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!
[...] 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
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
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:

- `format/2`
- `valid/1`
- `format_arg/2`
Closer to reality – SCCs

(Highlighted functions are exported)
Performance of sequential version

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

<table>
<thead>
<tr>
<th>Task</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile</td>
<td>114.67s</td>
</tr>
<tr>
<td>prepare</td>
<td>4.83s</td>
</tr>
<tr>
<td>order</td>
<td>11.16s</td>
</tr>
<tr>
<td>typesig 1</td>
<td>1408.07s</td>
</tr>
<tr>
<td>order</td>
<td>9.93s</td>
</tr>
<tr>
<td>refine 1</td>
<td>240.22s</td>
</tr>
<tr>
<td>order</td>
<td>15.14s</td>
</tr>
<tr>
<td>typesig 2</td>
<td>2443.59s</td>
</tr>
<tr>
<td>order</td>
<td>6.35s</td>
</tr>
<tr>
<td>refine 2</td>
<td>247.81s</td>
</tr>
<tr>
<td>order</td>
<td>0.28s</td>
</tr>
<tr>
<td>typesig 3</td>
<td>95.45s</td>
</tr>
<tr>
<td>order</td>
<td>0.12s</td>
</tr>
<tr>
<td>refine 3</td>
<td>28.99s</td>
</tr>
<tr>
<td>[round 4 &amp; 5]</td>
<td>&lt; 0.50s</td>
</tr>
<tr>
<td>warning</td>
<td>308.26s</td>
</tr>
<tr>
<td></td>
<td>done in 82m29.87s</td>
</tr>
</tbody>
</table>
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)
(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_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?

```erlang
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 $\rightarrow$ Too many half-finished jobs
- Allow only as many active workers as logical cores
- Erlang schedulers are efficient ($\approx 100\%$ 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)

<table>
<thead>
<tr>
<th>Phase</th>
<th>1 core</th>
<th>32 core</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile</td>
<td>114.67s</td>
<td>23.41s</td>
<td>4.9x</td>
</tr>
<tr>
<td>prepare</td>
<td>4.83s</td>
<td>5.59s</td>
<td>-</td>
</tr>
<tr>
<td>order</td>
<td>11.16s</td>
<td>11.47s</td>
<td>-</td>
</tr>
<tr>
<td>types1</td>
<td>1408.07s</td>
<td>78.61s</td>
<td>17.9x</td>
</tr>
<tr>
<td>order</td>
<td>9.93s</td>
<td>8.86s</td>
<td>-</td>
</tr>
<tr>
<td>refine1</td>
<td>240.22s</td>
<td>22.39s</td>
<td>10.7x</td>
</tr>
<tr>
<td>order</td>
<td>15.14s</td>
<td>15.23s</td>
<td>-</td>
</tr>
<tr>
<td>types2</td>
<td>2443.59s</td>
<td>110.74s</td>
<td>22.0x</td>
</tr>
<tr>
<td>order</td>
<td>6.35s</td>
<td>5.81s</td>
<td>-</td>
</tr>
<tr>
<td>refine2</td>
<td>247.81s</td>
<td>21.09s</td>
<td>11.7x</td>
</tr>
<tr>
<td>order</td>
<td>0.28s</td>
<td>0.27s</td>
<td>-</td>
</tr>
<tr>
<td>types3</td>
<td>95.45s</td>
<td>15.38s</td>
<td>6.2x</td>
</tr>
<tr>
<td>order</td>
<td>0.12s</td>
<td>0.11s</td>
<td>-</td>
</tr>
<tr>
<td>refine3</td>
<td>28.99s</td>
<td>3.15s</td>
<td>9.2x</td>
</tr>
<tr>
<td>round 4 &amp; 5</td>
<td>&lt;0.50s</td>
<td>&lt;0.50s</td>
<td>-</td>
</tr>
<tr>
<td>warning</td>
<td>308.26s</td>
<td>23.58s</td>
<td>13.0x</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>82m29.87s</td>
<td>6m0.80s</td>
<td>13.7x</td>
</tr>
</tbody>
</table>
Analyzing Erlang/OTP (AMD Bulldozer)

![Graph showing speedup with varying schedulers.](image)

- Compile
- Find types 1
- Refine types 1
- Find types 2
- Refine types 2
- Find types 3
- Refine types 3
- Warning
- Total

Stavros Aronis
Parallel Erlang - Speed beyond Concurrency
Analyzing Erlang/OTP (i7)

Speedup
Schedulers
compile
find_types1
refine_types1
find_types2
refine_types2
find_types3
refine_types3
warning
total

Stavros Aronis
Parallel Erlang - Speed beyond Concurrency
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!