OUTLINE: Benchmarking the OCaml compiler: our experience

Our proposed presentation would describe what we did to build continuous benchmarking websites[1] that take a controlled experiment approach to running the operf-micro and sandmark benchmarking suites against tracked git branches of the OCaml compiler.

Our goal is to provide statistically meaningful benchmarking of the OCaml compiler to aid our multicore upstreaming efforts through 2019. In particular we wanted an infrastructure that operated continuously on all compiler commits and could handle multiple compiler variants (e.g. flambda). This allows developers to be sure that no performance regressions slip through and also guides efforts as to where to focus when introducing new features like multicore which can have a non-trivial impact on performance, as well as introduce additional non-determinism into measurements due to parallelism.

Our high level aims were:

- Try to reuse OCaml benchmarks that exist and cover a range of micro and macro use cases.
- Try to incorporate best-practice and reuse tools where appropriate by looking at what other compiler development communities are doing.
- Provide visualizations that are easy to use for OCaml developers making it easy and less time consuming to merge complex features like multicore without 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.

The presentation will be structured in sections as follows.

Survey of OCaml benchmarks

We describe briefly the available open-source OCaml benchmarking suites: operf-micro and operf-macro. We introduce the sandmark[2] benchmarking suite we have developed which builds on operf-macro.

How other compiler communities handle continuous benchmarking

We give a survey of how other compiler communities (Python, LLVM, GHC, Rust) are handling continuous benchmarking to give an idea for the existing state of the art.

What we put together

We describe how we put our continuous benchmarking system together to track OCaml compiler branches and visualize the results. This will cover:

- The experimental setup and the important parameters to control.
- The visualizations we present to allow users to interact with the collected data.

Interesting experience to share

We will describe our experience of using the tools and what we learnt that may be useful to other people conducting performance benchmarking. In particular:

- The ability to look at a clean controlled series of reproducible experiments allows an easy way to assert there are no performance regressions and quickly identify what might have caused any regressions that have arisen.
- It takes care to get a clean experimental setup with high repeatability, but it can be done. It also then reveals some interesting properties in modern systems which we observed: the impact of address space layout randomization on performance; surprising ways that code layout can impact performance.
- You can quickly need a large system to cover all the variants that you want to collect data on. Strategies we used for managing this.

Future work and extensions

We can see multiple areas that things could be improved in the future:

- Rich front-end visualization for multi-dimensional performance data; e.g. full perf counters, including multiple metrics of garbage collection performance.
- How to create sandboxed repeatable performance inside containers on publicly available cloud compute infrastructure.
- How to incorporate the x86 front-end discoveries into code alignment emitted from the OCaml compiler.

For those interested in more details on the content of the presentation, there are more in depth details here: https://github.com/ocaml-bench/notes/blob/master/apr19.md

References

[^1]: http://bench.ocamllabs.io which presents operf-micro benchmark experiments and http://bench2.ocamllabs.io which presents sandmark based benchmark experiments.

[^2]: Code for sandmark https://github.com/ocamllabs/sandmark