Pass-Order Sensitivity in a WebAssembly Teaching Compiler (Tubular)

Aditya Prashant Chaudhari
Independent Research Technical Report

2025

Abstract

We study how the ordering of three classic compiler optimizations—function inlining, loop unrolling, and tail-recursion elimination—affects performance in a small WebAssembly compiler (Tubular). Across 10 curated kernels and 6 optimization variants, compiled under 6 pass orders and repeated 3 times (1080 total runs), the global mean difference between the best and worst fixed orders is only 0.25 %. Yet, specific structures (nested/stride-heavy loops or larger unrolls) show 3 % to 5 % mean best–worst gaps, with outliers approaching 10 %. We quantify robustness using regret distributions and show that a single order (`inline-tail-unroll`) minimizes mean and 95th-percentile regret, while per-benchmark winners vary, motivating per-program selection.

1 Introduction

Compiler pass ordering is a long-standing problem: classic analyses show that no universal order dominates across programs [1, 2]. Recent work explores search- and learning-based sequence selection that models interactions among passes [3, 4]. Tubular is a didactic compiler I developed for a small language that emits WebAssembly, making it ideal for controlled, reproducible experiments on pass-order effects.

This report asks: *When does pass order matter for small Wasm kernels, and by how much?*

2 Experimental Setup

Benchmarks. We use 10 kernels from Tubular's research suite (e.g., recursive Fibonacci, tail-factorial, loop summation, nested/stride-heavy loops).

- rt01-fib-recursive (exponential recursion)
- rt02-tail-factorial (tail recursion)
- rt03-loop-summation (simple loop)
- rt04-stride-heavy-loop (calls to heavier helpers)
- rt05-nested-mix (nested loops)
- rt06-string-wrapping (helper calls)
- rt07-helper-inline (inlining stress test)

- rt08-branchy-loop (control-heavy)
- rt09-matrix-mix (matrix-style nested loops)
- rt10-control-baseline (minimal optimization)

Each benchmark has six optimization variants combining inlining toggles and unroll factors. We evaluate all permutations of three passes (`inline`, `unroll`, `tail`), for 6 distinct orders, with 3 repeated collections.

Reproducibility. All experimental scripts and raw data are available at `https://github.com/AdityaC4/tubular-upgrade`. The data collection and repetition pipelines are implemented in `scripts/collect_data.py` and `scripts/repeat_collection.py`, while all CSV summaries and figures used in this report reside under `docs/figures/` and `artifacts/research/`. Each dataset is versioned and can be regenerated in a single command: `python3 scripts/collect_data.py`.

All timings were collected on a Windows 11 system running WSL2 (Debian 12) on an AMD Ryzen 7 7800X3D with 32 GiB DDR5 6000 MT/s RAM, using `Node.js v24.6.0` and `C++20` for codegen. CPU frequency scaling was pinned to performance mode; each benchmark included 10 warm-up iterations followed by 50 timed runs; we report the median of 50 runs per configuration. The entire experiment was repeated 3 times to assess stability.

Metrics. For each configuration we report median runtime (ms). We analyze: (i) global mean runtime by order; (ii) per-row *regret* relative to the best order for that (run, benchmark, variant); and (iii) best–worst gaps ($\Delta\%$).

3 Results

3.1 Risk Profile via Regret

Figure 1 plots the CDF of regret across orders. `inline-tail-unroll` is the most robust fixed order (mean regret 0.915 %, p95 2.400 %), even though it does not win every head-to-head comparison.

3.2 Who Wins Where

Figure 2 shows the share of wins per benchmark; bars sum to 100%. Columns with a clear dominant color indicate stable winners; fragmented columns suggest per-program selection.

3.3 How Decisive is the Winner?

Figure 3 shows, for each benchmark, the fraction of wins captured by its best order. Long bars indicate decisive winners; short bars indicate contested benchmarks.

3.4 Sensitivity vs. Unroll

Larger unroll factors modestly increase sensitivity. Figure 4 shows the distribution of best–worst gaps ($\Delta\%$) for unroll factors 0, 4, and 8; the medians rise with the factor.

4 Related Work

Classic texts emphasize that there is no universally optimal fixed pass order across programs and targets; effectiveness depends on program structure and interactions among optimizations [1, 2]. Complementary to this, surveys and recent methods develop search- or learning-based strategies

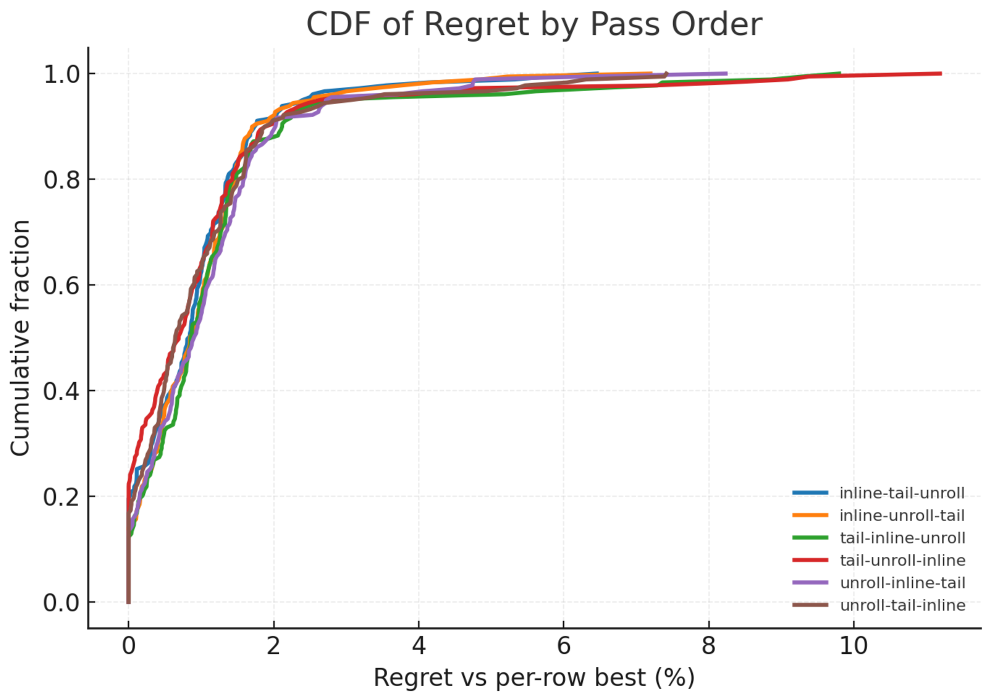

Figure 1: CDF of regret vs. per-row best across pass orders (further left and higher is better).

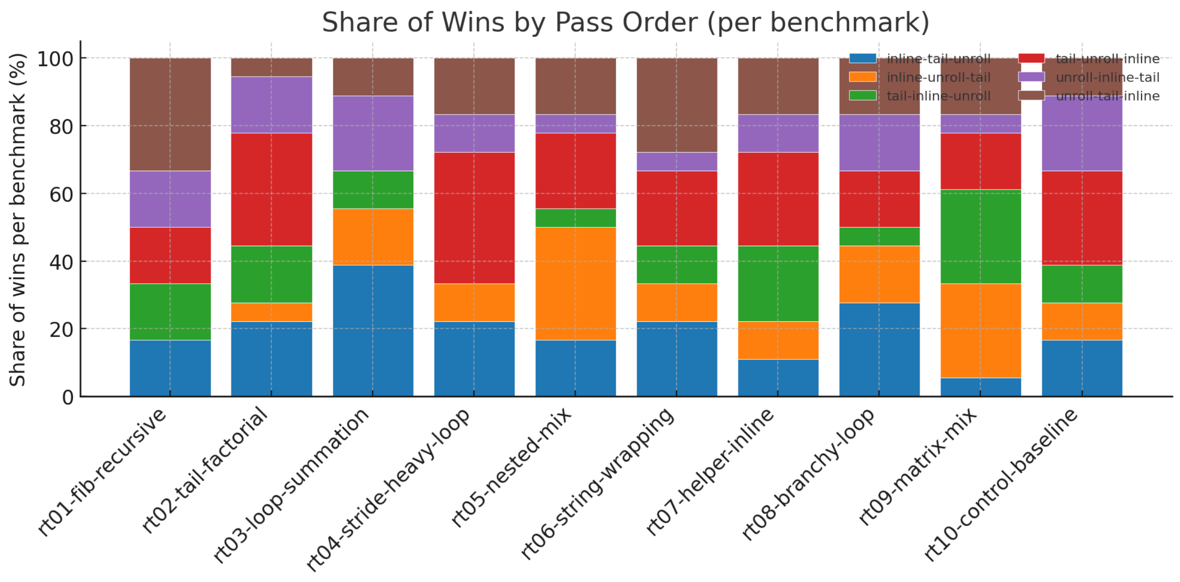

Figure 2: Share of wins by pass order for each benchmark (stacked to 100%).

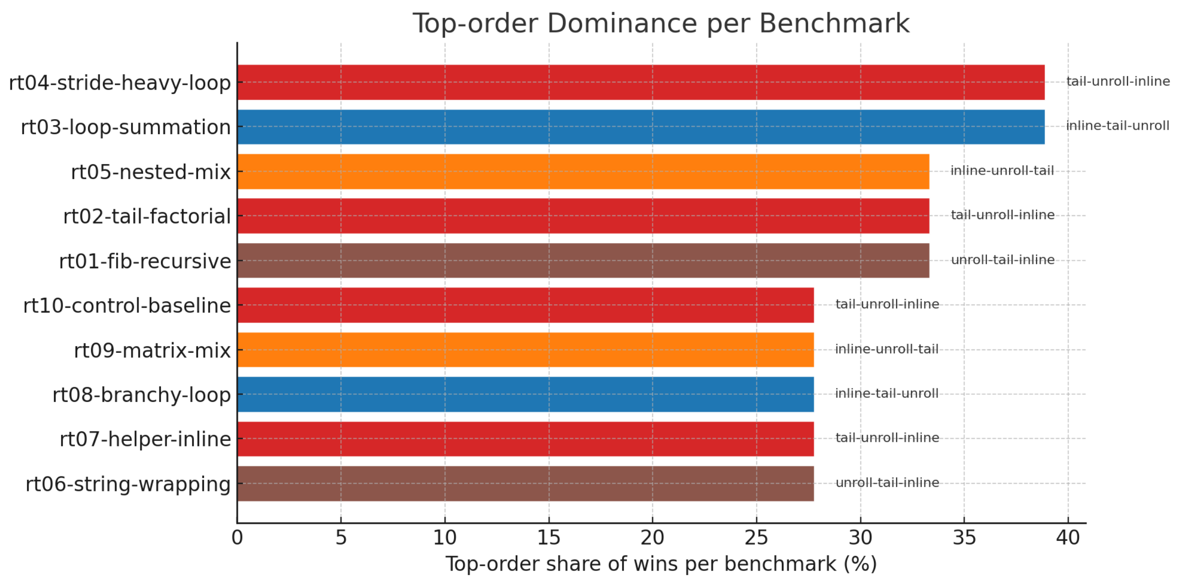

Figure 3: Top-order dominance per benchmark (bar length = % of wins for the best order).

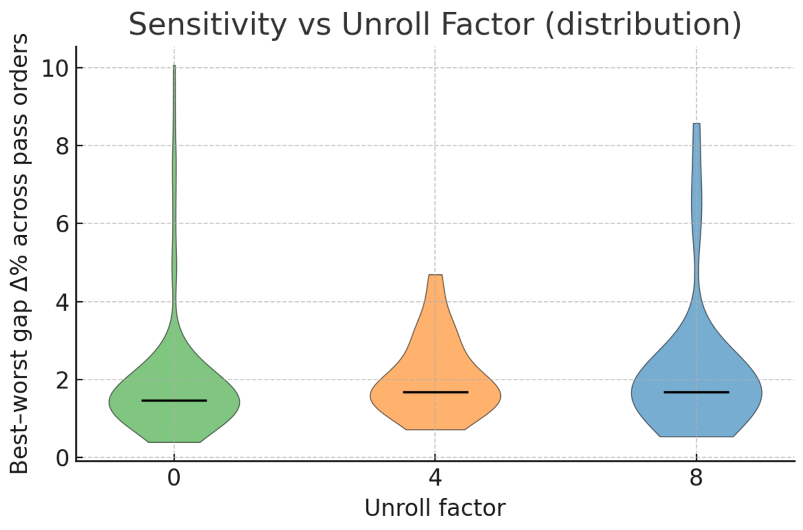

Figure 4: Sensitivity vs. unroll factor: distributions of best–worst gaps across orders. Solid bar = median.

that model pass dependencies and optimize sequences for average-case performance across workloads [3, 4]. Our study contributes a small, fully reproducible data point: in a constrained Wasm setting the *global* spread among fixed orders is modest, yet we observe *structure-specific sensitivity* (e.g., nested/stride-heavy loops and higher unroll factors). We operationalize robustness via regret, highlighting an angle orthogonal to maximizing mean speedup: identifying when per-program selection is warranted and which fixed order is safest when a single policy is required.

5 Summary Tables

Pass order	**Global mean runtime (ms)**
inline-tail-unroll	19.9338
inline-unroll-tail	19.9452
unroll-tail-inline	19.9501
tail-unroll-inline	19.9521
unroll-inline-tail	19.9648
tail-inline-unroll	19.9839

Table 1: Global mean runtime by pass order (lower is better). Spread between best and worst is 0.25 %. Global mean runtime = mean over all (benchmark, variant) rows of the median runtime per order.

Pass order	**Mean (%)**	**Median (%)**	**P95 (%)**	**P99 (%)**	**Max (%)**
inline-tail-unroll	0.915	0.826	2.400	5.419	6.456
inline-unroll-tail	0.972	0.848	2.530	4.927	7.196
unroll-tail-inline	1.000	0.642	3.071	6.535	7.415
tail-unroll-inline	1.011	0.689	2.638	9.153	11.188
unroll-inline-tail	1.074	0.900	2.723	5.098	8.232
tail-inline-unroll	1.171	0.858	2.775	8.948	9.797

Table 2: Regret statistics over all (run, benchmark, variant) rows (lower is better).

Benchmark	**Variant**	**Mean Δ%**
rt04-stride-heavy-loop	combo-inline-unroll	4.47
rt05-nested-mix	no-inline	4.01
rt05-nested-mix	unroll-8	3.95
rt04-stride-heavy-loop	unroll-8	3.67
rt05-nested-mix	combo-inline-unroll	3.51
rt04-stride-heavy-loop	baseline	3.40
rt03-loop-summation	no-inline	3.38
rt03-loop-summation	unroll-8	3.22
rt02-tail-factorial	unroll-4	3.01
rt04-stride-heavy-loop	no-inline	2.86

Table 3: Top-10 most pass-order-sensitive benchmark/variant pairs by mean best–worst gap.

6 Discussion

The average effect of pass ordering is small, but the *selective sensitivity* is real. Nested and stride-heavy loops, along with higher unroll factors, amplify interactions between inlining, unrolling, and tail-call optimization. Taken together, the stacked shares and dominance bars show that some benchmarks exhibit a stable fixed order, while others are contested; the regret CDF provides a principled way to select a *robust* default (`inline-tail-unroll`) when a single order must be chosen. These observations align with the broader literature—no single order dominates across programs [1, 2]—and complement sequence-selection approaches that target average-case gains [3, 4].

Limitations. Micro-kernels may underrepresent real-world programs; we used one toolchain and runtime. Future work should replicate these trends under Wasmtime/JIT settings, add additional passes (e.g., DCE/CSE), and evaluate simple learned/heuristic selectors for per-program order choice.

7 Conclusion

We present a reproducible, data-driven study of pass-order sensitivity in a teaching WebAssembly compiler. Global means differ by only 0.25 %, yet specific structures show 3 % to 5 % mean gaps, with occasional outliers near 10 %. A single order (`inline-tail-unroll`) minimizes regret, but *per-program selection* is the clearest path to improvement.

References

[1] K. D. Cooper and L. Torczon. *Engineering a Compiler*, 2nd ed. Morgan Kaufmann, 2011.

[2] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[3] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys*, 51(5), 2018.

[4] J. Liu, J. Fang, T. Wang, J. Xie, C. Huang, and Z. Wang. Efficient compiler optimization by modeling passes dependence. *CCF Transactions on High Performance Computing*, 6(6):588–607, 2024.