Optimizing for fast builds with GHC

Teo Camarasu

CircuitHub

https://informal.codes/talks/hew26

Optimization Workflow

1. Benchmark
2. Develop a theory about the bottleneck
3. Experiment
4. Reflect
5. Repeat

Factors

- Your source code
- GHC
- Build tools (cabal, stack, nix, etc)
- Compiler options
- Hardware of build machine(s)
- Randomness

Assumptions

Developing locally using HLS or ghcid/ghciwatch

- Build tool is cabal-install/stack
- Laptop with 16 cores and enough RAM
- At most 8 cores per package.
- GHC tuned for fast feedback
  - `-O0`
  - maybe `-fno-code`

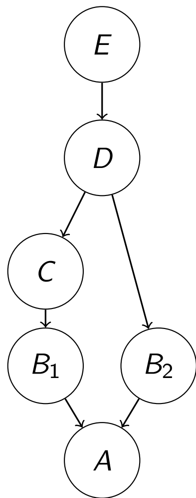
CI builds using nix

- A nix derivation per package
- Multiple CI builders with many cores
- At most 8 cores per package.
- GHC builds with optimizations

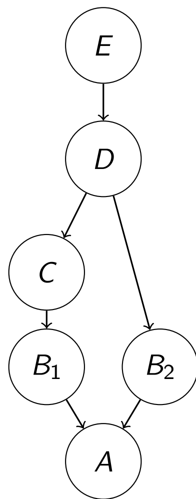
Project layers

- *Project* is a dag of packages
- *Package* is a dag of modules (cycles broken with .hs-boot files)
- *Module*

Analysing package compile times



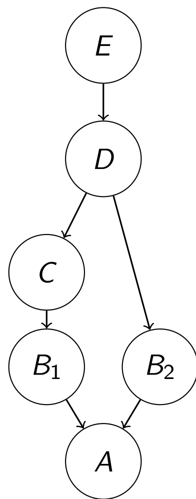
Analysing package compile times



Single core

Sum of time to compile each module.

Analysing package compile times



Single core

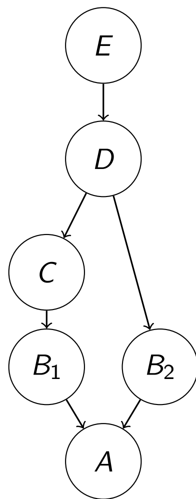
Sum of time to compile each module.

Perfect parallelism

Sum of time taken to compile each module divided by cores.

- Shape of module graph.
- Cannot divide single module between cores.

Analysing package compile times



Single core

Sum of time to compile each module.

Perfect parallelism

Sum of time taken to compile each module divided by cores.

- Shape of module graph.
- Cannot divide single module between cores.

Critical path

Path where the sum of the nodes is greatest. The time it would take if we had infinite cores.

Caching

GHC's recompilation avoidance

- Modules only recompile when necessary.
- Worst case: modules recompile if a transitive dependency changes.
- Critical path is the worst case cost of changing a single module.

Nix's caching

- If any transitive dependency changes then the derivation must rebuild.

Module

Identify slowest modules

- ghc-build-stats
- time-ghc-modules
- Other tools that analyse `-ddump-timings` output

ghc-build-stats

Plugin for collecting build statistics

```
build-depends: ghc-build-stats-plugin
ghc-options: -fplugin=GHC.BuildStats.Plugin
```

Executable for analysing the output

Add to cabal.project like so:

```
extra-packages: ghc-build-stats
```

Then run by:

```
find -name 'build-stats.ndjson'
cabal run ghc-build-stats -- <path>/build-stats.ndjson
```

Slowest modules to compile

Do you need it?

Do you actually need this module still?

Check with: weeder

Analyse Core

Generate Core by setting these `ghc-options`:

```
{-# OPTIONS_GHC -ddump-to-file
                -ddump-simpl
                -dsuppress-all
                -dno-suppress-core-sizes
#-}
```

Then find it using:

```
find -name '*.dump-simpl'
```

We can see the size of each term:

```
RHS size: {terms: 3, types: 0, coercions: 0, joins: 0/0}
```

Binary search for slow parts

1. Comment out half the file or stub it out with `undefined`
2. Check if the file is still slow to compile.
3. Repeat until you find a minimal cause.

Reasons for slowness

1. Excessive type level programming
   - Generics
   - Higher kinded data
2. Code size explosion
   - Excessive inlining/specialisation
   - Derived instances
   - Large data types (large-records)
   - TemplateHaskell sometimes
3. GHC bugs

Package

Analysis

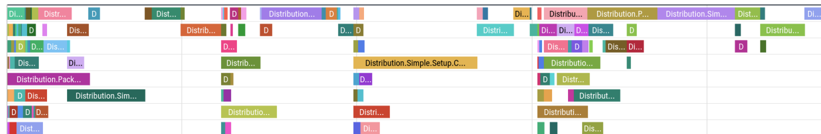
- graphmod
- ghc-build-stats

  Shows the critical path.

ghc-build-stats on Cabal

Chrome trace file displayed using perfetto.dev.

Critical path is 23, and total time is 55s.



Re-exports

Often a module will import A just to get something re-exported from B.

We can simply import B instead.

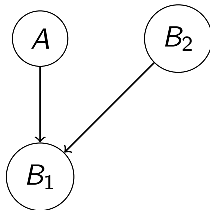
- Use HLS refine imports action.
- Issue to add warning: GHC#26836

Splitting modules

- Avoid creating linear chains.

  Splitting $A \rightarrow B$ into $A \rightarrow B_1 \rightarrow B_2$
- A module should have a clear purpose.
- Analyse modules using: calligraphy
- Avoid `Types` modules, be more descriptive.

Inverting dependencies



- Record of functions
- Typeclass
- Effect like effectful/bluefin

Project

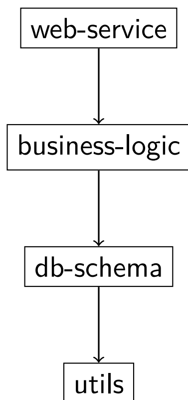
Analysis

- cabal-plan
- stack dot

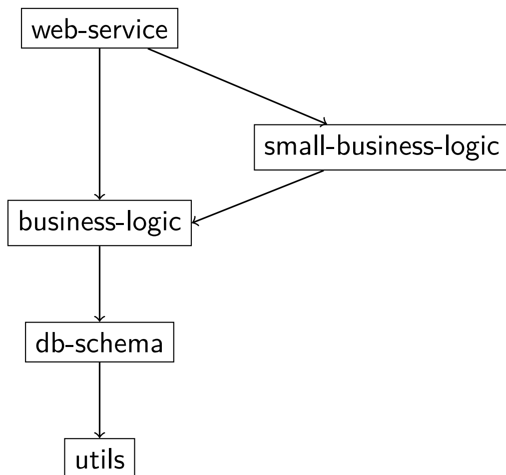
Why?

- critical path
- improve intra-package parallelism

Common shape

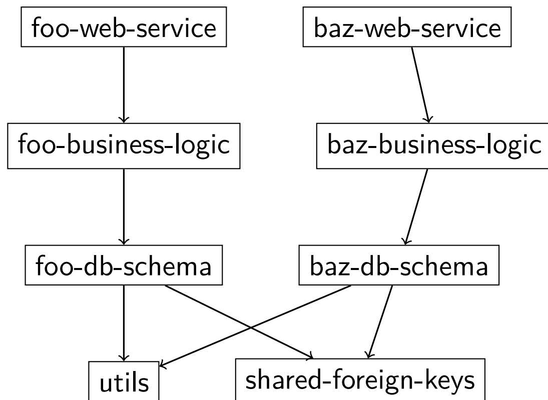


Don't split in the middle of the chain!



Splitting advice

- Things that get modified together should live near each other.
- Split by theme or team.
- Avoid tiny packages.
- Limit the amount of layers.



Eliminate external dependencies

Try deleting unused external dependencies

Conclusion

- Put care into your build times
- Minimise your critical path
- Make use of parallelism and caching