Parallel Erlang - Speed beyond Concurrency

Experience from Parallelizing Dialyzer

Stavros Aronis

UPPSALA
UNIVERSITET

RELEASE

Dialyzer

- Static analysis tool included in Erlang/OTP
- 30,000 lines of Erlang code

```
$ dialyzer --build_plt --apps erts kernel stdlib
  Compiling some key modules to native code... done in 0m12.27s
  Creating PLT /home/stavros/.dialyzer_plt ...
Unknown functions:
  ...
Unknown types:
  ...
 done in 0m26.42s
done (passed successfully)
```

```
$ dialyzer my_module.beam
  Checking whether the PLT is up-to-date... yes
  Proceeding with analysis... done in 0m0.38s
done (passed successfully)
```

Obligatory preaching!

Preaching!

> [...] the real value of static analysis for correctness issues is its ability to find problems **early** and **cheaply**, rather than in finding subtle but serious problems that cannot be found by other quality assurance methods.

- The Google FindBugs Fixit, Nathaniel Ayewah and William Pugh, 2010

Preaching!

> The main targets this Makefile supports are as follows:
>
> ...
>
> dialyzer: Build the dependency PLT and run dialyzer on the project

- Universal Makefile for Erlang Projects That Use Rebar, 4 Jun 2013

> "You MUST ensure that all commits pass all tests and do not have extra Dialyzer warnings."

- Cowboy's CONTRIBUTING.md, Loic Hoguin

Preaching!

> Dialyzer is **never** wrong.

- Fact

But...

- ... a tool is useful *if* you use it often
- ... you should use Dialyzer *at least* before you commit
- ... it should be **easy** and **fast**
- ... on modern, multicore machines

Let's make it parallel!

Internals of Dialyzer

Internals of Dialyzer

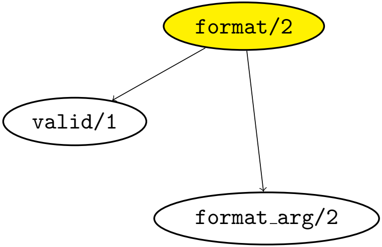
Original developers: Tobias Lindahl & Kostis Sagonas

- Type inference: A signature (spec) is inferred for each function
  `e.g. fun(atom(), [_]) -> 42 | 'ok' | {_,_}.`
- Two phases:
  - **bottom-up analysis:** from callees to callers (*typesig*)
    Find *all* the acceptable arguments and possible results
  - **top-down analysis:** from callers to callees (*refine*)
    Refine types, using dataflow (for the non-exported functions)
- Repeatedly, until fixpoint.
- **Final pass:** use types to report discrepancies.

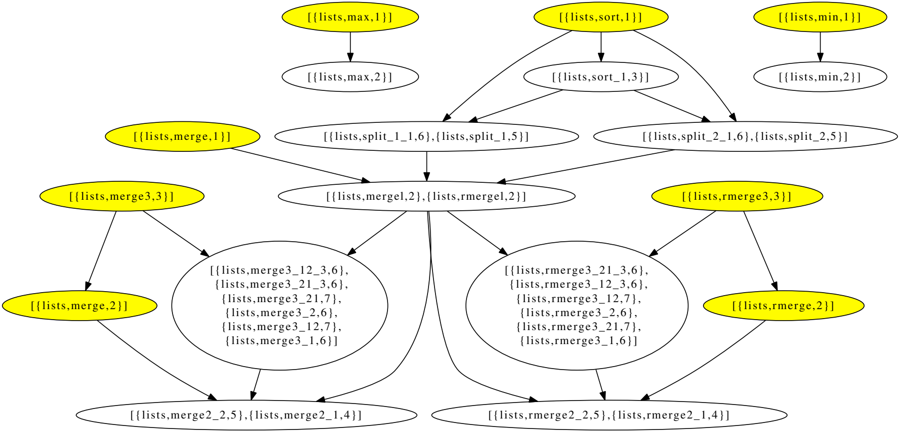
Example

```erlang
-module(example).

-export([format/2]).

format(Arg1, Arg2) ->
  case valid(Arg1) of
    true ->
      format_arg(valid, Arg2);
    false ->
      throw(invalid);
    undefined ->
      throw(unknown)
  end.

valid(Arg) when is_atom(Arg) -> true;
valid(_) -> false.

format_arg(Tag, Arg) -> {Tag, Arg}.
```

Callgraph:



Closer to reality – SCCs



[{lists,max,1}]

[{lists,sort,1}]

[{lists,min,1}]

[{lists,max,2}]

[{lists,sort_1,3}]

[{lists,min,2}]

[{lists,merge,1}]

[{lists,split_1_1,6},{lists,split_1,5}]

[{lists,split_2_1,6},{lists,split_2,5}]

[{lists,merge3,3}]

[{lists,mergel,2},{lists,rmergel,2}]

[{lists,rmerge3,3}]

[{lists,merge,2}]

[{lists,merge3_12_3,6},
{lists,merge3_21_3,6},
{lists,merge3_21,7},
{lists,merge3_2,6},
{lists,merge3_12,7},
{lists,merge3_1,6}]

[{lists,rmerge3_21_3,6},
{lists,rmerge3_12_3,6},
{lists,rmerge3_12,7},
{lists,rmerge3_2,6},
{lists,rmerge3_21,7},
{lists,rmerge3_1,6}]

[{lists,rmerge,2}]

[{lists,merge2_2,5},{lists,merge2_1,4}]

[{lists,rmerge2_2,5},{lists,rmerge2_1,4}]

(Highlighted functions are exported)

Performance of sequential version

```
$ dialyzer --statistics <all apps in OTP>:

compile        :  114.67s ( 1493 modules)
prepare        :    4.83s
order          :   11.16s
typesig 1      : 1408.07s (97347    SCCs)
order          :    9.93s
refine 1       :  240.22s ( 1493 modules)
order          :   15.14s
typesig 2      : 2443.59s (80323    SCCs)
order          :    6.35s
refine 2       :  247.81s ( 1414 modules)
order          :    0.28s
typesig 3      :   95.45s ( 2429    SCCs)
order          :    0.12s
refine  3      :   28.99s (  203 modules)
[round 4 & 5]  :  < 0.50s
warning        :  308.26s ( 1493 modules)

       done in 82m29.87s
```

Spawn, spawn, spawn...

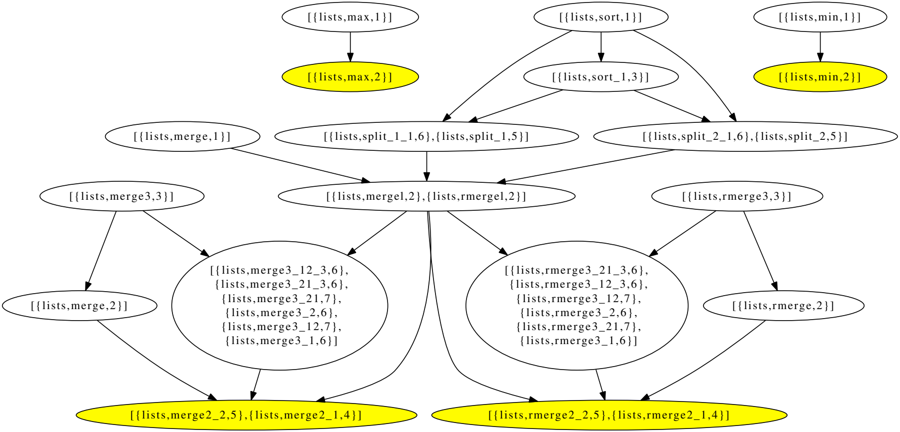
Distributing the work

The tasks for a “worker” are obvious:

- Prepare the code of a module
- Perform type analysis of an SCC
- Perform refinement of the functions in a module
- Scan a module for discrepancies

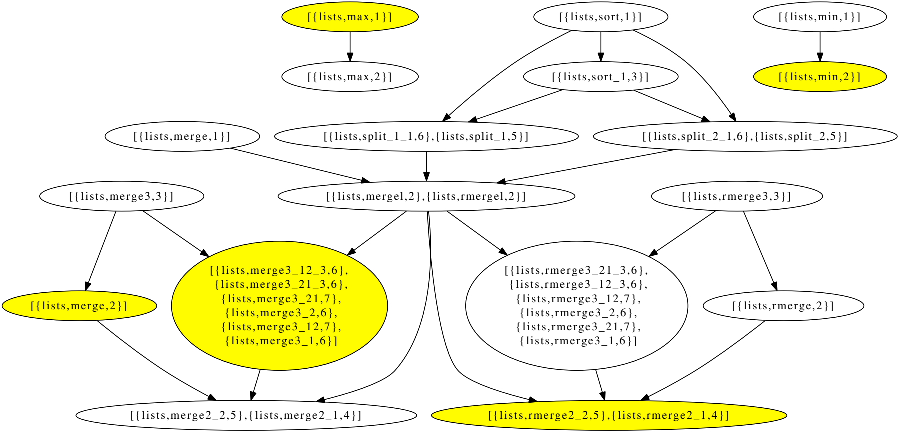
Workers should, however, respect the dependencies.

Coordination



(Highlighted SCCs are leaves of the callgraph)

Coordination



(After some have been analysed.)

Decision #1: Coordination

- Central "coordinator"?
  - Keep track of dependencies
  - Spawn workers when dependencies are satisfied
  - Bottleneck

Spawn, spawn, spawn!

- Distributed coordination:
  - Calculate and make available all dependencies in a public ETS table
  - Spawn **all** workers (`erl +P 1.000.000` !)
  - Each waits for a message from each dependency before it starts running
  - Some of them may be done before we finish spawning...
    (It's ok, sleep for a while)

Decision #2: Data sharing

- Data serving processes?
  - Linearization
  - Replication / Distribution → Too complex

Use more public ETS tables instead!

- Prepared code, dependencies, types are all in ETS
- Even for data from dependent processes?
  - Broadcast a type to *n* workers → Sequential
  - Just write it in ETS (with `write_concurrency`)
  - Everyone that needs it will read it (`concurrently`)

We are ready to go!

Sequential version

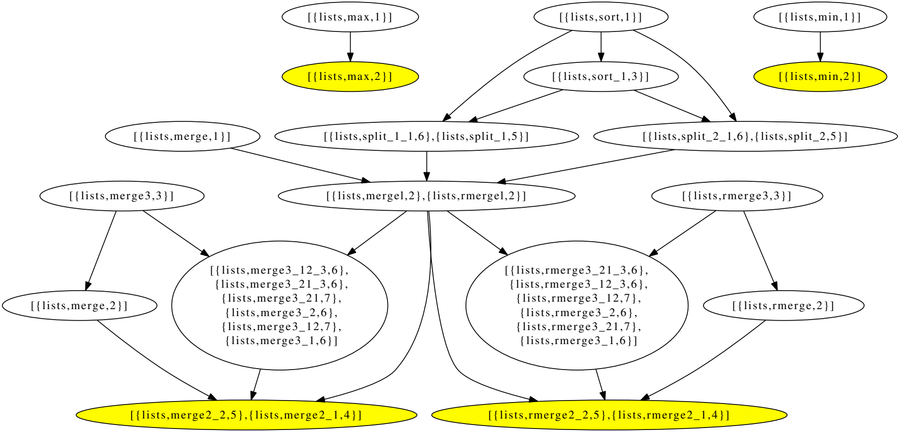
Suppose we just wanted to analyze leaf SCCs:

```erlang
sequential_analysis(SCCs, State) ->
  FoldFun = fun (SCC, Acc) -> find_type(SCC, Acc, State) end,
  Results = lists:foldl(FoldFun, [], SCCs),
  NewState = update_types(Results, State),
  ...

find_type(SCC, Acc, State) ->
  Code = retrieve_code(SCC, State),
  Type = analyze_code(Code, State),
  [{SCC, Type}|Acc].
```

Parallel version

```
parallel_analysis(SCCs, State) ->
  ParentPID = self(),
  FoldFun = fun (SCC, Counter) ->
                 spawn(fun () -> find_type(SCC, ParentPID, State) end),
                 Counter + 1
               end,
  Workers = lists:foldl(FoldFun, 0, SCCs),
  Results = receive_results(Workers, []),
  NewState = update_types(Results, State),
  ...

find_type(SCC, ParentPID, State) ->
  Code = retrieve_code(SCC, State),
  Type = analyze_code(Code, State),
  ParentPID ! {SCC, Type}.

receive_results(0, Acc) -> Acc;
receive_results(N, Acc) ->
  receive Result -> receive_result(N-1, [Result|Acc]) end.
```

Idle workers



Decision #3: Idle processes

- All workers are spawned right from the start
- Let them do preliminary tasks while waiting?

```
find_type(SCC, ParentPID, State) ->
  Code = retrieve_code(SCC, State),
  Type = analyze_code(Code, State),
  ParentPID ! {SCC, Type}.
```

Out of memory!

- Idle workers **must** not do *anything* until ready to run, in order to keep their heaps' size minimal
- `State` **must** contain the *bare* essentials.

Decision #4: Throttling

- When all dependencies have been satisfied let a worker run?

Out of memory!

- Erlang scheduling is preemptive
- Too many workers active → Too many half-finished jobs

- Allow only as many active workers as logical cores
- Erlang schedulers are efficient (≈100% CPU utilization when there are many ready workers)

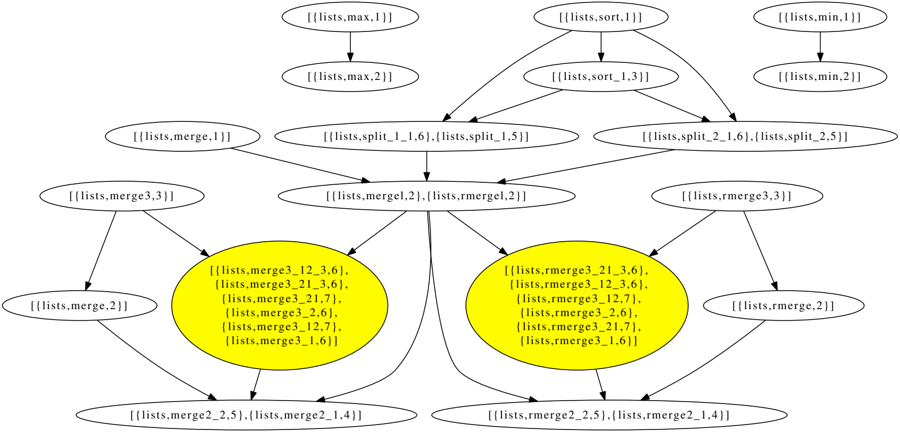
Decision #5: Granularity

- Does our parallel version perform well with any input?

Workers for big SCCs need more time!

- Split big SCCs into more workers...
- ... taking care of what is copied, of course!

Big SCCs



(Highlighted SCCs are “big”)

(The erl_parse module has much bigger...)

Decision #6: Sequential leftovers

- Initially we have a big callgraph with every function as a node
  - Filter out functions that have reached fixpoint (`digraph_utils:reaching/2`)
  - Graph condensation into SCCs (`digraph_utils:condensation/1`)

Expensive!

- Home made optimized version of the condensation algorithm
- The `digraph_utils` library is not really parallel...
- Reachability is ok for the time being

Was it easy?

- Already existing good structure
- Significant level of familiarity

- From 30,000 lines of Erlang code...

1,800 lines added, 1,000 lines deleted!

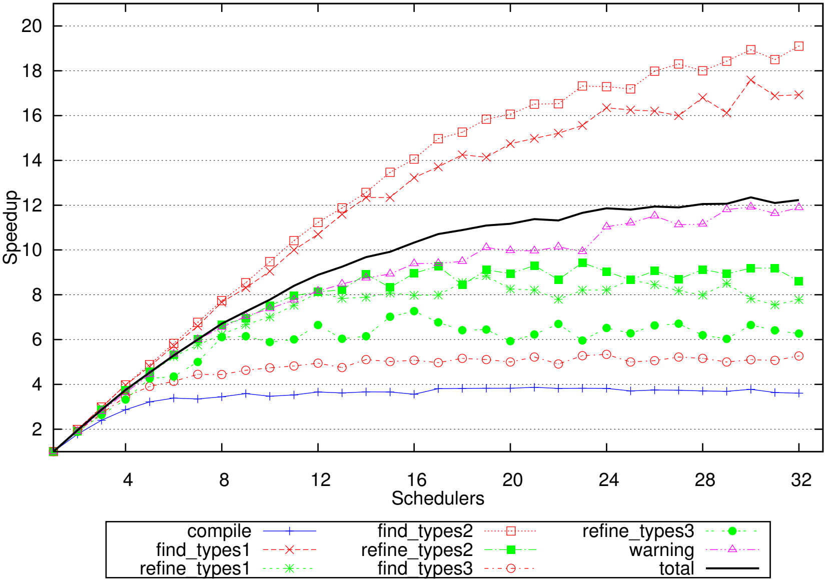
- 10% of the existing code affected
- ... mostly for the conversion of dictionary data structures to ETS tables

Was it worth it?

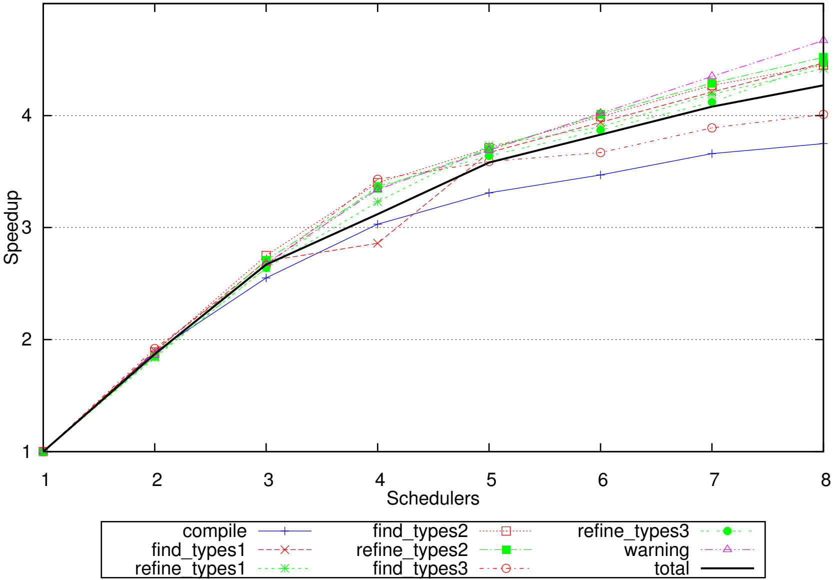
Analyzing Erlang/OTP (AMD Bulldozer)

Phase	1 core	32 core	Speedup
compile	114.67s	23.41s	4.9x
prepare	4.83s	5.59s	-
order	11.16s	11.47s	-
types1	1408.07s	78.61s	17.9x
order	9.93s	8.86s	-
refine1	240.22s	22.39s	10.7x
order	15.14s	15.23s	-
types2	2443.59s	110.74s	22.0x
order	6.35s	5.81s	-
refine2	247.81s	21.09s	11.7x
order	0.28s	0.27s	-
types3	95.45s	15.38s	6.2x
order	0.12s	0.11s	-
refine3	28.99s	3.15s	9.2x
round 4 & 5	<0.50s	<0.50s	-
warning	308.26s	23.58s	13.0x
Total	82m29.87s	6m0.80s	13.7x

Analyzing Erlang/OTP (AMD Bulldozer)



Analyzing Erlang/OTP (i7)



... the tides of time

- Published in Trends in Functional Programming 2012 symposium (June 2012)
- Refreshed results (June 2013, `R16B`) on AMD Bulldozer:
  - **1 scheduler:** 24m25s (was 82 minutes)

    Special thanks to Hans Bolinder (OTP) for *typesig* optimizations!
  - **32 schedulers:** ???
  - **16 schedulers:** ???

Conclusion

Parallel Dialyzer is already part of Erlang/OTP (`R15B03`)

Also, one of RELEASE's benchmarks

`http://www.release-project.eu`

Thank you!