Benchmarking the OCaml compiler: our experience

Tom Kelly

OCaml 2019, Berlin, 23 August 2019

Overview

- Our aims
- What OCaml benchmarking suites exist
- How are other compiler communities handling the benchmarking for continuous integration
- What we built and bits to look out for
- What we think can be learnt
- Interesting future directions

What we set out to do

- Make it easier to upstream multicore OCaml changes and know that each step along the way we haven't made OCaml code run significantly slower.
- Try to reuse benchmarking suite work that has been done to cover a range of micro and macro use cases.
- Try to incorporate best practice and reuse tools where appropriate by looking at what other communities are doing.
- Provide visualisations into the data to make it less time consuming for developers to spot and investigate performance regressions.
- Document the steps to setup the system including its experimental environment for benchmarking so that the knowledge is captured for others to reuse.

OCaml benchmarking suites

- operf-mirco: incorporates a Core_bench/Criterion style of running micro benchmarks and denoising the results.
- operf-macro: leverages opam to pull in full OCaml packages with dependencies to form application level use cases.
- sandmark: put together in response to our experience with multicore benchmarking which builds on operf-macro and uses a static opam setup to form application level use cases alongside benchmarks specifically aimed at multicore.

How other communities are handling benchmarking

- **Python (CPython and PyPy)**: common benchmarking suite is run across builds and displayed using an open-source web app called Codespeed.

- **LLVM**: micro-benchmarks built on google-benchmark (and ability to pull in external suites such as SPEC CPU 2006). Have a continuous integration site and LNT software backing it.

- **GHC**: hard regression test suites for performance gives problems and they are wanting to migrate to a data driven approach.

- **Rust:** have tools for benchmarks (covering compile and runtime performance) and a web app to display them. Commit the data collected into a GitHub repo.

What we built and the bits to look out for

- Operf-micro and sandmark benchmark suites using Codespeed to display our results.

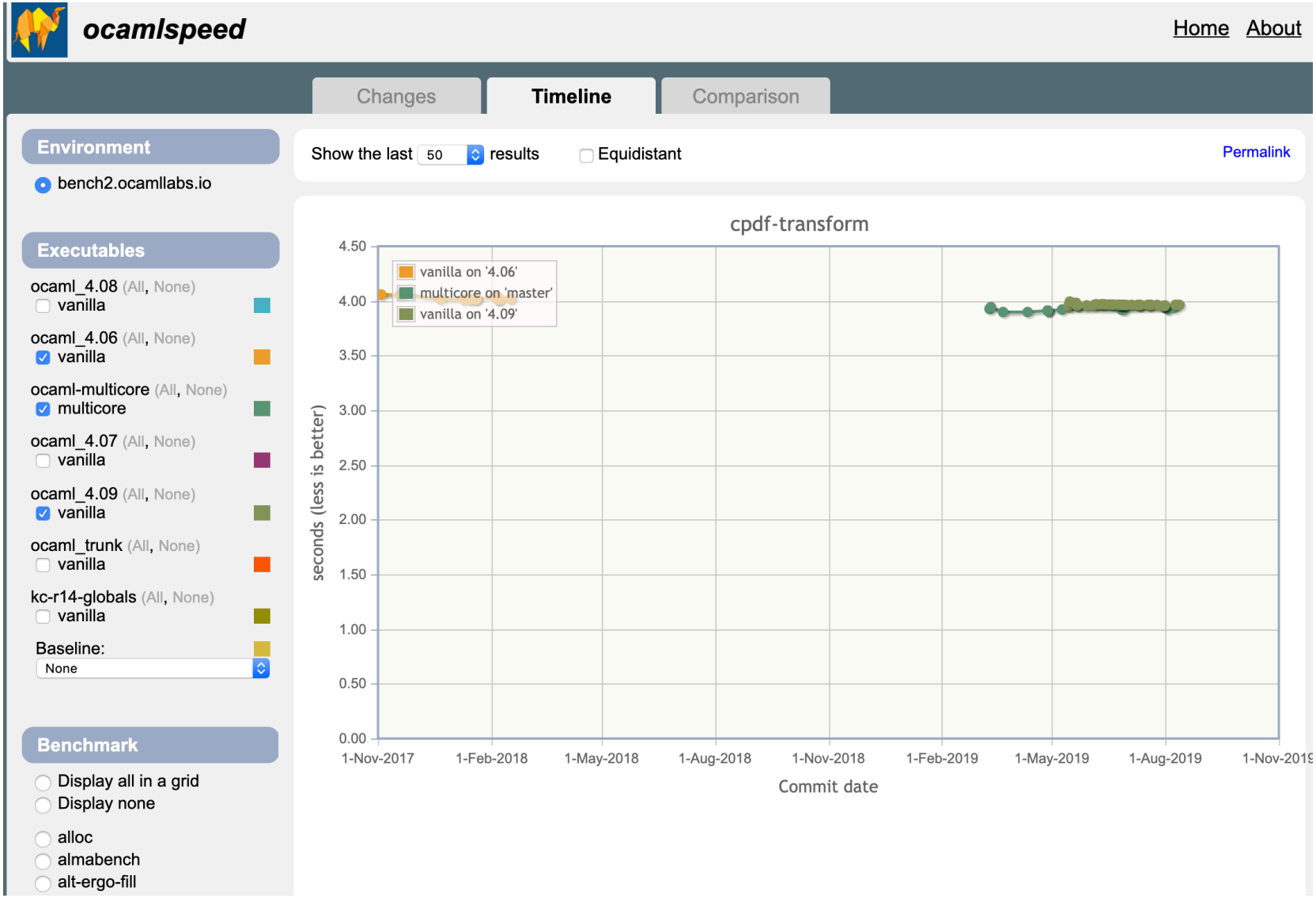
- Run the benchmark suite on each commit along a git branch which we map using first-parent and commit timestamps.

- We wanted to make the experimental environment in which we collect the data from benchmark runs as clean as we could to make things repeatable and follow up inspection easier.

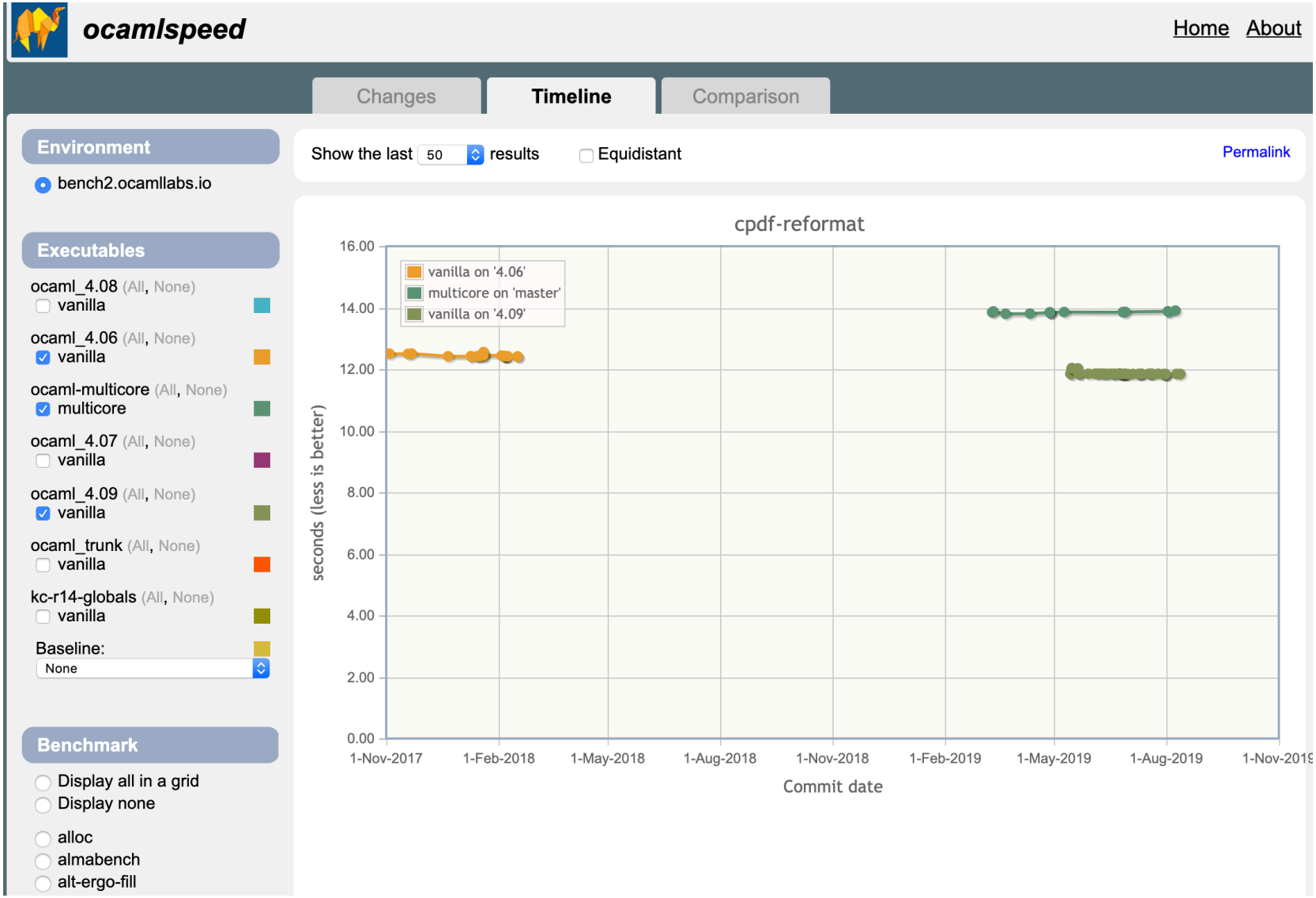
Experimental setup

- **Hyperthreading**: we switched this off in the BIOS
- **Turbo boost:** disabled to avoid clock wandering depending on machine temperature
- **pstate:** set this to performance rather than powersave
- **Linux CPU isolation:** remove our experimental CPUs from the kernel scheduler
- **Interrupts:** move interrupts away from the experimental CPUs
- **Address Space Layout Randomisation (ASLR):** turned off to make repeatable *(open issue what to do here)*

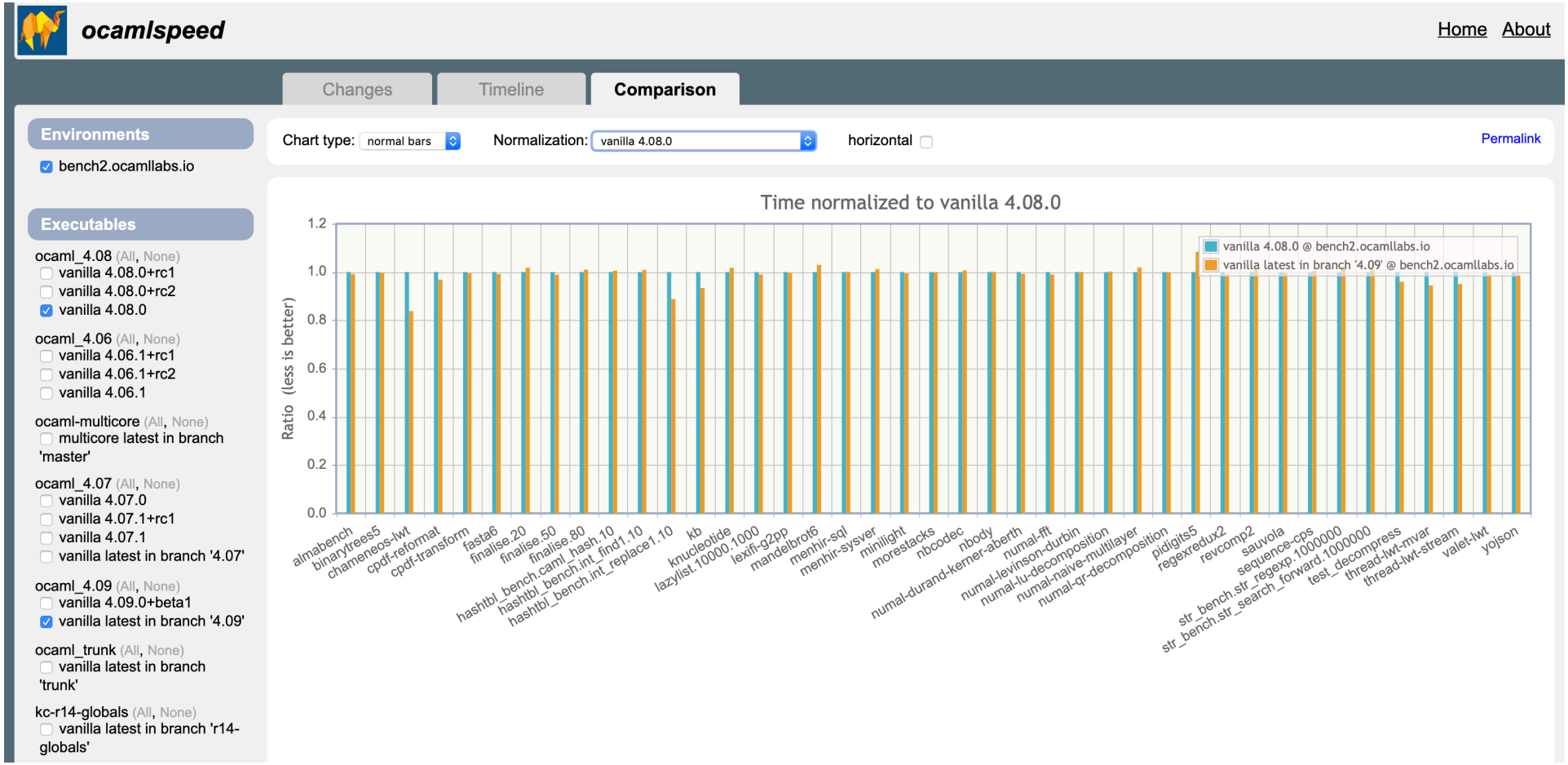
Timelines by commit



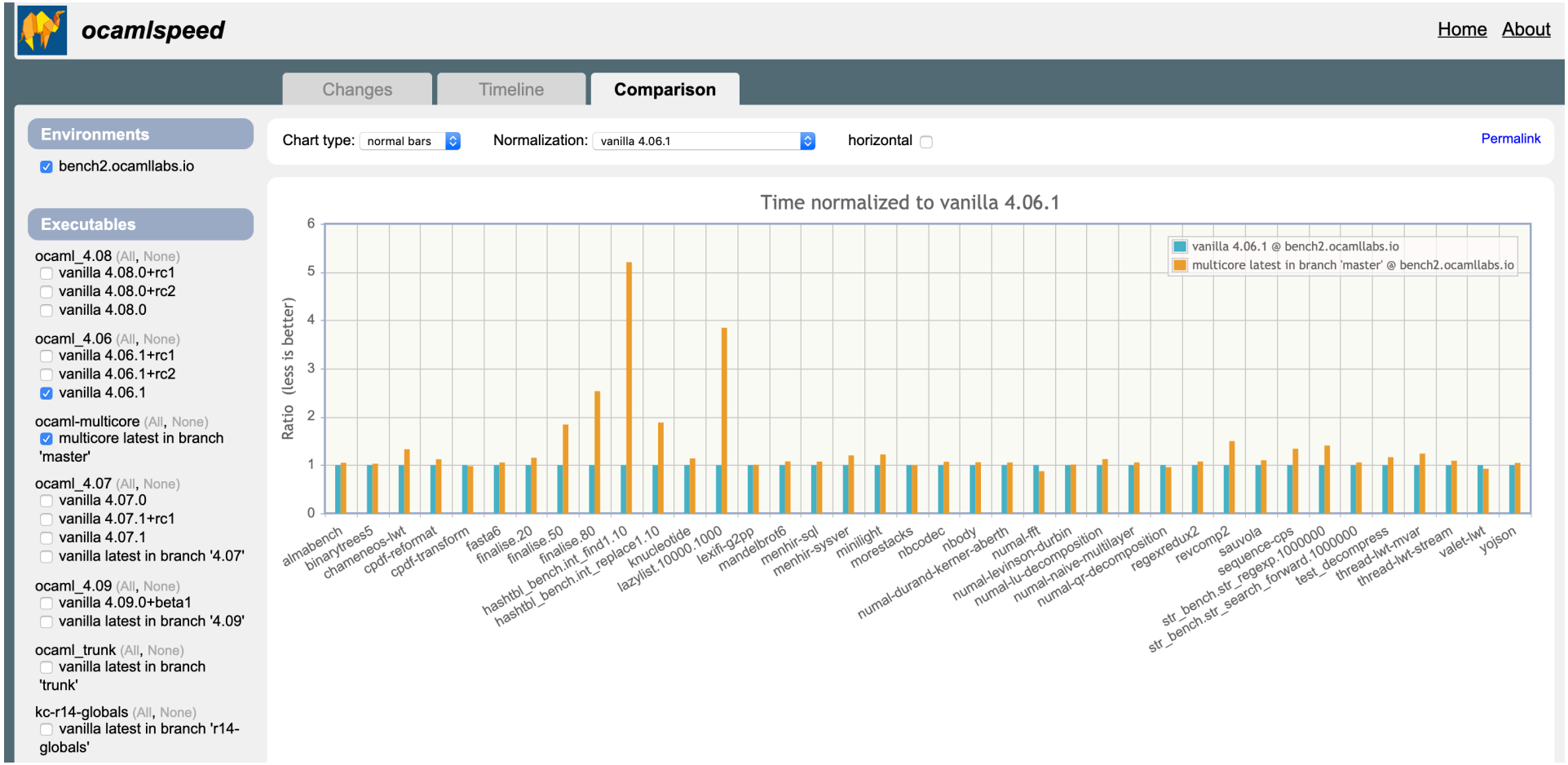
Timelines by commit



Comparisons



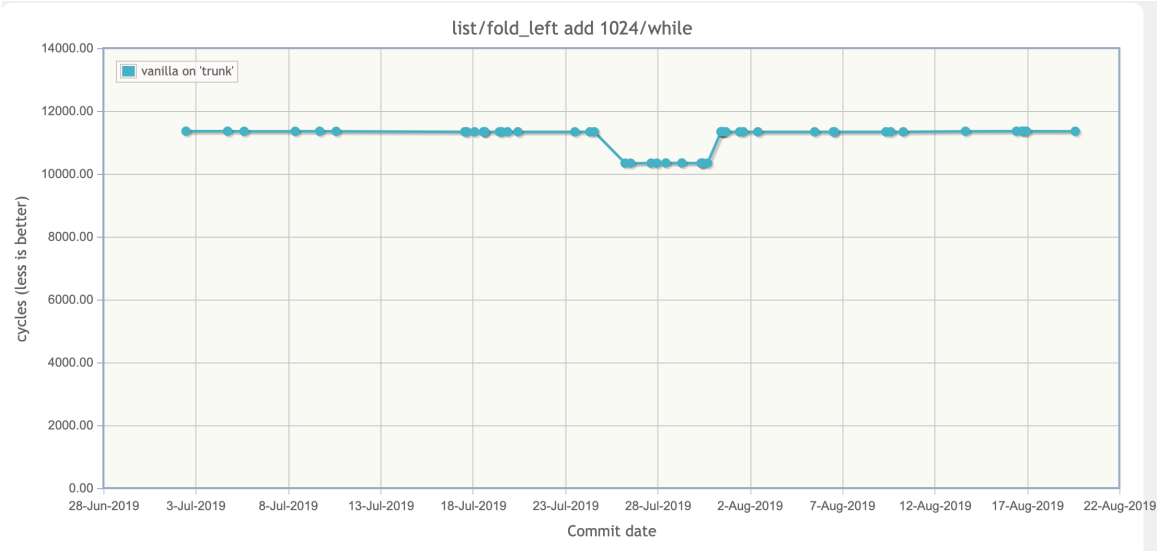
Comparisons



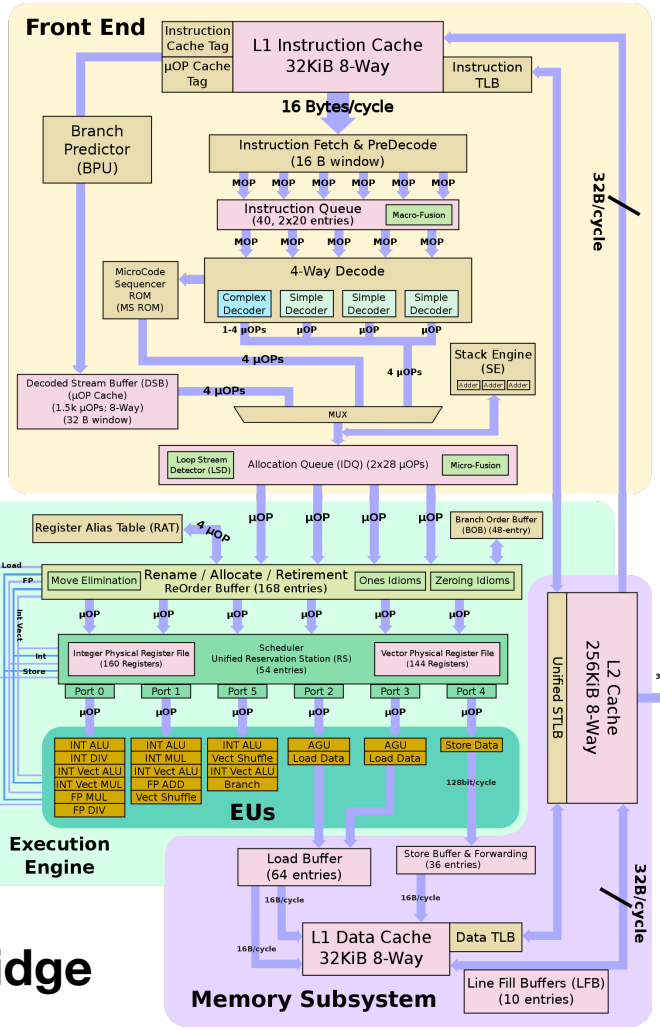
Not always obvious what is causing performance differences

- Ability to run repeatably and drill into results has allowed us to observe micro-architecture effects up close.

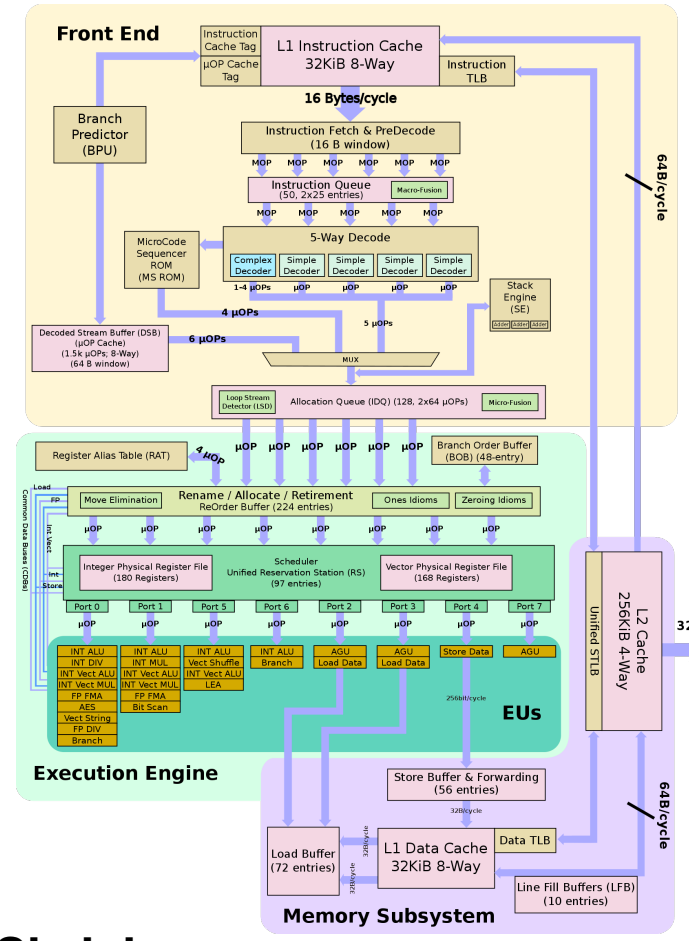
- One of them appeared bizarre (to me at least). The code changes were in another area of the runtime with no change in the executed instructions but the performance was verifiably moving by ~10%.



Modern processors are complex systems



Ivybridge



Skylake

(Wikichips diagrams)

Micro-architectural effects can be significant

- I dived into this case and it turned out to be due to the 32B DSB blocks. There's no easy answer here for what the compiler can do (in the absence of profile guidance).

- Silicon resource is limited, when you hit the edge of that it becomes the bottleneck (see Zia Ansari's LLVM dev-talk for a deeper dive).

- How should (backend) compiler writers respond to this?

  - Be aware and alert, sometimes the performance benchmarks can be sensitive to micro architectural effects (Linux perf is your friend here).

  - The Intel Optimisation manual gives you an understanding of the bottlenecks and handy guidelines to follow. We should try to avoid the known problematic bottlenecks when we can.

  - LLVM has complex block alignment heuristics and more. Unclear how we take some of the best practice into the OCaml compiler.

Macro-benchmark maintenance load and relevance

- Maintaining a reasonable sized collection of macro-benchmarks against trunk OCaml has proved to be a fair bit of work (unless we take a relaxed approach to breakage until things are fixed).

- Another worry is the relevance of the macro benchmarks. Maybe we need a way to standardise a ‘benchmark’ target for a package. This would allow the users to maintain test cases they care about and can update the test cases as the software evolves.

Future directions

- Try to add some profiling output; add some ‘why’ to the measurements?
- Attempt to containerise and make this a ‘service’?
- Scale across architectures (ARM, multiple x86 variants)?

Conclusion

- Benchmarking is important to ensure that you can avoid performance regressions when changing the OCaml runtime and compiler.
- We've been successful at surfacing performance issues with OCaml multicore before it is upstreamed.
- Benchmarking is only part of the story, knowing you (might) have a problem isn't the same as knowing why you have a problem (or don't have a problem at all). Profiling becomes an important part of the picture.