Towards a Benchmarking Service for OCaml

Luis Eduardo de Souza Amorim
eduardo@tarides.com
Tarides
Australia

Tim McGilchrist
tim@tarides.com
Tarides
Australia

Abstract

In this talk, we discuss our efforts to implement a new benchmarking system for OCaml, with the aim of building a successor to the Sandmark suite as a comprehensive community-facing OCaml performance tracking system. We take inspiration from the well-established DaCapo benchmark framework for Java, collecting macro-benchmarks from a diverse set of OCaml real-world applications. In the talk, we explain why this benchmarking system is a crucial part of keeping track of performance regressions, how the benchmarking setup is structured, and some preliminary outcomes from the initial prototype. We end the talk by presenting the next steps for this project.

1 Introduction

OCaml is a language in constant evolution. Features such as effect handlers [11] and support for parallelism with domains [10] are some of the most recent changes impacting OCaml's runtime.

A key requirement for tracking performance changes in runtime systems is benchmarking. Languages like Java have, for instance, the DaCapo [2] and Renaissance [7] benchmark suites, along with JMH [6] for micro-benchmarking, while Rust tracks compiler performance through the rustc-perf performance dashboard [12].

The Sandmark suite of OCaml benchmarks [3] helped support the work for retrofitting parallelism into OCaml's garbage collector. While such a set of benchmarks was useful as a stress test of the new runtime, it does not represent what users experience in their OCaml applications. This intuition played out after the initial multicore release of OCaml, when industrial and open-source users like Jane Street [13], Frama-C [1], rocq [8] and others reported performance regressions.

For that reason, we argue that OCaml needs a modern performance observability system with representative benchmarks, repeatable runs that collect runtime metrics, and clear regression feedback. The goal is to track performance impacts observed by users, catch regressions earlier, and track which changes are responsible for such regressions.

We aim to create a publicly accessible, continuously updated performance dashboard for the OCaml ecosystem, supporting:

- Tracking changes between major releases.
- Tracking the impact of pull requests before merging.
- Studying runtime behaviour under different workloads.
- Building confidence in OCaml's performance trajectory.

In this paper we will describe the steps towards realising this vision and the current status of the project. Finally, we discuss the future direction of this work.

2 Background

Sandmark [3] was developed as a benchmarking suite to track performance in multi-core OCaml, in particular, with respect to its garbage collector. Sandmark consists of around a hundred micro-benchmarks and a few macro-benchmarks, but the project never became a standardised way of tracking runtime performance, especially as the runtime kept evolving past version 5. One likely reason is that, even though it proved useful for stress testing and retrofitting parallelism into the garbage collector, it did not provide a way to track the actual user experience.

To address that, our work is inspired by the DaCapo benchmarking suite for Java, and consists of a collection of real-world OCaml applications, so that most benchmarks run for 5–25 seconds and can be modified for longer runs. This not only exposes the user experience in terms of performance regressions, but it allows us to expose regressions that would otherwise not be captured by smaller micro-benchmarks.

We propose benchmarking OCaml as a service. We initially envision three use cases for this service: *intra-runtime benchmarking* for performing GC parameter sweeps, *inter-runtime benchmarking* for tracking performance regressions or improvements between different runtimes and their variants, and finally to *capture regressions* between major versions or releases. Note that while these may look like three distinct use cases, they can be interconnected to detect regressions as we will show in Section 3.1.

The infrastructure should be extensible in terms of supporting different runtimes and different types of workloads, covering a diverse set of OCaml applications. In the end, we want to support local runs and extend this work to implement a request-based service that can run on pull requests, for example.

In this project, our goal is to reuse existing OCaml tools. For instance, we use `opam monorepo` [4] to vendor benchmarks and make sure that their source is fixed when using different compiler versions. We use `opam compiler` [5] (an opam plugin) to build compilers from pull requests or git commits. Finally, we collect runtime metrics using `olly` from `runtime_events_tools` [9] and using `perf` to capture hardware performance counters. In this way, the whole infrastructure can benefit from improvements and bug fixes, and we dog-food our own tools for performance observability.

3 Current Status

We started prototyping the benchmarking framework with a mechanism to run the benchmarks locally. Ideally, this setup can be extended to implement our framework as a service to run on demand, on pull requests or when doing a release. The framework consists of source programs, an orchestrator responsible for building the runtimes, running the benchmarks and producing structured output. This output can then be used as input to a visualisation layer, which enables reasoning about the benchmarking results.

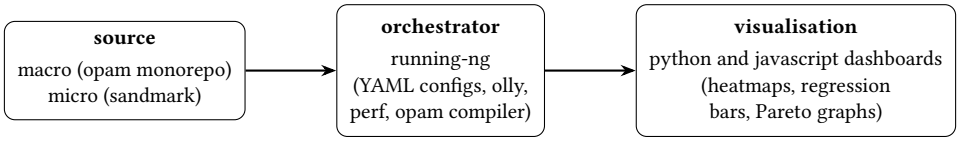


Figure 1: Overview of the benchmarking pipeline. Micro-benchmarks from sandmark or vendored macro-benchmarks are passed to the `running-ng` orchestrator, which builds the runtimes and respective executables, runs the benchmarks under `perf` and `olly`, and emits structured JSON consumed by the visualisation tools.

Concretely, we have extracted the micro-benchmarks from the Sandmark suite as standalone benchmarks[1], which can be used with our framework or independently with a small amount of opam configuration. These micro-benchmarks could easily be extended and maintained by the community. However, as pointed out earlier, our goal is to use macro-benchmarks that consist of real-world OCaml applications. We intend to focus on mostly macro-benchmarks, but as shown in Figure 1, one may run both micro- and macro-benchmarks locally. The current list of macro-benchmarks[2] is presented in Table 1.

To run the benchmarks we prototyped a set of running scripts[3] that are based on those from DaCapo. They orchestrate benchmarking runs by building the compiler, building the binaries, isolating each run and wrapping them with `perf` and `olly` to collect metrics. A run is specified with a YAML file, defining which runtimes to build, which benchmarks to run, and which runtime parameters to sweep. At the end of the run, we use the structured JSON output from `perf` and `olly`, combined with some metadata about the run, which can be used as input to a visualisation component. As part of our prototype we implemented such a component based on Python notebooks and the Observable Framework[4].

Take, for example, the use case of checking for the performance overhead of enabling frame pointers in the compiler. One may write a YAML file as in Figure 2.

The `running-ng` scripts take a YAML file like the one from Figure 2 as input and build each of the compiler variants using `opam compiler`. Then, the scripts build each of the macro-benchmarks which are vendored using `opam monorepo` inside the macro-benches repository. Finally, they run the benchmarks attaching `olly` and `perf` to the target process and output the results as structured JSON.

In our prototype, we have defined three different notebooks to answer different questions about the resulting data.

- *Notebook A*: The first notebook consists of a regression dashboard. Based on the comparisons defined in the YAML file, we intend to ask the question: *did this compiler change regress or improve the benchmarks?* We look at instruction counts, wall time and max RSS in order to define regressions, and capture them with tornado plots ranking top-n regressions and improvements.
- *Notebook B*: The second notebook contains the absolute values, not ratios. It helps a developer or researcher asking: *what is the runtime actually doing?* To answer that it contains various bar plots with the absolute values for wall time, max RSS, GC overhead and latency. This is very useful when analysing regressions, e.g., before doing a major release.
- *Notebook C*: The third notebook is designed specifically for parameter sweeps. It tries to help answer three questions: (i) *are there parameters better than the default for this particular*

[1] https://github.com/ocaml-bench/benches
[2] https://github.com/ocaml-bench/macro-benches
[3] https://github.com/udesou/running-ng
[4] https://observablehq.github.io/framework/

Table 1: Macro-benchmarks currently supported by the suite: 21 tools spanning 30 programs (some benchmarks have more than one program) across 13 application categories.

Benchmark	Category	Runtime features
`menhir`	Text processing	Polymorphic hash, lazy
`cpdf`	Text/media	C stubs, polymorphic hash
`alt-ergo`	SMT solver	Ephemerons[a], custom blocks, C stubs, signals
`coq/rocq`	Proof assistant	Ephemerons, Marshal, lazy
`ahrefs-devkit`	GC stress	Custom blocks, finalisers, C stubs, polymorphic hash
`irmin`	Database	Polymorphic hash, C stubs
`ocamlformat`	Text Formatting	Polymorphic hash
`decompress`	Compression	Custom blocks, finalisers, off-heap allocation
`eio`	Concurrency	Effects, atomics
`sedlex`	Text processing	-
`yojson`	Text processing	-
`zarith`	ML/Numerics	Custom blocks[b], C stubs
`owl`	ML/Numerics	Custom blocks, finalisers, off-heap allocation, C stubs
`pplacer`	Bioinformatics	Custom blocks, finalisers, off-heap allocation, C stubs
`ocamlc-self-compile`	Compiler	Marshal, custom blocks, polymorphic hash
`ocamlc-compile-uucp`	Compiler	Marshal, polymorphic hash
`liquidsoap-lang`	Compiler	Lazy, atomics
`liq-video-frames`	Text/media	Custom blocks, finalisers, off-heap allocation
`frama-c`	Static analysis	Ephemerons
`goblint`	Static analysis	Ephemerons, custom blocks, C stubs
`js_of_ocaml`	Compiler	Marshal, lazy

[a] alt-ergo is the only benchmark that uses an ephemeron *data* field (`Ephemeron.K1.Make` as a memo cache in the combined-theory solver). The other ephemeron benchmarks use weak arrays with no data, so the data-clearing branch of `caml_ephe_clean` is exercised here alone.
[b] zarith's custom blocks carry no finaliser and declare no out-of-heap memory: the GMP limbs are stored inline in the block, so it exercises the custom-block allocation and compare/hash/serialise operations but neither finalisation nor GC pacing.

```yaml
includes:                  # macro-benchmark suite
  - "../base/ocaml/macro_base.yml"

runtimes:
  ocaml-5.5.0:             # baseline
    type: OCaml
    version: "5.5.0"
  ocaml-5.5.0-fp:          # frame pointers on
    type: OCaml
    version: "5.5.0"
    configure_args:
      - --enable-frame-pointers

configs:                   # configurations to run
  - "ocaml-5.5.0|perf_grp1|re-23|md-2"
  - "ocaml-5.5.0-fp|perf_grp1|re-23|md-2"

comparisons:               # how to compare results
  - label: "frame-pointer overhead"
    a: ocaml-5.5.0
    b: ocaml-5.5.0-fp
```

Figure 2: A minimal configuration measuring the overhead of frame pointers: two runtimes built differing only in `-enable-frame-pointers`, run under the same benchmarks and compared directly.

workload?, (ii) *should we ship new default values for the parameters?*, and (iii) *how do intra-runtime configurations affect inter-runtime comparisons?*

To validate our current setup, we ran a few benchmarking experiments as shown in Section 3.1, which contains a few snapshots from the notebooks above.

3.1 Early Results

Even though our project remains a prototype, we were able to obtain concrete results from our experiments. We list three different experiments and our findings from each of them: (1) a comparison between the last stable release at the time, OCaml 5.4.1, and OCaml 5.5.0-beta1 while doing a parameter sweep, (2) a comparison between 5.5.0 with and without frame pointers and flambda enabled, (3) a regression comparison for the pull request OCaml#14571[5], which modifies the minor GC to improve performance. Instead of discussing each of these in depth, we refer to the results in https://github.com/ocaml-bench/macro-benches/tree/master/results for more information, focusing on the particular outcomes of using our benchmark suite.

Regressions and parameter sweeps. While running experiment (1), we tried to track regressions between major releases, aiming to understand the changes between each version and how they impacted the runtime performance. One thing we noticed is that the default parameters may not always reflect whether there has been a regression. Take, for instance, the heatmap from Figure 3 corresponding to one of the benchmarks in the suite (`liq_parse_typecheck`, a benchmark which parses and typechecks the same ~80-line Liquidsoap script multiple times). Looking at the figure, we see that the inter-runtime analysis on the default parameters for `s` (the minor heap size in words) and `o` (space overhead in %) shows a regression of 5.9% on wall time between 5.5.0-beta1 and 5.4.1. However, when looking at the configuration `(s,o) = (262144,40)`, we found an improvement of about 34% for both wall time and max RSS between the same runtimes. This surfaced the need to do a parameter sweep to correctly identify whether a benchmark truly regressed or not, indicating the interconnectedness between the use cases we listed previously.

Confidence on variant overhead. We used experiment (2) to gain confidence on the actual overhead for compiler variants, such as frame pointers and flambda. With this, we hope to enable developers to back up their claims about the overhead or improvements that their code may add to the runtime. For instance, our benchmark run (under the default parameters on Linux AMD64) showed an overhead of $< 5\%$ on wall time across all the macro-benchmarks, while flambda showed an improvement of up to 40% on wall time for the same workloads as shown in Figures 4a and 4b.

Validating performance claims. When running the macro-benchmarks for this particular change we expected to see some improvement in the instructions executed, since the PR focuses on improving the fast path for minor GC allocations. In general this result is validated by the improvements in Figure 5. However, we found a major regression for the owl benchmark that after further investigation turned out to be a glibc malloc tuning problem.

Finding bugs/regressions earlier and improving related OCaml tools. When running our experiments we were able to find bugs in the runtime we were testing. We discovered the issue OCaml#14842[6] which happened when compiling one of our benchmarks using the latest trunk version. Experiment (3) also showed a bug in the related pull request, which was revealed by one of the benchmarks, and similarly, showed the performance impact of the minor GC changes across all the benchmarks before the PR got merged. We also discovered a bug in `olly`[7] due to terminating programs before their completion and getting invalid runtime metrics.

4 Conclusion and Future Work

In the previous sections we presented our prototype for developing a benchmarking service for OCaml. The primary work has covered the benchmarks themselves, the orchestrator, and a visualisation layer. However, for the service to be complete, there are a few aspects that will need to be covered next.

Infrastructure. We mentioned how Sandmark never became the standard way of tracking performance for OCaml. While we believe having macro-benchmarks is a step in that direction, we still need to enhance the current infrastructure to allow users and developers to access the benchmarking service beyond running it on their local machines. To achieve that, we plan to implement a request-based service that can run on pull requests, or be invoked by developers before a release. We hope to draw inspiration from OCaml's

[5] https://github.com/ocaml/ocaml/pull/14571

[6] https://github.com/ocaml/ocaml/issues/14842

[7] https://github.com/tarides/runtime_events_tools/issues/92

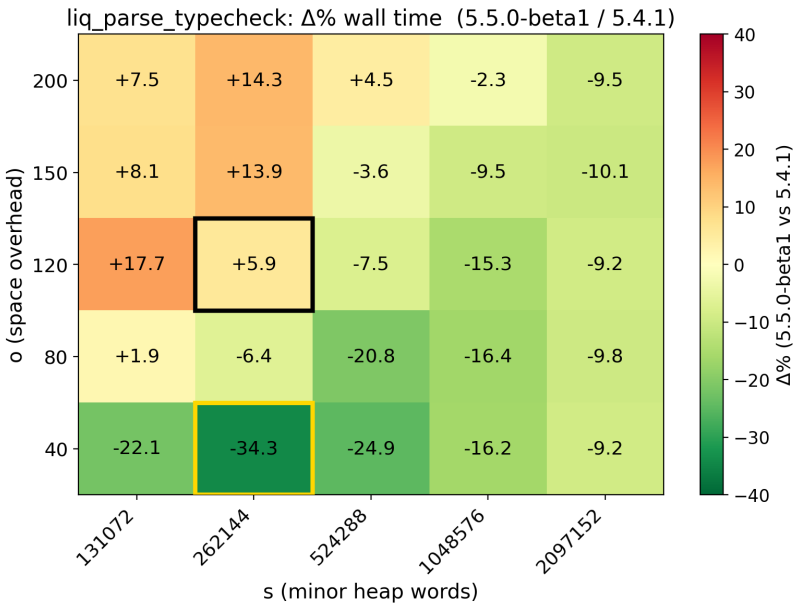


Figure 3: Parameter sweep for `liq_parse_typecheck` benchmark showing a regression on the default parameters and an improvement on `(s,o) = (262144,40)`.

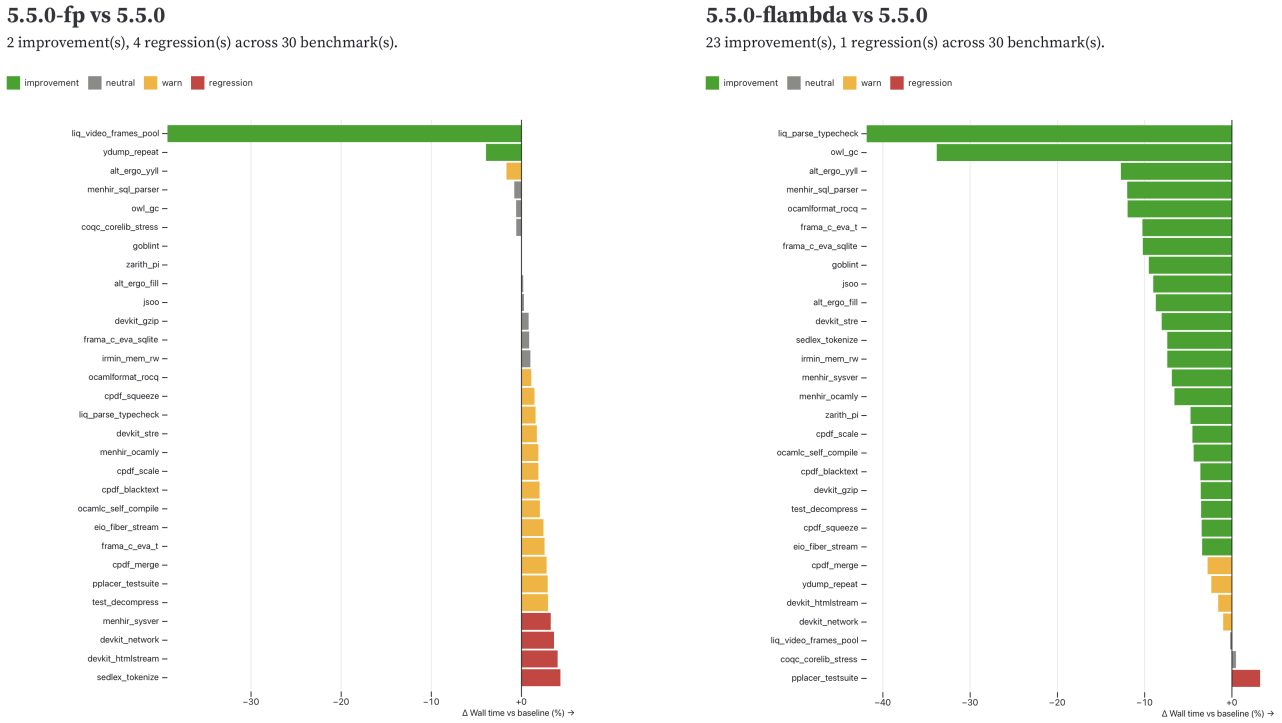
OCurrent[8] services, learning from other systems, while making sure that we address all the requirements of our benchmarking infrastructure. We might also consider replacing `running-ng` with an OCaml version, maintaining the stance of using OCaml-based tools throughout the system.

Benchmark Coverage. Table 1 showed the different benchmarks and categories that we currently cover in our suite. However, this should be an ongoing list as more developers decide to benchmark their applications. Our goal is that such new benchmarks can be maintained by their own developers, which relieves us of some of the maintenance burden. Besides that, there are a few different categories that we would like to cover, such as multi-domain benchmarks, ephemerons, and benchmarks that exercise compaction. Finally, we would like to investigate how we could extend our system to benchmark OxCaml runtime versions.

Runtime Metrics. As mentioned, we lack benchmarking programs that exercise compaction. To properly measure the effect of compaction, we might need to extend our infrastructure to capture fragmentation (live and used memory) throughout the execution of a program. That will likely involve extending `olly` and `runtime_events_tools` to keep track of this information and present it to the user[9]. Finally, we also intend to benchmark the runtime in memory-constrained systems, such as Unikernels. That will also require further extensions to our tooling to capture metrics in a non-obvious way, since `olly` cannot straightforwardly attach to Unikernels.

Platform Support. One of the important goals identified for the macro-benchmarks is to make them runnable locally. Our current efforts focus on support for Linux on AMD64 and ARM64, but it should be possible to extend this support to other architectures. Supporting macOS using Instruments and FreeBSD using `pmcstat` as a replacement for `perf` would provide a path to enabling benchmarking in those platforms. Windows support needs further investigation.

[8] https://github.com/ocurrent/overview
[9] https://github.com/tarides/runtime_events_tools/issues/94



(a) Frame pointers **(b) Flambda**

Figure 4: Per-benchmark overhead of OCaml 5.5.0 compiler variants relative to the unmodified baseline: (a) frame pointers and (b) flambda.

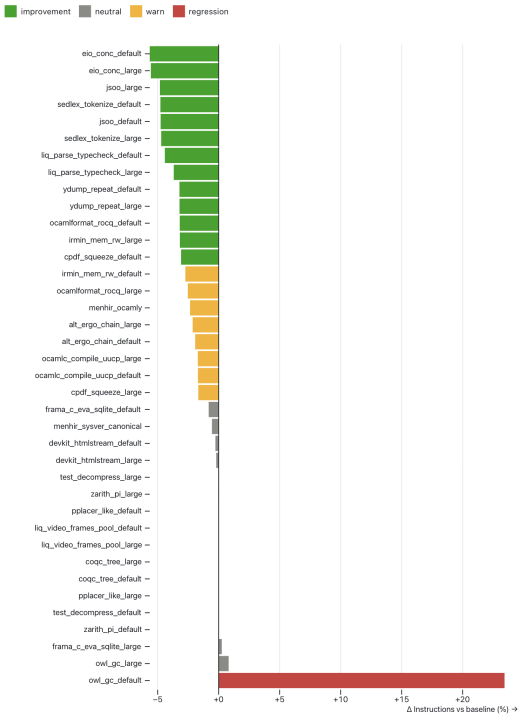


Figure 5: Regression check for OCaml#14571 showing an outlier regression in the owl benchmark.

References

[1] Andre (maroneze). [n. d.]. Memory consumption of Frama-C in OCaml 5 beta1 is about 10x larger. https://github.com/ocaml/ocaml/issues/11662. Accessed: 2026-06-29.

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, 169–190. doi:10.1145/1167473.1167488

[3] OCaml Benchmarking Working Group. [n. d.]. Sandmark: A Benchmark Suite for the OCaml Compiler. https://github.com/ocaml-bench/sandmark. Accessed: 2026-06-29.

[4] OCaml Opam. [n. d.]. opam-monorepo: opam-monorepo is an opam plugin designed to assemble standalone Dune workspaces with your projects and all of their dependencies, letting you build it all from scratch using only dune and ocaml. https://github.com/tarides/opam-monorepo. Accessed: 2026-06-29.

[5] Opam Compiler. [n. d.]. Opam Compiler: An opam plugin to manage compiler installations. https://github.com/ocaml-opam/opam-compiler. Accessed: 2026-06-29.

[6] OpenJDK. [n. d.]. JMH: Java Microbenchmark Harness. https://github.com/openjdk/jmh. Accessed: 2026-06-29.

[7] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. ACM, 31–47. doi:10.1145/3314221.3314637

[8] Romain Beauxis. [n. d.]. Regression with default GC settings between 4.14.2 and 5.1.1. https://github.com/ocaml/ocaml/issues/13123. Accessed: 2026-06-29.

[9] Runtime Events Tools. [n. d.]. Runtime Events Tools: A collection of observability tools around the runtime events tracing system introduced in OCaml 5.0. https://github.com/ocaml-opam/opam-compiler. Accessed: 2026-06-29.

[10] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP, Article 113 (2020). doi:10.1145/3408995

[11] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, 206–221. doi:10.1145/3453483.3454039

[12] The Rust Project. [n. d.]. rustc-perf: Rust Compiler Performance Monitoring and Benchmarking. https://github.com/rust-lang/rustc-perf. Accessed: 2026-06-29.

[13] Yaron Minsky. [n. d.]. The Saga of Multicore OCaml. https://www.youtube.com/watch?v=6VeJdUNhay4. Accessed: 2026-06-29.