Towards a Benchmarking Service for OCaml

Luis Eduardo de Souza Amorim and Tim McGilchrist

Tarides

Why?

- OCaml changes **fast**
- Changes may be **substantial**, e.g. OCaml 4 -> OCaml 5
- OxCaml makes things even more complex
- *What are the performance impacts of a particular change?*

OCaml needs a modern performance observability system with representative benchmarks, repeatable runs that collect GC/runtime metrics and presents clear regression feedback.

Vision

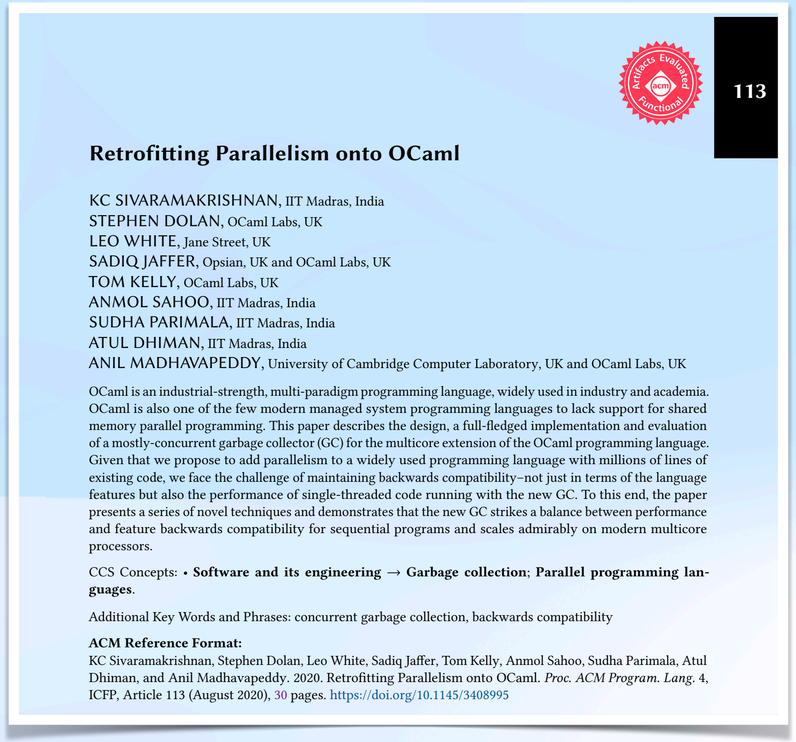
A publicly accessible, continuously-updated performance dashboard for the OCaml ecosystem, supporting:

- Developers tracking the impact of runtime/compiler PRs before merge
- Release teams comparing major versions (4.14 -> 5.x -> OxCaml)
- Developers studying GC behaviour under different workloads
- The community gaining confidence in OCaml's performance trajectory

Goals

- **Local-first, CI-ready**: same tool works for a developer benchmarking locally and for automated CI runs. No separate setup required.
- **Application runtime performance**: focus on the performance of generated code, not compiler throughput or build times.
- **Re-runnable on current hardware**: benchmarks should build and run on modern Linux distributions and hardware without requiring old platform versions.
- **Pluggable and extensible**: new benchmarks, new compiler versions, new GC parameters, new profiling tools can be added without restructuring the infrastructure.
- **Platform support**: Linux AMD64 and ARM64 as the primary CI targets. Other Linux architectures, FreeBSD and macOS supported as local targets.

Prior work



113

Retrofitting Parallelism onto OCaml

KC SIVARAMAKRISHNAN, IIT Madras, India
STEPHEN DOLAN, OCaml Labs, UK
LEO WHITE, Jane Street, UK
SADIQ JAFFER, Opsian, UK and OCaml Labs, UK
TOM KELLY, OCaml Labs, UK
ANMOL SAHOO, IIT Madras, India
SUDHA PARIMALA, IIT Madras, India
ATUL DHIMAN, IIT Madras, India
ANIL MADHAVAPEDDY, University of Cambridge Computer Laboratory, UK and OCaml Labs, UK

OCaml is an industrial-strength, multi-paradigm programming language, widely used in industry and academia. OCaml is also one of the few modern managed system programming languages to lack support for shared memory parallel programming. This paper describes the design, a full-fledged implementation and evaluation of a mostly-concurrent garbage collector (GC) for the multicore extension of the OCaml programming language. Given that we propose to add parallelism to a widely used programming language with millions of lines of existing code, we face the challenge of maintaining backwards compatibility–not just in terms of the language features but also the performance of single-threaded code running with the new GC. To this end, the paper presents a series of novel techniques and demonstrates that the new GC strikes a balance between performance and feature backwards compatibility for sequential programs and scales admirably on modern multicore processors.

CCS Concepts: • **Software and its engineering → Garbage collection**; **Parallel programming languages**.

Additional Key Words and Phrases: concurrent garbage collection, backwards compatibility

ACM Reference Format:
KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113 (August 2020), 30 pages. https://doi.org/10.1145/3408995

OCaml 4 -> OCaml 5
Sandmark

The DaCapo Benchmarks: Java Benchmarking Development and Analysis *

Stephen M Blackburn[α β], Robin Garner[β], Chris Hoffmann[γ], Asjad M Khan[γ], Kathryn S McKinley[δ], Rotem Bentzur[ε], Amer Diwan[ζ], Daniel Feinberg[ε], Daniel Frampton[β], Samuel Z Guyer[η], Martin Hirzel[θ], Antony Hosking[ι], Maria Jump[δ], Han Lee[α], J Eliot B Moss[γ], Aashish Phansalkar[δ], Darko Stefanović[ε], Thomas VanDrunen[κ], Daniel von Dincklage[ζ], Ben Wiedermann[δ]
[α]Intel, [β]Australian National University, [γ]University of Massachusetts at Amherst, [δ]University of Texas at Austin, [ε]University of New Mexico, [ζ]University of Colorado, [η]Tufts, [θ]IBM TJ Watson Research Center, [ι]Purdue University, [κ]Wheaton College

Abstract
Since benchmarks drive computer science research and industry product development, which ones we use and how we evaluate them are key questions for the community. Despite complex runtime tradeoffs due to dynamic compilation and garbage collection required for Java programs, many evaluations still use methodologies developed for C, C++, and Fortran. SPEC, the dominant purveyor of benchmarks, compounded this problem by institutionalizing these methodologies for their Java benchmark suite. This paper recommends benchmarking selection and evaluation methodologies, and introduces the DaCapo benchmarks, a set of open source, client-side Java benchmarks. We demonstrate that the complex interactions of (1) architecture, (2) compiler, (3) virtual machine, (4) memory management, and (5) application require more extensive evaluation than C, C++, and Fortran which stress (4) much less, and do not require (3). We use and introduce new value, time-series, and statistical metrics for static and dynamic properties such as code complexity, code size, heap composition, and pointer mutations. No benchmark suite is definitive, but these metrics show that DaCapo improves over SPEC Java in a variety of ways, including more complex code, richer object behaviors, and more demanding memory system requirements. This paper takes a step towards improving methodologies for choosing and evaluating benchmarks to foster innovation in system design and implementation for Java and other managed languages.

Categories and Subject Descriptors C.4 [*Measurement Techniques*]
General Terms Measurement, Performance
Keywords methodology, benchmark, DaCapo, Java, SPEC

1. Introduction

When researchers explore new system features and optimizations, they typically evaluate them with benchmarks. If the idea does not improve a set of interesting benchmarks, researchers are unlikely to submit the idea for publication, or if they do, the community is unlikely to accept it. Thus, benchmarks set standards for innovation and can encourage or stifle it.

For Java, industry and academia typically use the SPEC Java benchmarks (the SPECjvm98 benchmarks and SPECjbb2000 [37, 38]). When SPEC introduced these benchmarks, their evaluation rules and the community's evaluation metrics glossed over some of the key questions for Java benchmarking. For example, (1) SPEC reporting of the "best" execution time is taken from multiple iterations of the benchmark within a single execution of the virtual machine, which will typically eliminate compile time. (2) In addition to steady state application performance, a key question for Java virtual machines (JVMs) is the tradeoff between compile and application time, yet SPEC does not require this metric, and the community often does not report it. (3) SPEC does not require reports on multiple heap sizes and thus does not explore the space-time tradeoff automatic memory management (garbage collection) must make. SPEC specifies three possible heap sizes, all of which over-provision the heap. Some researchers and industry evaluations of course do vary and report these metrics, but many do not.

This paper introduces the DaCapo benchmarks, a set of general purpose, realistic, freely available Java applications. This paper also recommends a number of methodologies for choosing and evaluating Java benchmarks, virtual machines, and their memory management systems. Some of these methodologies are already in use. For example, Eeckhout et al. recommend that hardware vendors use multiple JVMs for benchmarking because applications vary significantly based on JVM [19]. We recommend and use this methodology on three commercial JVMs, confirming none is a consistent winner and benchmark variation is large. We recommend here a deterministic methodology for evaluating compiler optimizations that holds the compiler workload constant, as well as the standard steady-state stable performance methodology. For evaluating garbage collectors, we recommend multiple heap sizes and deterministic compiler configurations. We also suggest new and previous methodologies for selecting benchmarks and comparing them. For example, we recommend time-series data versus single values, including heap composition and pointer distances for live objects as well as allocated objects. We also recommend principal component analysis [13, 18, 19] to assess differences between benchmarks.

Java Benchmarking
DaCapo suite

Inspired by Sandmark for tracking OCaml evolution and DaCapo for using real-world macro-benchmarks.

Micro-benchmarks

https://github.com/ocaml-bench/benches

- Extract Sandmark benchmarks into a standalone project
- Approximately 200 microbenchmarks
- Help with stress testing the runtime and capture specific regressions but do not capture user experience.
- Is this useful? Who will maintain this?
- More micro-benchmarks in PRs and Issues

Macro-benchmarks

https://github.com/ocaml-bench/macro-benches

- DaCapo-style OCaml macro-benchmarks based on realistic applications
- Capture regressions that may be observed by users
- Target application areas that are important to OCaml
- 20 tools, 33 distinct workloads with configurable running times
- 13 different categories e.g. Compilers, Static analysis, Web, Proof assistants.
- 11 different runtime features e.g. ephemerons, effects, C stubs, signals, custom blocks, off-heap allocation.

What we have so far?

1 Input

- Micro & macro benchmarks
- Runtime versions & variants
- GC parameter sweeps
- Comparison matrices

2 Orchestrator (running-ng)

- Build runtimes
- Prepare environment
- Build binaries
- Execute runs
- Collect metrics

3 Output

- JSON sidecars
- Runtime metrics
  - Wall time
  - RSS
  - Olly & Perf statistics
- Metadata

4 Analysis

Notebooks

- Heatmaps
- Version comparisons
- Pareto graphs

LLM reports

- Initial analysis
- Improvement/ regression alerts
- Tables & summaries

5 Insights

- Detect regressions early
- Understand trade-offs
- Find optimal GC settings
- Inform developers

YAML Configs

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

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

configs:
  - "ocaml-5.5.0|perf_grp1|re-25|md-2"
  - “ocaml-5.5.0-fp|perf_grp1|re-25|md-2”
  - “ocaml-5.5.0-fp_flambda|perf_grp1|re-25|md-2”

comparisons:
  - label: “frame pointer comparison"
    a: ocaml-5.5.0
    b:
     - ocaml-5.5.0-fp
     - ocaml-5.5.0-fp-flambda
```

Python Notebooks

A — Regression Dashboard

Audience: anyone asking *"did this compiler change regress or improve the benchmarks?"*

Structure (Option C — per-comparison). Every section under §4 corresponds to one comparison block declared in `runbms.yml` (or supplied via `COMPARISONS_OVERRIDE`). Each block is a self-contained story: instruction-count deltas, wall-time and max-RSS tornado plots per pair, a tradeoff scatter restricted to the block's variants, and ranked top-N regressions/improvements.

When `runbms.yml` declares no `comparisons:` block, the loader falls back to a single default block (`BASELINE` vs every other variant) — the rendered output is then equivalent to a traditional single-baseline view.

Configuration knobs are in the next cell. See README.md §5.6 for the YAML schema.

C — GC Parameter Sweep

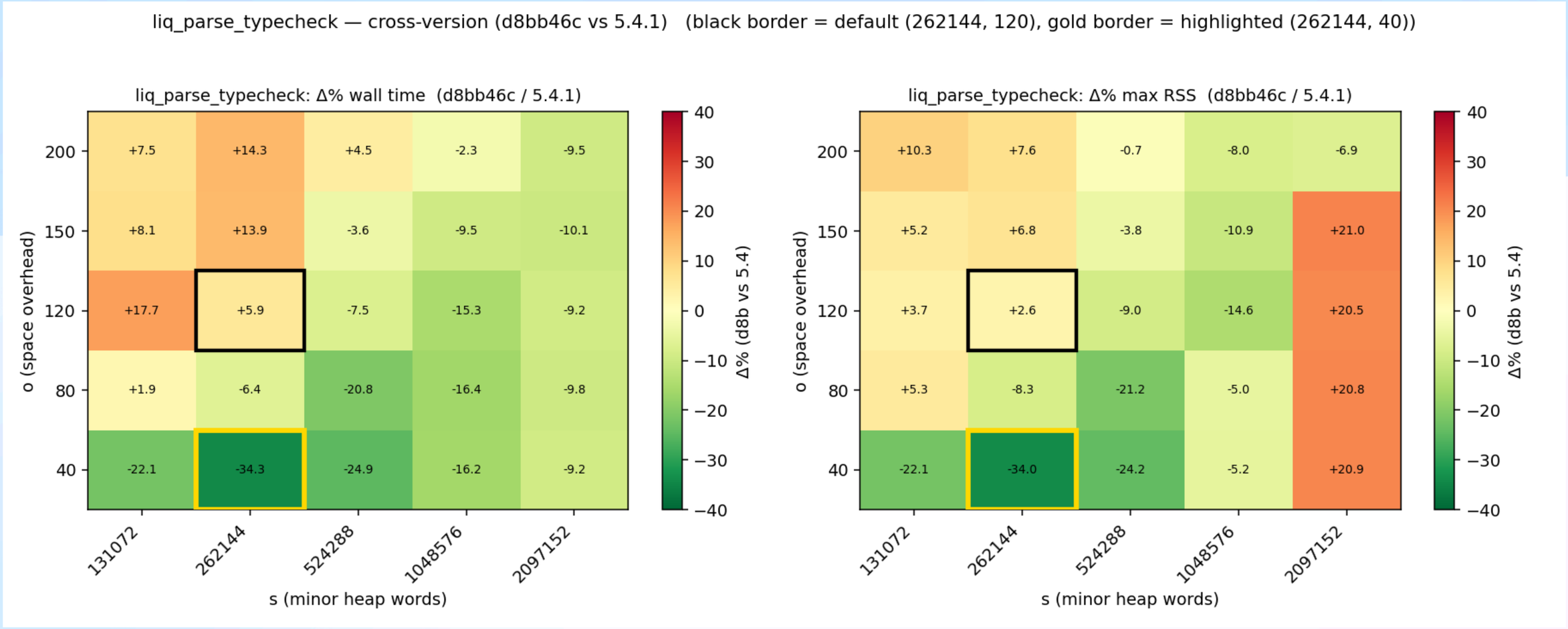
Audience: GC/runtime researcher or release engineer.

This notebook is organised around **three explicit questions**. Each section is labelled with the question it answers; cells inside a section share a reference point and an aggregation level.

§	Question	Reference point	Aggregation	Audience
3	**Q1 — Better than default for this workload?**	this runtime's default cell	per-benchmark	app developer
4	**Q3 — Should we ship a new default?**	this runtime's default cell	geomean across benchmarks (+ worst-case shown)	release engineer
5	**Q2 — How does this config affect a cross-runtime comparison?**	the *other* runtime at the *same* cell	per-benchmark, per-cell	runtime maintainer

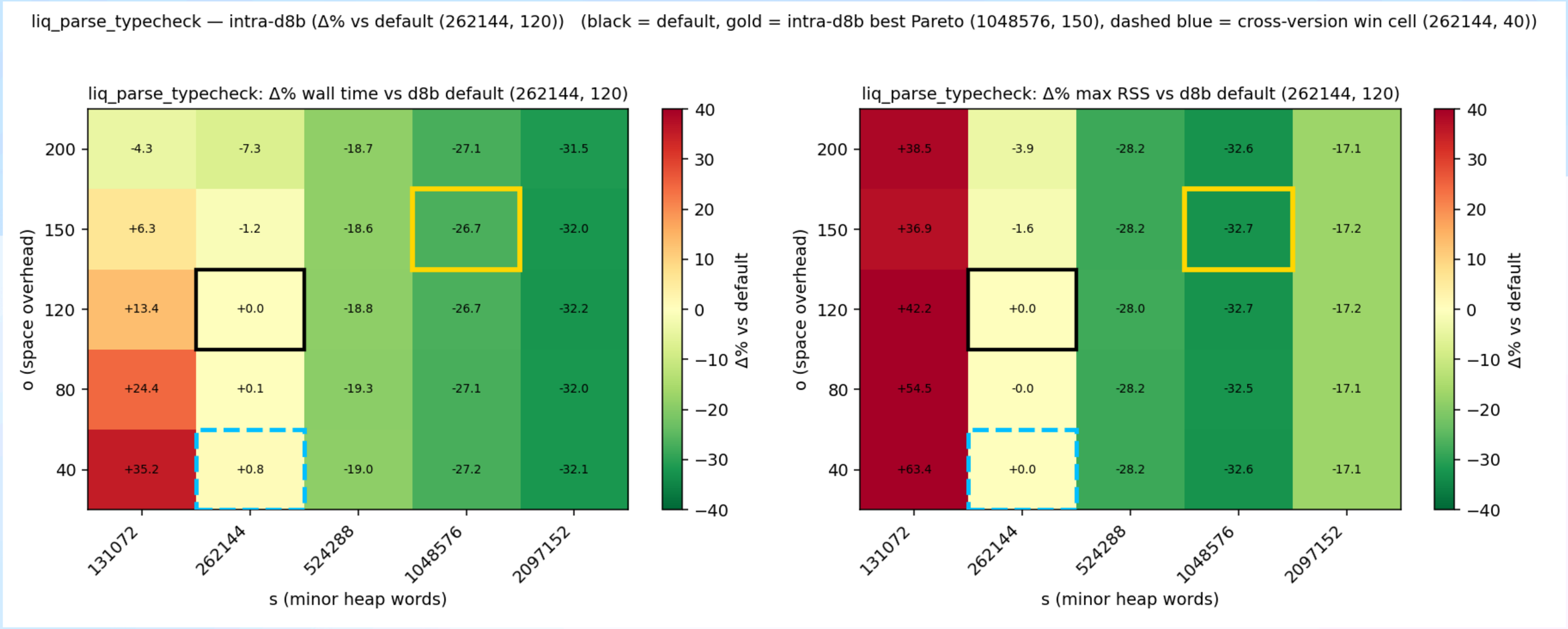
Early Results

Regression on Default Parameters



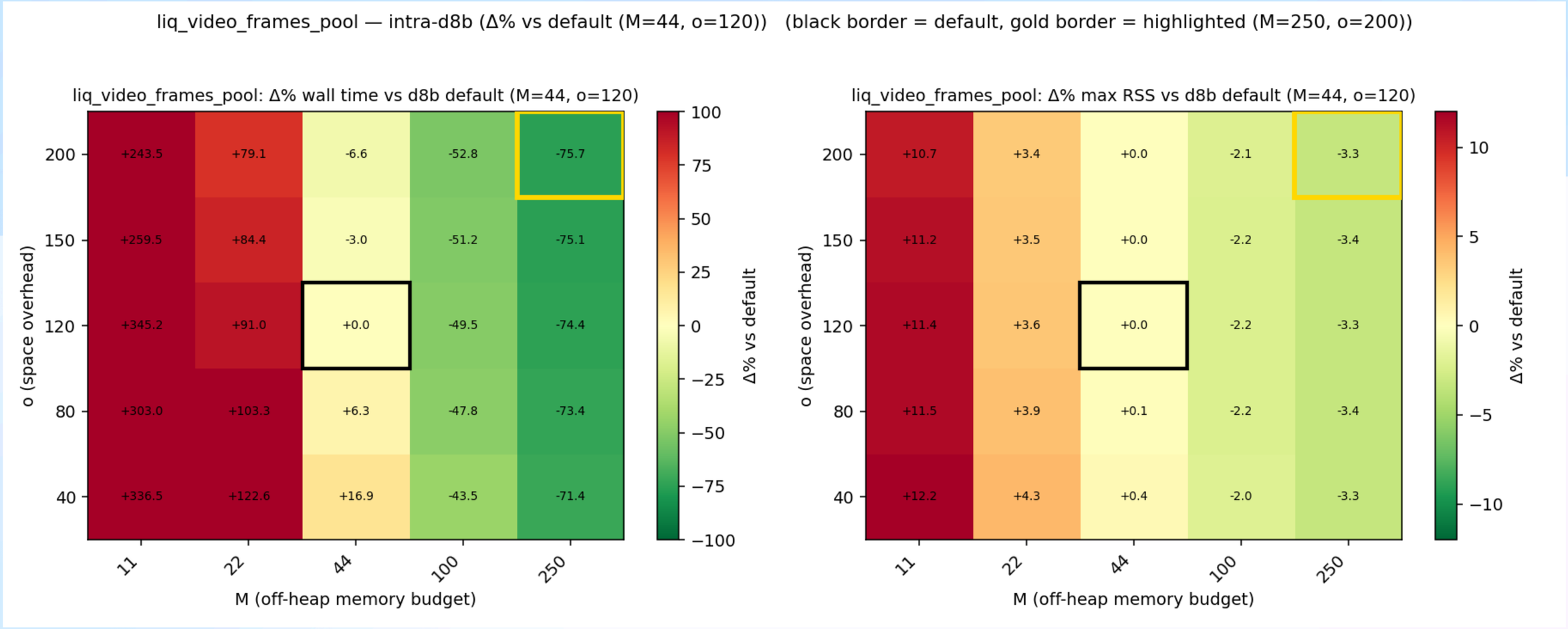
Early Results

Intra-runtime Parameter Sweep



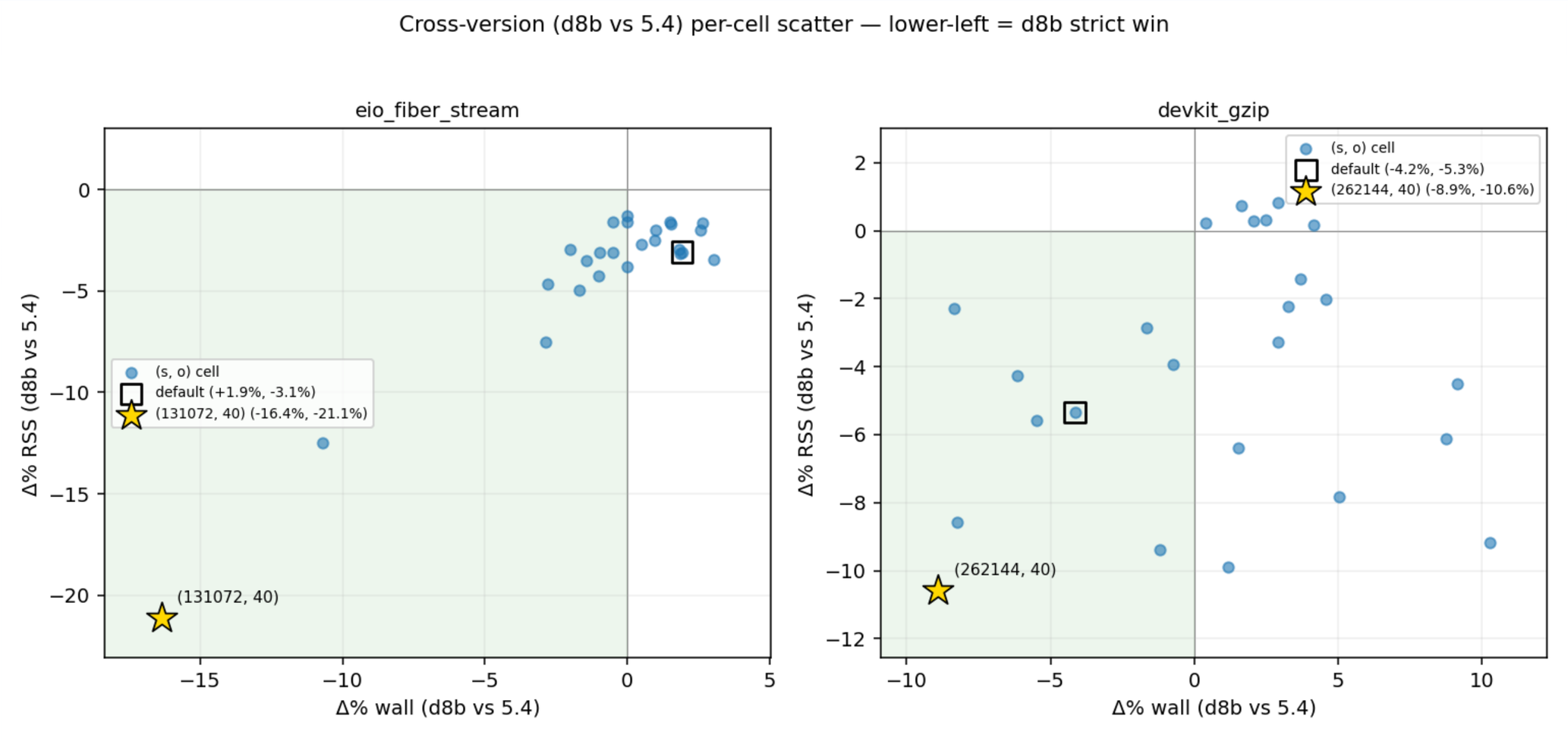
Early Results

Tracking OCaml Issue #14533



Early Results

Pareto: tracking time/space tradeoff



Early Results

Faster Minor GC #14751

OCaml Benchmark Dashboard

Machine

Host	CPU
monolith	AMD Ryzen 9 9950X 16-Core Processor
Cores	Kernel
32	6.17.0-41-generic

Run

Run	Created
monolith-2026-08-18-Tue-044952	2026-08-18T09:52:10Z
Produced by	Tools
running-ng (native)	perf 6.17.13 · olly 0.5.4-24-g8d3d6cc
Runtimes	trunk-3b14dfbc · pr14571

Regression view. Pick a **baseline** and a **comparison** — each is a runtime, plus a value for each swept parameter (`space_overhead`, `minor_heap`, `gc_plan`, …) if the run has any. Defaults to the comparison declared in the run manifest, but any pair works (e.g. LXR vs Bactrian, or trunk vs the PR at a chosen `(s, o)`). Negative Δ is better; thresholds ±1% warn, ±3% regression/improvement.

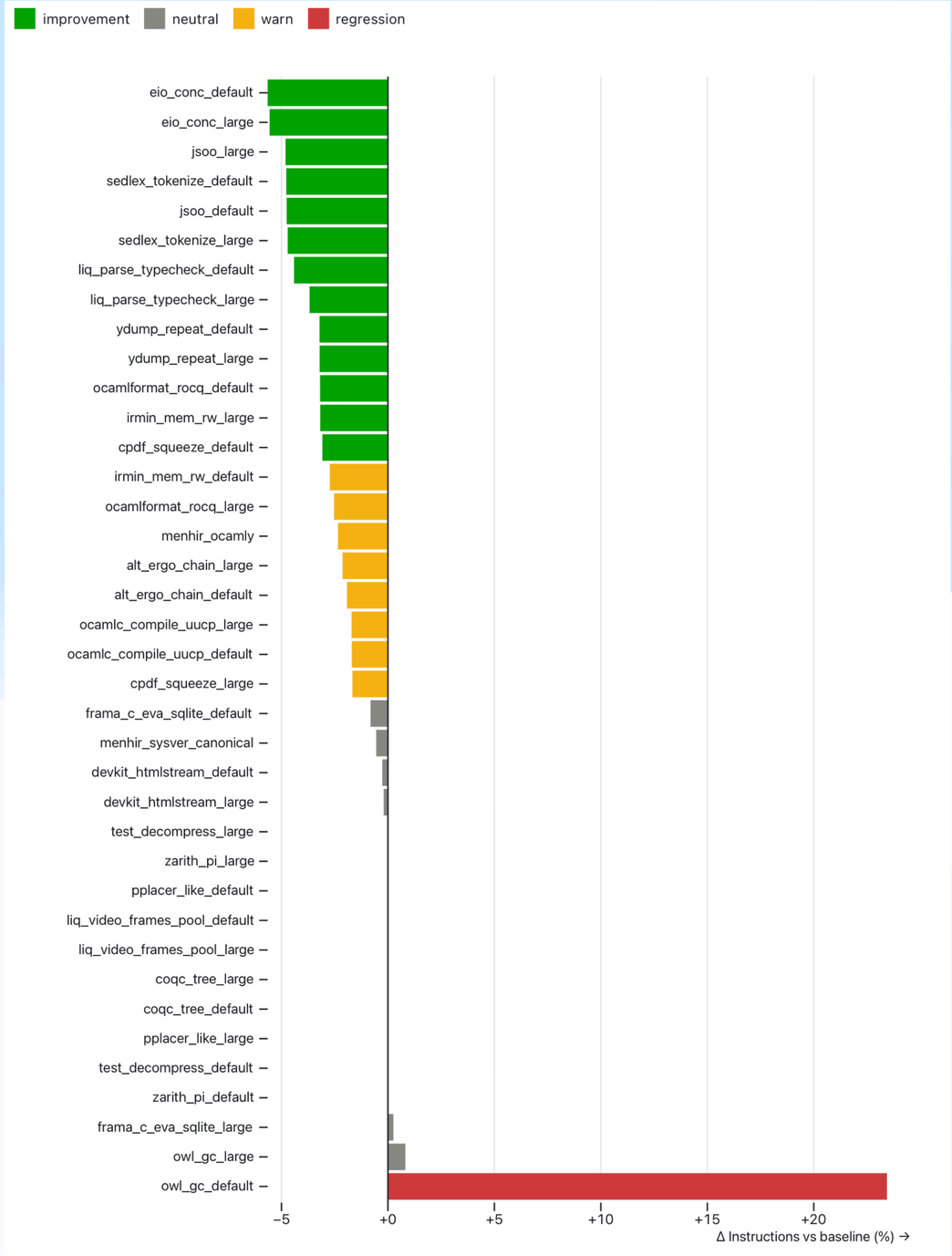
Metric: Instructions

Baseline

runtime: trunk-3b14dfbc

Compare

runtime: pr14571

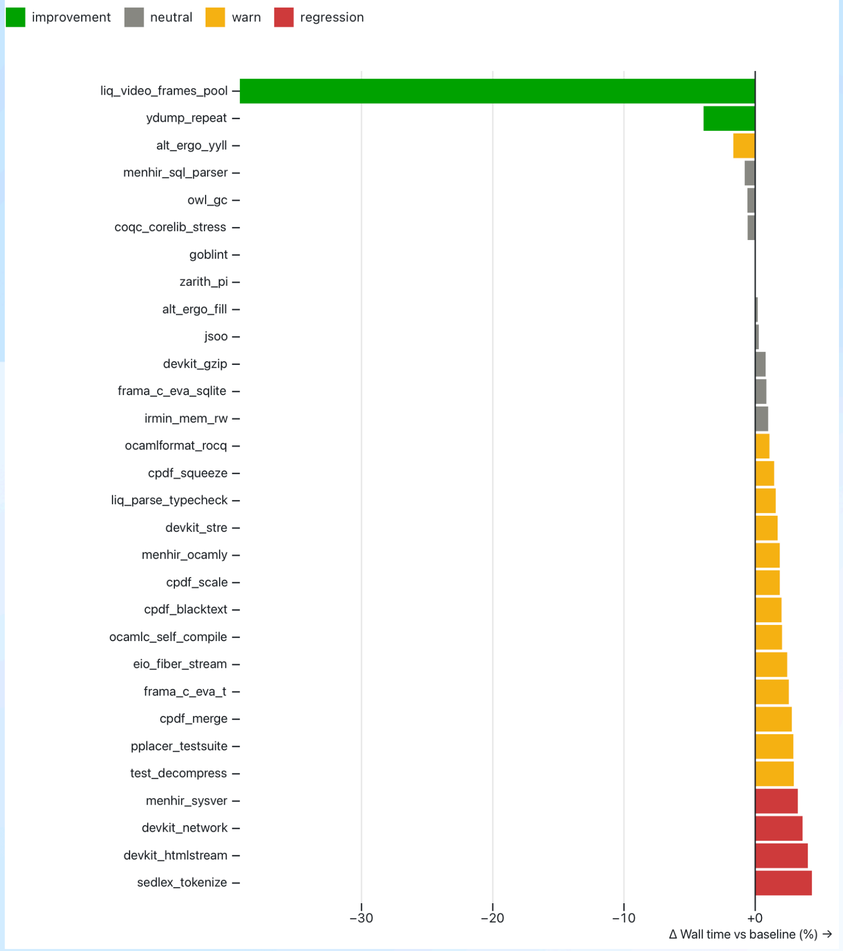


Early Results

Measuring Runtime Variants

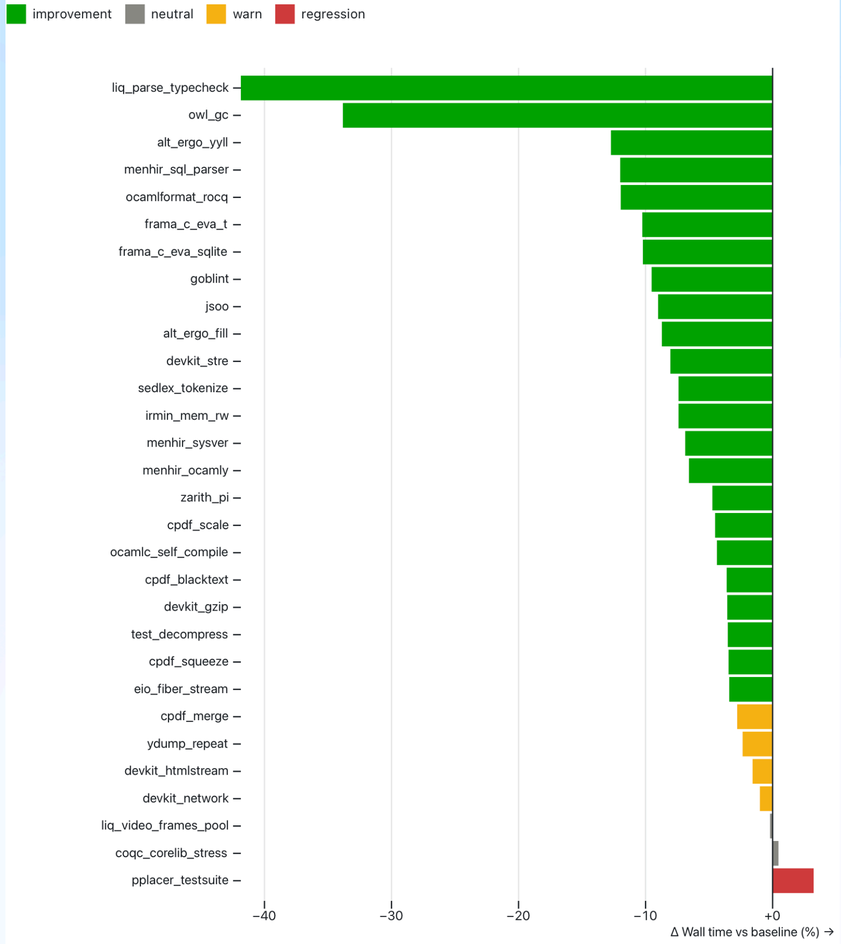
5.5.0-fp vs 5.5.0

2 improvement(s), 4 regression(s) across 30 benchmark(s).



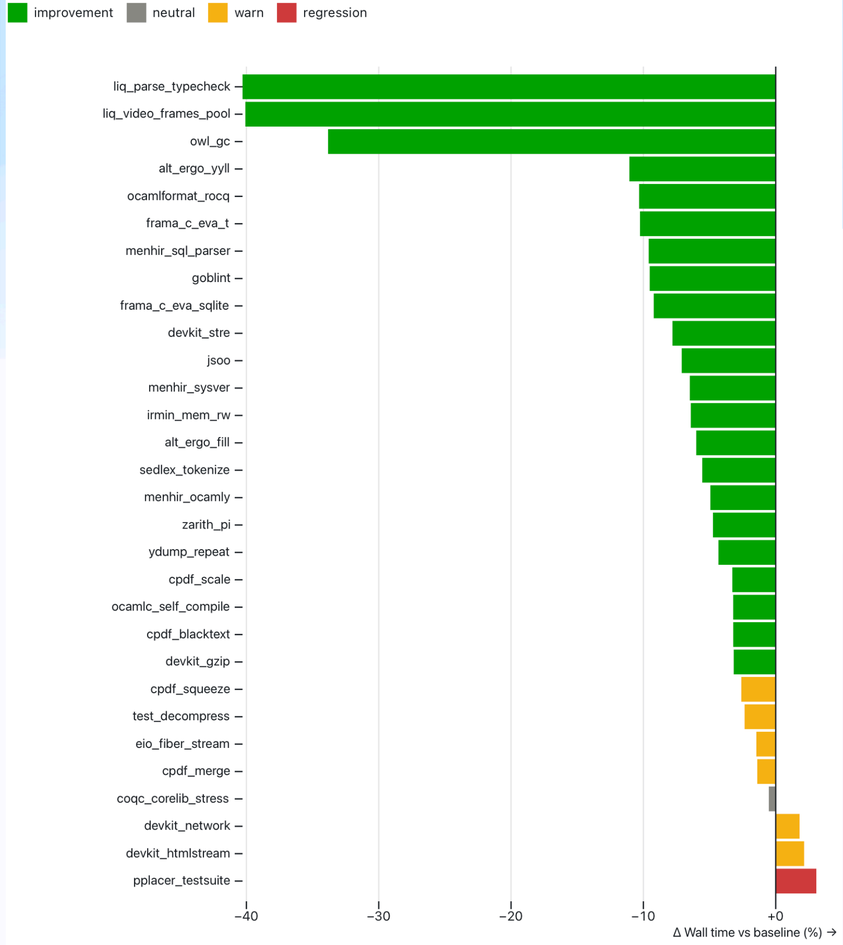
5.5.0-flambda vs 5.5.0

23 improvement(s), 1 regression(s) across 30 benchmark(s).



5.5.0-fp-flambda vs 5.5.0

22 improvement(s), 1 regression(s) across 30 benchmark(s).



Early Results

New GC-pacing logic #14796

OCaml Benchmark Dashboard

Machine

Host	CPU
monolith	AMD Ryzen 9 9950X 16-Core Processor
Cores	Kernel
32	6.17.0-19-generic

Run

Run	Created
monolith-2026-07-18-Sat-025524	2026-07-18T17:46:32Z
Produced by	Tools
running-ng (native)	perf 6.17.13 · olly 0.5.4-12-g3d37a0b

Runtimes trunk-e53a0322 · gc-pacing-72c712d4

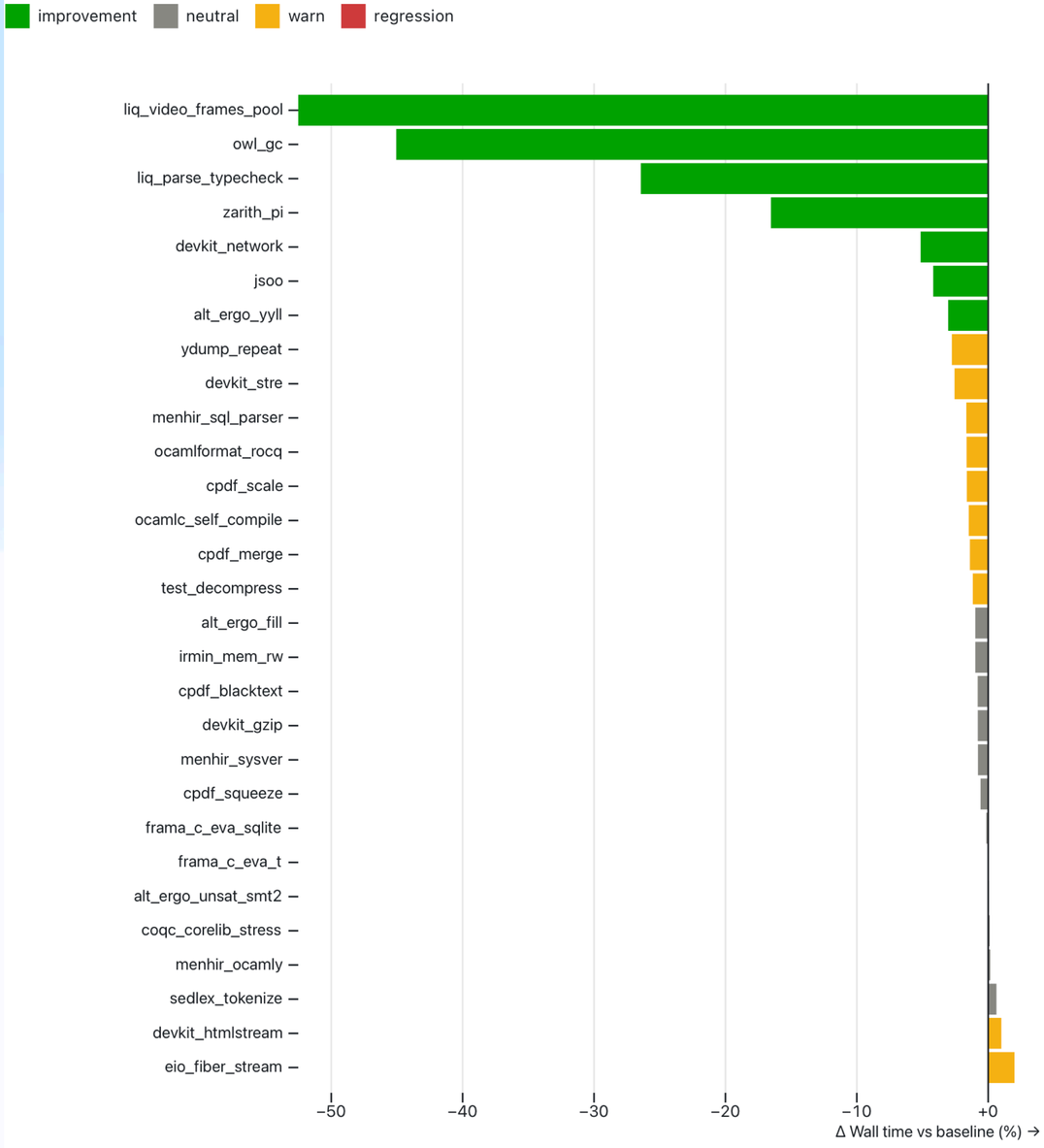
Regression view: each **comparison declared in the run manifest** is rendered below. Pick the metric to compare; negative Δ is better (fewer instructions, less time). Thresholds: ±1% warn, ±3% regression/improvement.

If this run swept GC parameters, pick the point to compare at — otherwise the same runtime shows up once per parameter combination. (Parameters with a single value are omitted; use Parameter sweeps to see the whole grid.)

Metric	Wall time
minor_heap	262,144
space_overhead	120

gc-pacing PR vs trunk merge-base — (s, o) sweep 2026-07-13

7 improvement(s), 0 regression(s) across 29 benchmark(s).



Outcomes

- Benchmarks are runnable locally on Linux
- Extracted suite of micro benchmarks from Sandmark
- New suite of macro benchmarks in the DaCapo-style
- Improvements to runtime events implementation e.g. #13011, #14189, #14522, #14966, #14969, #15000, #15001, #15010, #15016
- Improvements to profiling tools like runtime_events_tools and perf
- Deeper analysis of runtime changes e.g. #14571, #14796

Future work

- Better *Benchmark coverage:*
  - *runtime features e.g. c*ompaction, multi-domain, OxCaml
  - principal component analysis
- *Infrastructure*: Setting up service to run benchmarks on tuned machines
- Improving runtime events *metrics*: memory fragmentation, STW cost
- Support *local running* on more platforms: macOS, FreeBSD, Windows

Benchmarking OCaml

Thank you!

running-ng

macro-benches

benches

OCaml