 2295m 192m 2144 S 99.9  4.8 179:40.46 beam.smp
    1 root      16   0   664  244  208 S  0.0  0.0   0:01.50 init
    2 root      RT   0     0    0    0 S  0.0  0.0   0:00.02 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    4 root      RT   0     0    0    0 S  0.0  0.0   0:00.01 migration/1
    5 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/1
```


This slide shows a snapshot of the Unix ”top” utility.

Highlighted are the load figures for the two CPUs, indicating a reasonably (but not perfectly) even load, as well as the TIME column, just to show that the system was indeed capable of running for a while.

TGC Results (dtop)

```
ppb1_bs13-R3A@blade_ size 2345(131M, cpu% 107  procs  10371, runq   0  15:15:53
memory[kB]:  proc   58223, atom    1768, bin      170, code   29772, ets   39215

pid         name                            current             msgq   mem cpu
<0.5872.491 prfTarg                         (prfPrc:pinf/2)        0  2036  22
<0.18323.47 (erlang:apply/2)                (gcpServ:recv1/3)      0    17  10
<0.18436.47 (erlang:apply/2)                (gcpServ:recv1/3)      0    24   5
<0.1813.0>  sysProc                         (gen_server:loop/6)    0   981   2
<0.27384.47 (pthTcpNetHandler:init/1)       (gen_server:loop/6)    0   587   1
<0.18350.47 (erlang:apply/2)                (gcpTransportProxy:    0     8   1
<0.1935.0>  ccpcServer_n                    (gen_server:loop/6)    0   587   0
<0.18526.47 (erlang:apply/2)                (gcpTransportProxy:    0     6   0
<0.1923.0>  sbm                             (gen_server:loop/6)    0  1719   0
<0.3603.0>  (erlang:apply/2)                (gcpServ:recv1/3)      0     5   0
```


This slide shows a snapshot of the Erlang utility ”dtop” (modeled after ”top”).

Highlighted are the number of simultaneous processes (10 371), and the length of the run queue (0). The run queue is a fairly reliable indicator of how loaded the system is. If the system is not overloaded, the length of the run queue is often zero, or close to zero.

TGC results (performance)

Traffic scenario	IS/GCP 1slot/board	IS/GEP Dual core One core running 2slots/board	IS/GEP Dual core Two cores running 2slots/board	AXD CPB5	AXD CPB6
POTS-POTS /AGW	X call/sec	2.3X call/sec One core used	4.3X call/sec OTP R11_3 beta+patches	0.4X call/sec	2.1X call/sec
ISUP-ISUP /Inter MGW	3.6X call/sec	7.7X call/sec One core used	13X call/sec OTP R11_3 beta+patches	.55X call/sec	7.6X call/sec
ISUP-ISUP /Intra MGW	5.5X call/sec		26X call/sec	3.17X call/sec	14X call/sec

ProTest - Property-Based Testing 10 Ulf Wiger, Ericsson AB 2009-01-16 ERICSSON

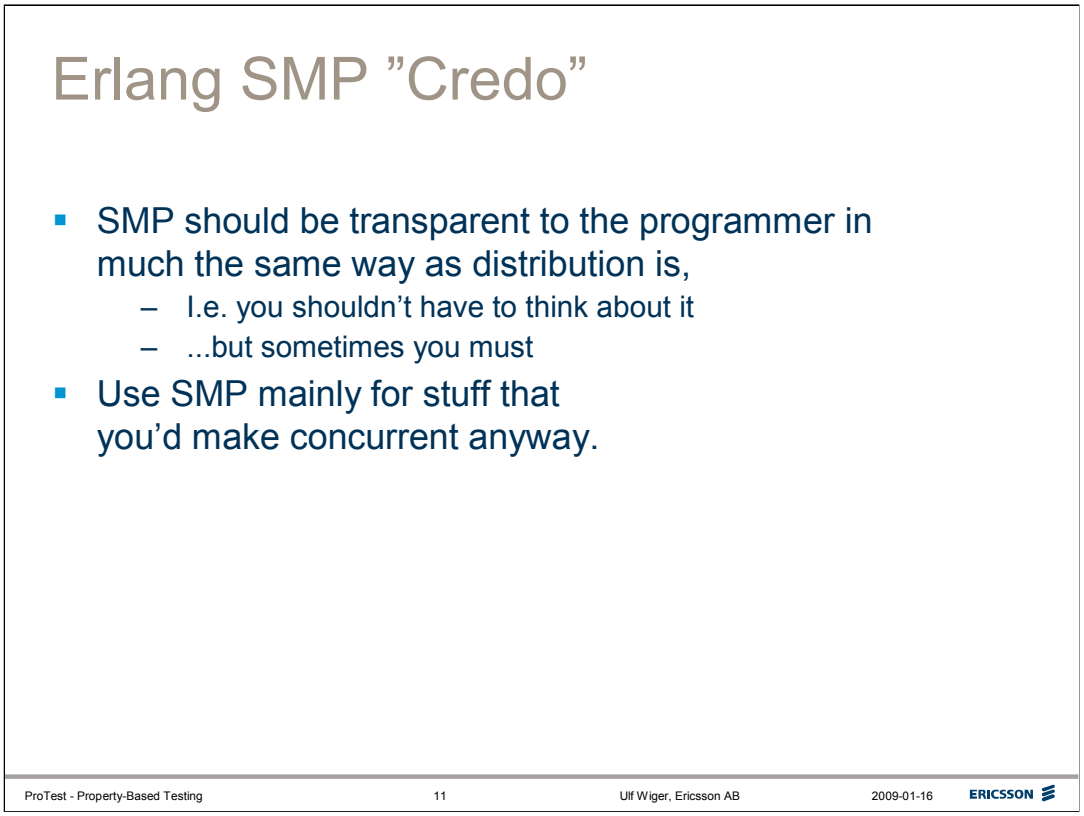
This table shows relative performance figures for the TGC.

The reference is a single-core PowerPC. The actual performance is not revealed, but represented here as a factor of X.

The ”GEP” processor is a dual-core AMD64, so to get a fair comparison, we ran the test using the non-SMP emulator on the GEP, using only a single CPU. Then we used the SMP emulator and observed a significant speedup.

The CPB5 and CPB6 boards are other single-core references. The CPB6 is roughly as fast as the AMD64 using non-SMP Erlang. This is roughly as fast as we can make it go on a single-core, given the power and space budget on the board.

The observed speedup going from single-core to dual-core was ca 1.7, which must be seen as a very good result.



It is important to understand that the Erlang approach to SMP is that existing programs should benefit from multi-core unchanged, and that programmers should not have to write special code for SMP scalability.

For one thing, this means that a tutorial on multi-core programming in Erlang would amount to much the same as a basic Erlang programming tutorial...

In the interest of time, I will skip the basic Erlang part (hoping that the audience has assimilated that already, or is smart enough to follow anyway), and focus on those aspects which *are* different about SMP Erlang.

The basic philosophy of Erlang can be described as ”model naturally concurrent activities, and create as many processes as your problem calls for – no more, no less.”. We call this Concurrency-Oriented Programming (COP), and it ows much to C.A.R. Hoare’s work on CSP.

Programming Examples

Traditional implementation of lists:map/2:

```
map(F, L) ->
   [F(X) || X <- L].
```

(Simple) SMP implementation: pmap/2:

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

Not quite the same semantics...

Order: preserved
Exceptions: hangs

ProTest - Property-Based Testing 12 Ulf Wiger, Ericsson AB 2009-01-16 ERICSSON

Let’s go through one programming example: parallelizing the map function.

This can actually be beneficial in many cases, if the work performed on each list element is expensive enough.

The original implementation of lists:map/2 is

```
map(F, L)  when is_function(F, 1), is_list(L) ->
   [F(X) || X <- L].
```

(The actual implementation in lists.erl is a bit different, but it *could* look like this.)

The function body uses a list comprehension, which reads as: ”Return the list of F(X), where X is taken from the list L.”

For example, lists:map(fun(X) -> X+1 end, [1,2,3]) returns [2,3,4].

In our first parallel map, we iterate throug the list L, and spawn a process for each element. Each process will evaluate F(X) (with its specific value of X) and send the result back to the parent process; after this, it dies, since there is nothing more to evaluate. The operation, expressed as a list comprehension, will result in a list of process identifiers – one for each element in the list. The parent process will then iterate through the list of Pids and receive the result messages, producing a list of the evaluated result for each X. The result list will be in order, since the collection uses ”selective receive” for each Pid. No matter in which order the result messages arrive, the parent process will match the result from the first Pid, and then the next, etc.

We thus preserve order, but there are other differences (other than that the whole map operation may run faster or slower):

-If either application F(X) raises an exception, the operation will hang, since the parent process will never receive a message.

-If the application F(X) has side-effects, or depends on the process environment (e.g. a mnesia transaction or the process dictionary), the result of the map is undefined.

More pmap alternatives

Catches errors and includes them in the map result:

```
pmap1(F, L) ->
   Parent = self(),
   Pids = [spawn(fun() ->
                    Parent ! {self(), catch F(X)}
                end) || X <- L],
   [receive {Pid,Res} -> Res end || Pid <- Pids].
```


We can try to fix our pmap so that it doesn't hang if an evaluator process crashes.

The simplest way is to insert a catch, i.e. Parent ! {self(), catch F(X)}.

This has the obvious drawback that the map operation, rather than failing like the original map would, now includes error values in the list, leaving it up to the programmer to sort things out.

More pmap alternatives

Catches first error, terminates rest and raises exception:

```
pmap2(F, L) ->
    await2(spawn_jobs(F, L)).

spawn_jobs(F, L) ->
    Parent = self(),
    [spawn(fun(X) -> Parent ! {self(),catch {ok,F(X)}}
     || X <- L].

await2([H|T]) ->
    receive
      {H, {ok, Res}} -> [Res | await2(T)];
      {H, {'EXIT',_} = Err} ->
          [exit(Pid,kill) || Pid <- T],
          [receive {P,_} -> ok after 0 -> ok end || P <- T],
          erlang:error(Err)
    end;
await2([]) -> [].
```


We amend the function some more, first changing the catch pattern so that we can distinguish valid results from invalid results (using catch {ok, F(X)}, which is an old Erlang trick).

In our collection function, we unwrap valid results. If we encounter an invalid result, we send 'kill' messages to the rest of the pids and raise an exception. But we must also flush any results that may already have arrived from the remaining processes.

(Later on in the presentation, we will learn that this is a dubious approach, but based on what we know so far, it seems to work.)

At this point we're beginning to suspect that there are other things we haven't considered. What if, for example, the evaluator processes are killed by an exit message? (This would be silly, but can, and therefore probably will, happen). If this happens, the catch is ineffective, and we'd have to use a monitor to detect if the evaluator crashes.

Pmap discussion

- Extra overhead for each function application:
  - (spawn + msg send + msg receive + process exit)
- Erlang processes are reasonably lightweight
  - But carry lots of debugging info etc.
  - Message passing is (normally) copying
    ...worst case, flattening
- Measure to see if granularity is suitable
- Exception handling semantics force tradeoffs
- Preserving order is easy, but has a cost

- ...but map is inherently sequential!
- Use SMP mainly for naturally concurrent patterns

To summarize, we noticed that a naive parallelization of a sequential function can be somewhat problematic. To go with the Erlang grain, we really do need to consider the effect of exceptions, since Erlang relies heavily on dynamic typing and run-time pattern-matching. Remember that exceptions should be expected, and the software should know how to react. Hanging forever is usually an unwanted result.

We know that our pmap doesn't cover all aspects, but even so, we've introduced considerable overhead – not just spawning processes and sending messages, but also relying on selective receive to preserve order, at quadratic complexity. We could do better, complexity-wise, by sorting the results ourselves, but this would likely be slower for small lists, and would of course complicate the implementation further.

Once we're satisfied with the workings of our pmap, we need to measure to see where the break-even point is between using the sequential map and using our parallel one. We might decide that the overhead is too high for a ”safe” pmap, and decide to go with a simpler version (perhaps we ”know” that there will be no exceptions).

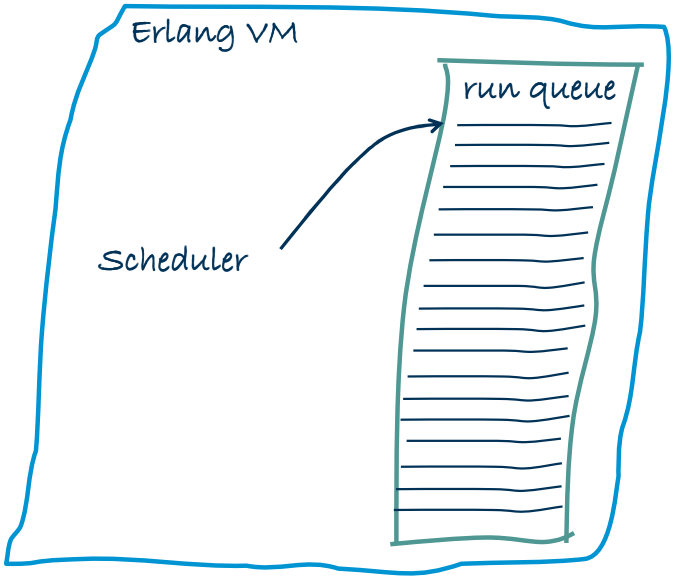
In either case, parallelizing sequential operations isn't as straightforward as it may seem, and Erlang doesn't always yield good result.

We remind ourselves that the idea behind SMP Erlang is to speed up programs that are written in the traditional Erlang style.

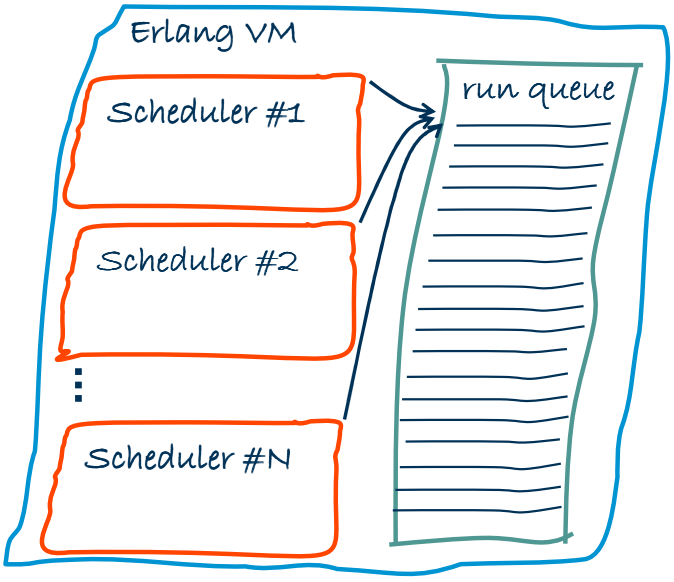
Benchmarks & Internals

ERICSSON
TAKING YOU FORWARD

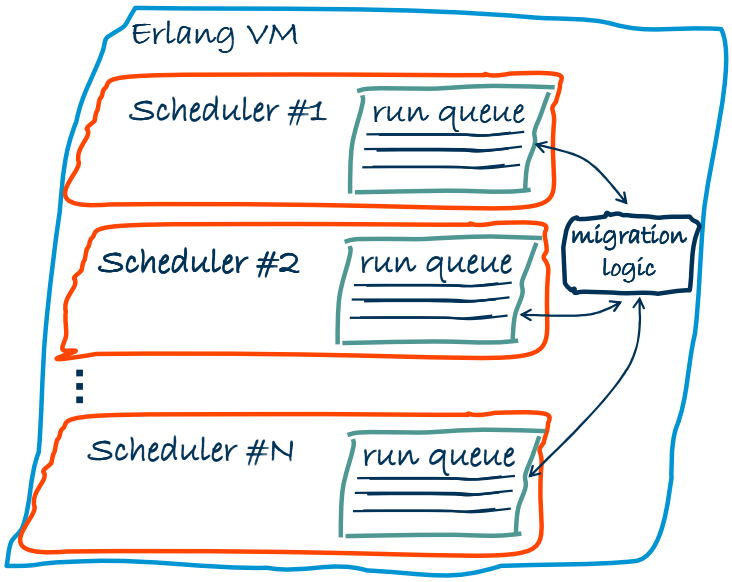
non-SMP VM



Current SMP VM

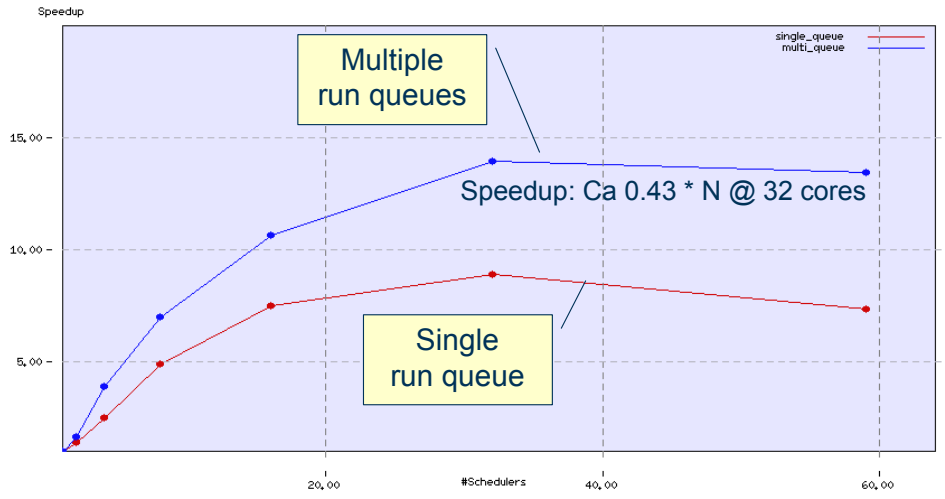


SMP VM in Erlang/OTP R13



Some benchmarks

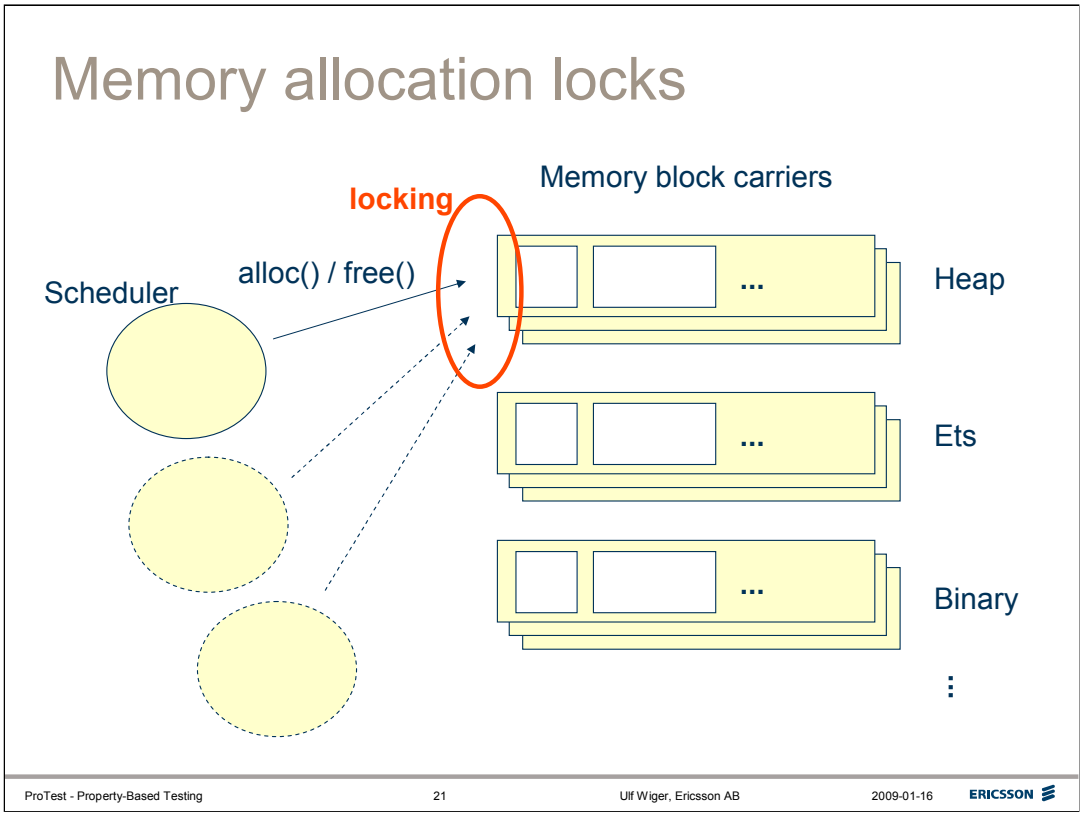
- Speedup of ”Big Bang” on a Tilera Tile64 chip (R13α)
  - 1000 processes, all talking to each other



ProTest - Property-Based Testing 20 Ulf Wiger, Ericsson AB 2009-01-16 ERICSSON

Returning to naturally concurrent patterns, we look at the famous ”Big Bang” benchmark. This benchmark starts 1000 processes and lets them all talk to each other. Not very realistic, perhaps, but it does show some interesting features of the virtual machine. For one thing, this benchmark is unusual in that it can give better than 100% speedup per core (at least for a few cores), for various reasons.

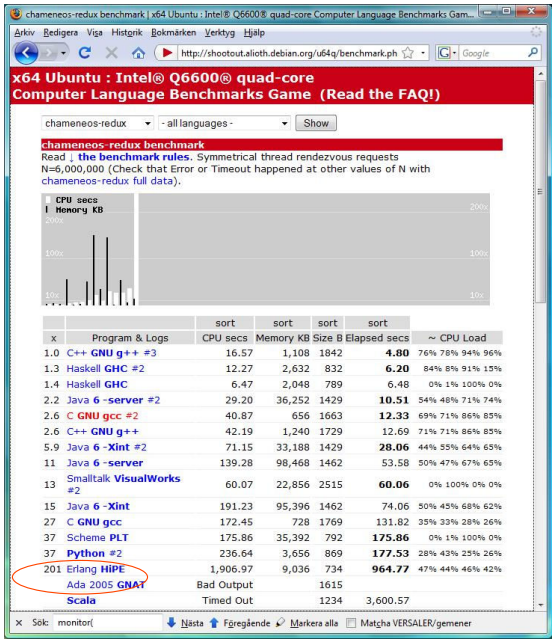
In this case, the Big Bang benchmark was run on an experimental Tilera Tile64 chip. This chip architecture has an on-chip message-passing switch of sorts, but this is likely of little use to the Erlang VM, since it uses POSIX threads and shared memory. Even so, we get pretty impressive speedup up to 32 cores. After that, the program actually runs slower. Closer analysis indicates that memory allocation locks start becoming a problem with many cores. The OTP team has a beta version of the emulator using multiple run queues (one per scheduler thread), rather than a single run queue from which all schedulers fetch jobs. The performance of multiple run queues (blue) vs. single run queue (red) is shown in the graph. Multiple run queues are almost always better, but they also facilitate other optimizations, such as different structuring of memory in order to reduce lock contention. This has not been implemented yet.



The Erlang VM uses ”carriers” in order to reduce memory fragmentation in long-running systems.

In the current SMP VM, all scheduler threads use the same carriers, and there’s no way to fix an erlang process to a specific scheduler. This leads to frequent locking of memory carriers, which becomes the dominating factor as the number of cores increase (e.g. beyond 32 cores).

A really bad benchmark result



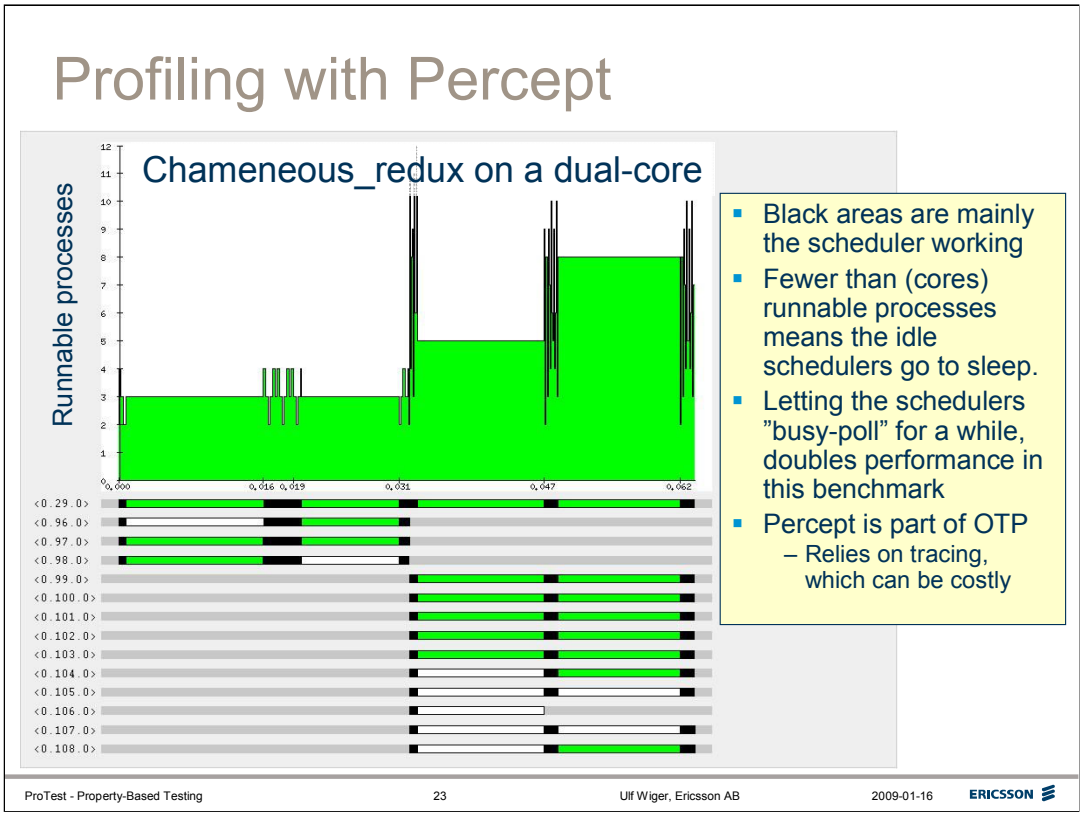
- Chameneos Redux in The Shootout
- Gets worse, the more cores are added...
- Successful entries:
  - C++ (pthread_spinlock)
  - Haskell (MVar)
  - Java (synchronized + busy loop wait)

There are other benchmarks that are less flattering to Erlang. One of the worst known to-date is the ”chameneos_redux” in the Computer Language Shootout. It is basically centered around rendezvous, and a very poor match for message-passing concurrency (esp of the granularity that Erlang supports). One may note that the Scala entry, using much the same approach as Erlang, timed out...

We note that the best entries use some form of shared-memory mutex (spinlocks, MVars, etc.) The difference in performance is staggering.

To add insult to injury, the OTP team has observed that this benchmark runs slower the more cores you throw at it.

On the next slide, we will try to see what is going on.



OTP comes with an SMP profiling tool called Percept. It relies on tracing, and can hard to use in large systems (esp. under some load). Short snapshots are recommended, since one may otherwise gather so much data that percept cannot handle it.

In the chart above, we see black areas. These are intervals were no useful work is done, and the time is taken up entirely by the scheduler. We can also see that the program causes frequent jumps in the number of runnable processes. The scheduler threads will be put to sleep by the Linux kernel if there are no available jobs, and a kernel call is needed to wake them up again. Therefore, if a program quickly moves beween having (> cores) runnable processes and having (< cores), some scheduler threads will constantly go to sleep and wake up, causing tremendous overhead. Experiments with letting the schedulers busy-poll for work helps this benchmark a lot, but of course raises total CPU consumption (something that many of the other benchmarks do as well, but it’s perhaps not a good thing to always do this.)

For good scalability in SMP Erlang, it is good to always have enough runnable processes to keep all schedulers busy.

Improvements in the next release

- Multiple scheduler queues
- Optimized ETS access

Possible future improvements

- Shared-heap clusters
- Parallel eval (e.g. similar to F#)
- Fully asynchronous message passing
  (EXITs are already fully asynchronous)
- ...

The $10,000 Question

How do you test and debug
your program on multicore?

ERICSSON
TAKING YOU FORWARD

Current situation

- Absense of shared memory makes writing and debugging concurrent programs easier than otherwise.
- SMP is in some ways similar to distributed programs (which, admittedly, is hard).
- Selective msg reception → simpler state machines.
- Erlang has great tracing facilities for debugging concurrent programs. This helps in SMP also.

- But this is not nearly good enough!

Erlang programmers like to brag about how running on SMP is just like running on non-SMP. While it is true that Erlang programmers are spared from some of the nastiest smp-related bugs, debugging an SMP system is still more difficult than debugging a non-SMP system.

Going from non-SMP to SMP, one will have to contend with true concurrency and non-determinism. In non-SMP, the scheduler will only let one process run at a time, so there is no actual parallelism, and the scheduler is actually very predictable.

Going to a distributed setting introduces a whole set of additional challenges:

-the communication medium can fail

-message passing is delayed (rather than being instantaneous)

-nodes can fail and reappear, new nodes can be added and nodes can be removed, in a running system

Compared to this, debugging SMP is relatively easy...

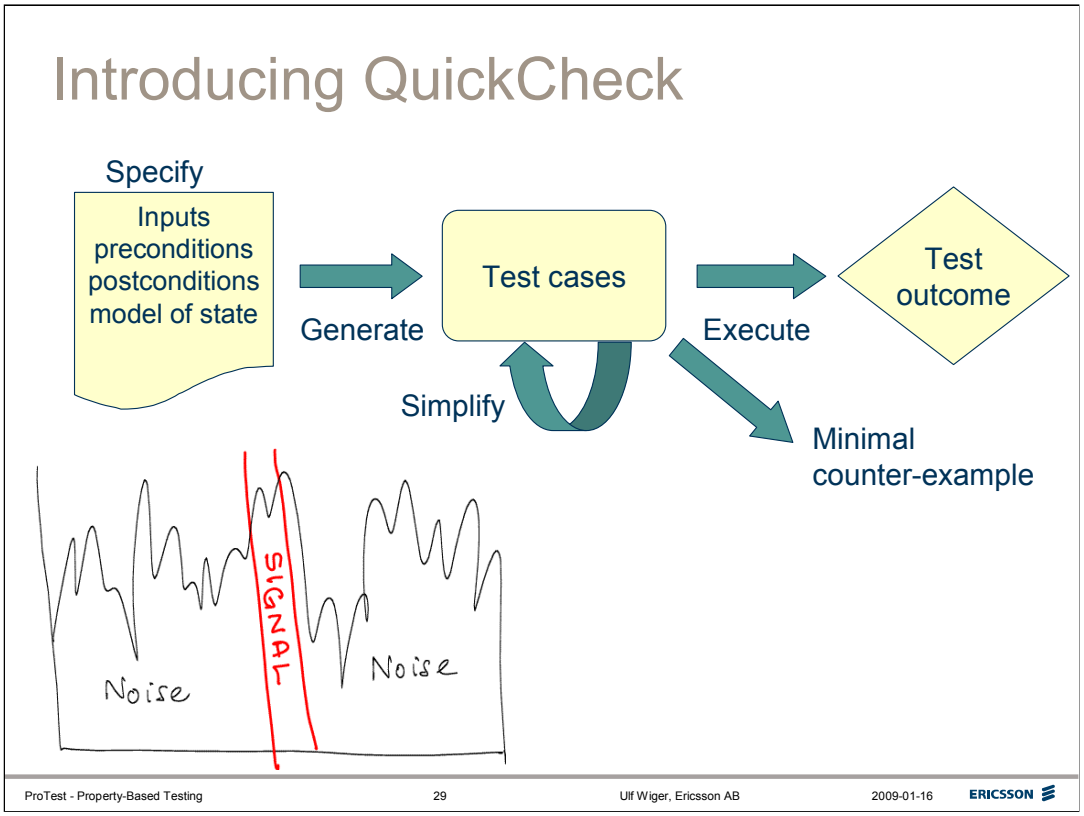
Still, we should remember that the state of debugging is by no means satisfactory. Some bugs are notoriously difficult to find, and of these, some get much more likely (and more difficult to debug!) in SMP.

Hypothesis

- Timing-related bugs are often triggered only when running large systems
- But this is mainly because timing is different in large systems, compared to unit test
- If we can control timing aspects in unit test, many concurrency bugs can be found by running small examples

Among the very trickiest bugs are timing-related bugs. Typically, a component may pass unit test, where (hopefully) all the foreseen combinations of events are tested. Then, when running a larger system, timing aspects may be altered in ways that were *not* foreseen, and scary things happen. The system may become unresponsive, crash, or perhaps start behaving erratically.

We hypothesize that these bugs can often be found in unit test, if we can only find a way to play around with timing conditions there. The next challenge becomes to analyze the error...



QuickCheck can be described as ”Property-based testing”

•A model is used to describe how to stimulate the component, and how it is supposed to react.

•QuickCheck then generates random combinations of input and checks the results against the model.

•If a discrepancy is found, QuickCheck simplifies the input (reducing values, removing elements, etc.), in order to find a *minimal* counter-example.

•When testing stateful components, QuickCheck maintains a logical state and selects input that is valid for each state.

The simplification step is vital, and addresses the problem that random testing normally generates a lot of noise – it can be very difficult to see which part of the random test data that actually contributes to the error. Through the simplification process, QuickCheck can be seen as extracting the ”error signal” from out of the noise.

In fact, to the programmer familiar with QuickCheck’s simplification heuristics, the minimal counter-example can provide many hints about the likely cause of the error.

QuickCheck example

A simple property

```
prop_lists_delete() ->
  ?FORALL(I, int(),
    ?FORALL(List, list(int()),
          not lists:member(
                I, lists:delete(I,List)))).
```

Test run

```
1> eqc:quickcheck(example:prop_lists_delete()).
.......................................
Failed! after 42 tests
-8
[5,-13,-8,-8,-9]
Shrinking.................(16 times)
-8
[-8,-8]
false
```


The property above reads as:

”For all I (of type integer), it should hold that
for all List (of type list of integers),
deleting I from the list List
means that I is not a member of List”

Testing the property, QuickCheck quickly finds a counter-example – the list [5,-13,-8,-8,-9].

It is not immediately obvious why this example fails, but after the simplification (”shrinking”), we are left with a list of two elements [-8, -8].

We can deduce that the list must have at least two elements (or QuickCheck would have removed one), with the same value (or QuickCheck would have reduced one of them further.

Indeed, lists:delete(Value, List) removes only the first occurence of Value, so if there are more than one in the list, it will not be true that Value is not a member of the list.

Case study: proc_reg

- Extended process registry for Erlang*
- Early (2005) prototyped used tricks with ETS to ensure mutual exclusion without a server process
- An experiment with QuickCheck revealed a bug, but we couldn't diagnose it
- So I discarded that code. The gen_server-based replacement has now been used in the field for 3 years.
- We tried again with a new experimental QuickCheck (Part of the ProTest EU project)

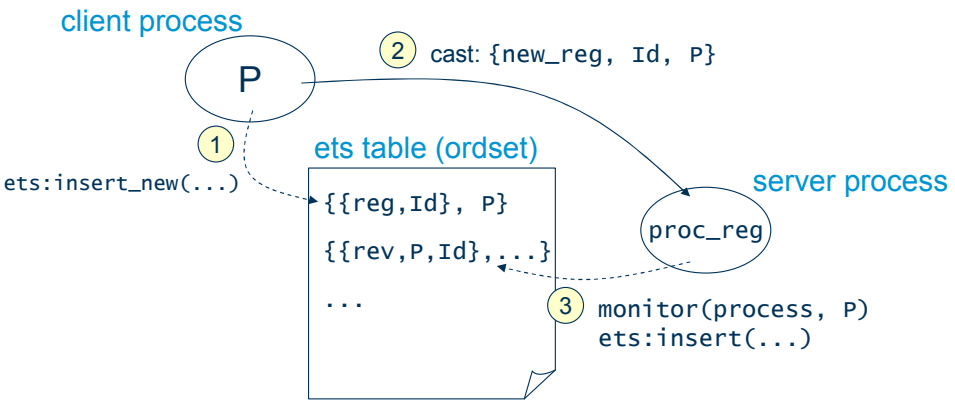
* Wiger: "Extended Process Registry for Erlang" ACM SIGPLAN Erlang Workshop 2007

The idea behind the extended process registry (proc_reg) was to capture a common pattern in our products. We were regularly implementing different mapping functions between some form of context and the corresponding process(es). A general form of index seemed useful, one where a process could be associated with any term either unique or not.

The first proc_reg implementation included an optimization in order to avoid relying on a central server for all registrations (in line with the philosophy of "first optimize, then make it work"). A fairly ambitious test suite ensure that the code worked, after having revealed a number of strange corner cases due to subtle scheduler behaviour.

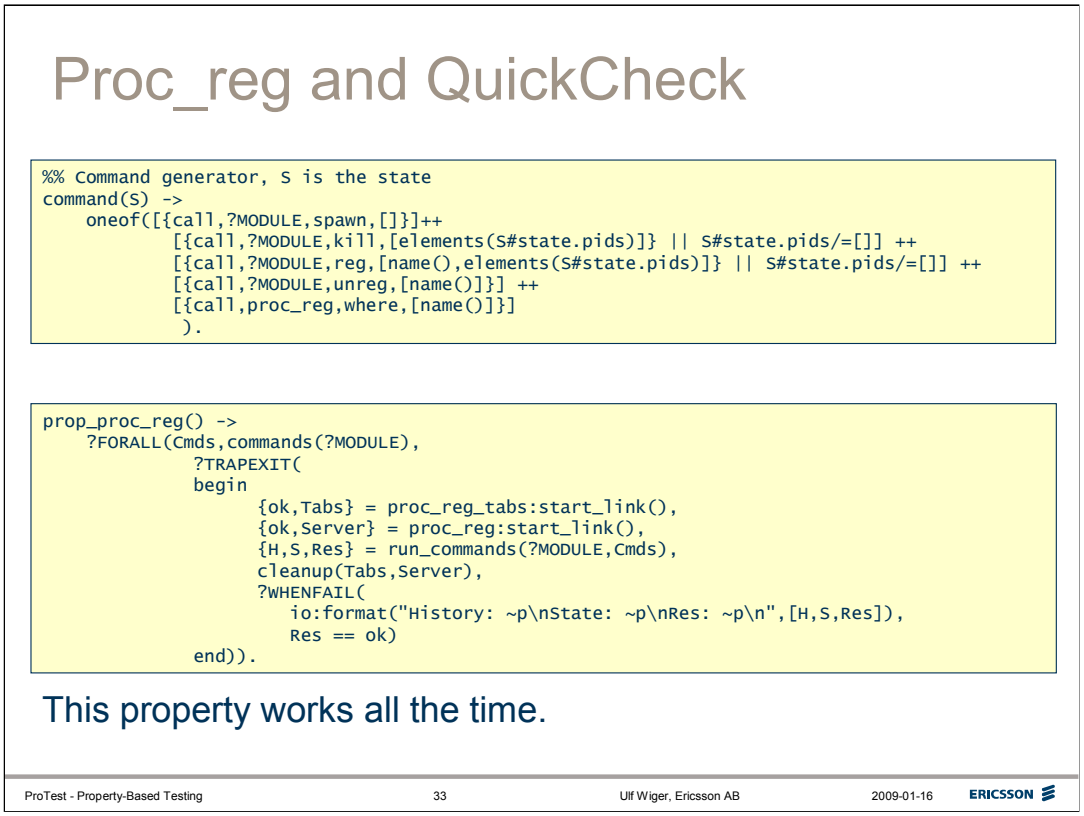
When we later needed some small, but still interesting, concurrent program for experiments with QuickCheck, proc_reg seemed just right. At the time, QuickCheck had no real facilities for testing concurrent programs, so quite a few contortions were required – but in the end, QuickCheck was able to find two strange aspects of proc_reg. The first was that it allowed for registration of dead processes. We considered this a bug until we discovered that Erlang's built-in registration functions behaved in the same way! The other issue was a process crash, which took many hours to understand. We finally concluded that the optimization in proc_reg still had some race condition, even though we didn't fully understand it at the time. While analyzing the problem, we wrote a reference implementation based on a central registry process. This implementation passed all tests, so I finally decided to throw away the optimized version and go with the (only slightly slower) safe version. This software has now been running in commercial products for ca 3 years.

Short intro to proc_reg



- Key API:

```
reg(Id, Pid) -> true  | badarg()
unreg(Id)    -> true  | badarg()
where(Id)    -> pid() | undefined
```



This slide shows some of the QuickCheck specification for proc_reg.

The **command(S)** function selects a valid command given the current state. In this case, the command is one of the commands (spawn, kill, reg, unreg, where). Kill can only be selected if there are registered processes (and one of these processes will then be selected to be killed). The reg command can only be selected if there are processes to register.

The **prop_proc_reg()** function describes a ”property”. It says ”for all runs of commands, generated using the specification ?MODULE (the current module), we should (while trapping exits) start proc_reg, run the commands, and cleanup; and the result of running the commands should be ’ok’”.

For this to be effective, we also need to check specify what it means for a command to be successful (next slide).

QuickCheck post-conditions

```
postcondition(S,{call,_,where,[Name]},Res) ->
    Res == proplists:get_value(Name,S#state.regs);
postcondition(S,{call,_,unreg,[Name]},Res) ->
    case Res of
          true ->
              unregister_ok(S,Name);
          {'EXIT',_} ->
              not unregister_ok(S,Name)
    end;
postcondition(S,{call,_,reg,[Name,Pid]},Res) ->
    case Res of
          true ->
              register_ok(S,Name,Pid);
          {'EXIT',_} ->
              not register_ok(S,Name,Pid)
    end;
postcondition(_S,{call,_,_,_},_Res) ->
    true.

unregister_ok(S,Name) ->
    lists:keymember(Name,1,S#state.regs).

register_ok(S,Name,Pid) ->
    not lists:keymember(Name,1,S#state.regs).
```

- Surprisingly few
- This is the heart of the specification

QuickCheck post-conditions are evaluated for the return value of each command. A post-condition can be specified for each (command, result, state) triple, using Erlang pattern-matching. The state in this case is QuickCheck's logical representation of what the state of the code under test should be (the specifications for state transitions is not shown here).

In this case the **where** command should return the Pid of the process having registered the name. The **unreg** command should return true if there was a live process associated with the name, and exit otherwise. The **reg** command should return true if the name is not already taken, and exit otherwise.

The specification so far doesn't make use of parallelism, and so, for proc_reg, it succeeds all the time.

Parallelizing the property

```
prop_parallel() ->
    ?FORALL(PCmds={_,{_ACmds,_BCmds}},parallel2:pcommands(?MODULE),
              ?ALWAYS(5,
              begin
                    {ok,Tabs} = proc_reg_tabs:start_link(),
                    {ok,Server} = proc_reg:start_link(),
                    {H,AB,Res} = parallel2:run_pcommands(?MODULE,PCmds),
                    kill_all_pids({H,AB}),
                    cleanup(Tabs,Server),
                    ?WHENFAIL(
                       io:format("Sequential: ~p\nParallel: ~p\nRes: ~p\n",[H,AB,Res]),
                       Res == ok)
              end)).
```

- QuickCheck tries different ways to parallelize some commands without violating preconditions
- The property fails if there is no possible interleaving of the parallel commands that satisfy the postconditions
- Shrinking is not deterministic, but works surprisingly well...

It turns out that we can take the very same specification and simply enhance the property a bit to make QuickCheck explore parallelism. This requires SMP Erlang on a multi-core computer.

The ?ALWAYS(5, Expr) is interesting. Since re-testing the sequence may result in different timing behaviour, we specify that a command sequence must succeed 5 times in a row in order to be considered successful.

Using a custom scheduler

```
prop_scheduled(Verbose) ->
    ?FORALL(PCmds={_,{_ACmds,_BCmds}},parallel2:pcommands(?MODULE),
    ?ALWAYS(5,
    ?FORALL(Seed,seed(),
       begin
           L = scheduler:start([{seed,Seed},{verbose,Verbose}],
                 fun() ->
                         {ok,Tabs} = proc_reg_tabs:start_link(),
                         {ok,Server} = proc_reg:start_link(),
                         {H,AB,Res} = parallel2:run_pcommands(?MODULE, PCmds),
                         kill_all_pids({H,AB}),
                         cleanup(Tabs,Server),
                         {H,AB,Res}
                 end),
           delete_tables(),
           {H,AB={A,B},Res} = proplists:get_value(result,L),
           ?WHENFAIL(
             ..., Res == ok)
       end))).
```

- The code under test must first be instrumented
- The scheduler controls important scheduling events, and is deterministic

There is also a special scheduler designed to work with QuickCheck. Since it is written in Erlang, the code under test must be instrumented in order to make use of it. This scheduler uses a pseudo-random sequence to generate ”interesting” scheduling variations in a fully repeatable fashion. Since it is deterministic, QuickCheck can simplify the test case.

Bug # 1

```
{[{set,{var,6},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,8},{call,proc_reg_eqc,kill,[{var,6}]}}],
 {[{set,{var,11},{call,proc_reg_eqc,reg,[a,{var,6}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[a,{var,6}]}}]}]}
{2829189918,7603131136,617056688}
Sequential: [{{state,[],[],[]},<0.10470.0>},
             {{state,[<0.10470.0>],[],[]},ok},
             {{state,[<0.10470.0>],[],[<0.10470.0>]},ok}]
Parallel: {[{{call,proc_reg_eqc,reg,[a,<0.10470.0>]},
             {'EXIT',{badarg,[{proc_reg,reg,2},
                              {proc_reg_eqc,reg,2},
                              {parallel2,run,2},
                              {parallel2,'-run_pcommands/3-fun-0-',3}]}}}],
          [{{call,proc_reg_eqc,reg,[a,<0.10470.0>]},true}]}
Res: no_possible_interleaving
```

"Sequential prefix"

Parallel component

- This violates the specification
  - Registering a dead process should always return 'true'
- Absolutely minimal counter-example, but still hard to understand
- QuickCheck can also output a .dot file...

Running the test with parallelization, QuickCheck will find a counter-example. It does not always shrink down perfectly, but re-running the test a few times, we can fairly quickly come up with a nice counter-example:

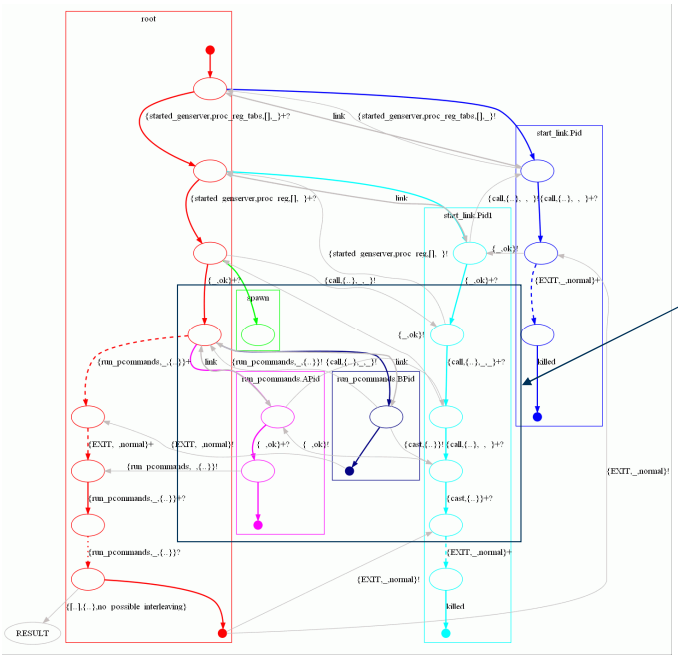
-spawn a process

-kill the process just spawned

-in parallel, register the (dead) process as 'a' from two different client processes

When this is run, occasionally, the **reg** command will return 'true for one process, but exit for the other.

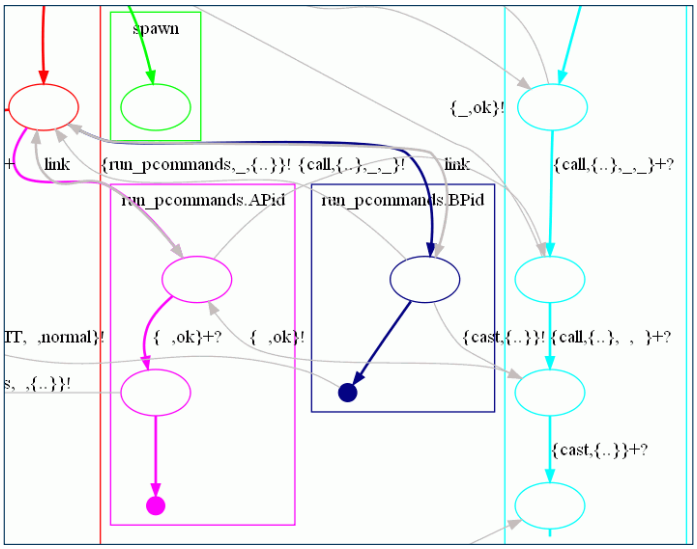
Generated state chart



- QuickCheck-generated .dot file, visualized with GraphViz
- We'll zoom in on the interesting part

Process link
Message
Process
State

State chart detail



- `BPid`'s registration succeeds, and it sends a cast to the admin server
- `APid` finds that the name is taken, but by a dead process – it asks the server for an audit (the call)
- But the call reaches the server before the cast ("multi-node semantics")
- The server uses a reverse lookup (updated when the cast is received) to find the name.
- Since the messages arrived in an unexpected order, the name remains after the audit, and `APid` fails unexpectedly.

Bug fix #1

```
do_reg(Id, Pid) ->
    Now = erlang:now(),
    RegEntry = {Id, Pid, Now},
    case ets:insert_new(proc_reg, RegEntry) of
          false ->
              false;
          true ->
              ?gen_server:cast(proc_reg, {new_reg, Id, Pid, Now}),
              true
    end.
```

```
do_reg(Id, Pid) ->
    Now = erlang:now(),
    RegEntry = {{reg,Id}, Pid, Now},
    RevEntry = {{rev,Pid,Id},undefined,undefined},
    case ets:insert_new(proc_reg, [RegEntry,RevEntry]) of
        false ->
            false;
        true ->
            ?gen_server:cast(proc_reg, {new_reg, Id, Pid, Now}),
            true
    end.
```

- Insert a dummy reverse mapping immediately
- ets:insert(Objects) is atomic
- The server simply overwrites the dummy entry

Bug # 2

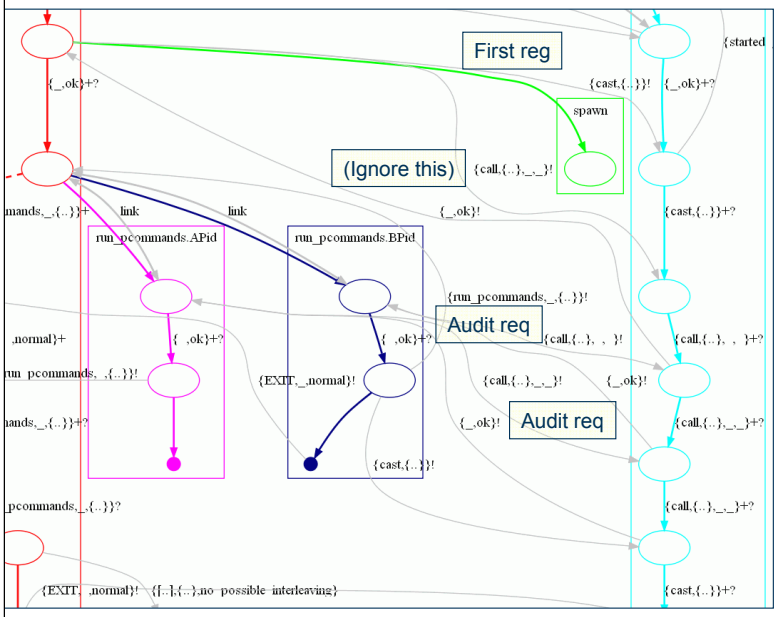
```
Shrinking............(12 times)
{[{set,{var,5},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,23},{call,proc_reg_eqc,kill,[{var,5}]}},
  {set,{var,24},{call,proc_reg_eqc,reg,[b,{var,5}]}}],
 {[{set,{var,25},{call,proc_reg_eqc,reg,[b,{var,5}]}}],
  [{set,{var,26},{call,proc_reg_eqc,reg,[b,{var,5}]}}]}]}
{-9065357021,-6036499020,-6410890974}
Sequential: [{{state,[],[],[]},<0.26845.2>},
             {{state,[<0.26845.2>],[],[]},ok},
             {{state,[<0.26845.2>],[],[<0.26845.2>]},true},
             {{state,[<0.26845.2>],[],[<0.26845.2>]},ok}]
Parallel: {[{{call,proc_reg_eqc,reg,[b,<0.26845.2>]},
             {'EXIT',{badarg,[{proc_reg,reg,2},
                              {proc_reg_eqc,reg,2},
                              {parallel2,run,2},
                              {parallel2,'-run_pcommands/3-fun-0-',3}]}}}],
          [{{call,proc_reg_eqc,reg,[b,<0.26845.2>]},true}]}
Res: no_possible_interleaving
```

”Sequential prefix”

Parallel component

- Still problematic to register a dead process!
- Let’s look at the graph...

Chart detail, Bug # 2



- Since the name is already registered, both `APid` and `BPid` request an audit
- Both then assume that it will be ok to register, but one still fails.
- This is ok (valid race condition), but not if it's a dead process!!!!

Bug fix # 2

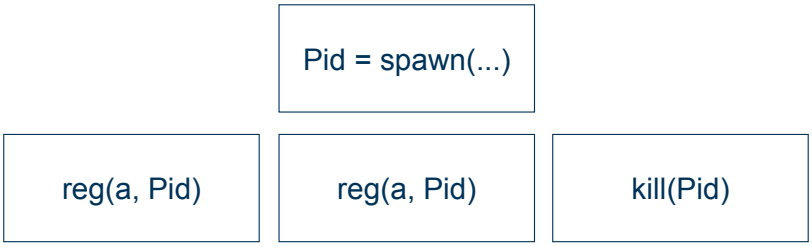
```
do_reg(Id, Pid) ->
    Now = erlang:now(),
    RegEntry = {{reg,Id}, Pid, Now},
    RevEntry = {{rev,Pid,Id},undefined,undefined},
    case ets:insert_new(proc_reg, [RegEntry,RevEntry]) of
        false ->
            false;
        true ->
            ?gen_server:cast(proc_reg, {new_reg, Id, Pid, Now}),
            true
    end.
```

```
do_reg(Id, Pid) ->
 case is_process_alive(Pid) of
          false ->
              where(Id) == undefined;
          true ->
              Now = erlang:now(),
              ...
    end.
```

- Don’t ever insert a dead process in the registry (duh...)
- After this fix, the code passed 20 000 tests with the custom scheduler.

But wait...

The code still has a race condition! (...or?)



- What does the specification say about a process dying while it's being registered?
- If it were ok to register the same (Name, Pid) twice, it would be easy (but the register/2 BIF doesn't allow it...)
- For now, QuickCheck only works with two parallel sequences.
- Alternative approach: Require processes to register themselves!

Single-node vs Multi-node semantics

- Within an Erlang node, message delivery is always immediate (”single-node semantics”)
- In a distributed system, delivery is asynchronous
- Atomic message delivery is problematic on many-core

- Erlang never did guarantee immediate delivery
- But it’s very easy to make this assumption

Consequence for the Programmer

- SMP Erlang is likely to become even more asynchronous
- This will increase the risk of very subtle race conditions
- Hopefully, we can find them in unit test with tools like QuickCheck