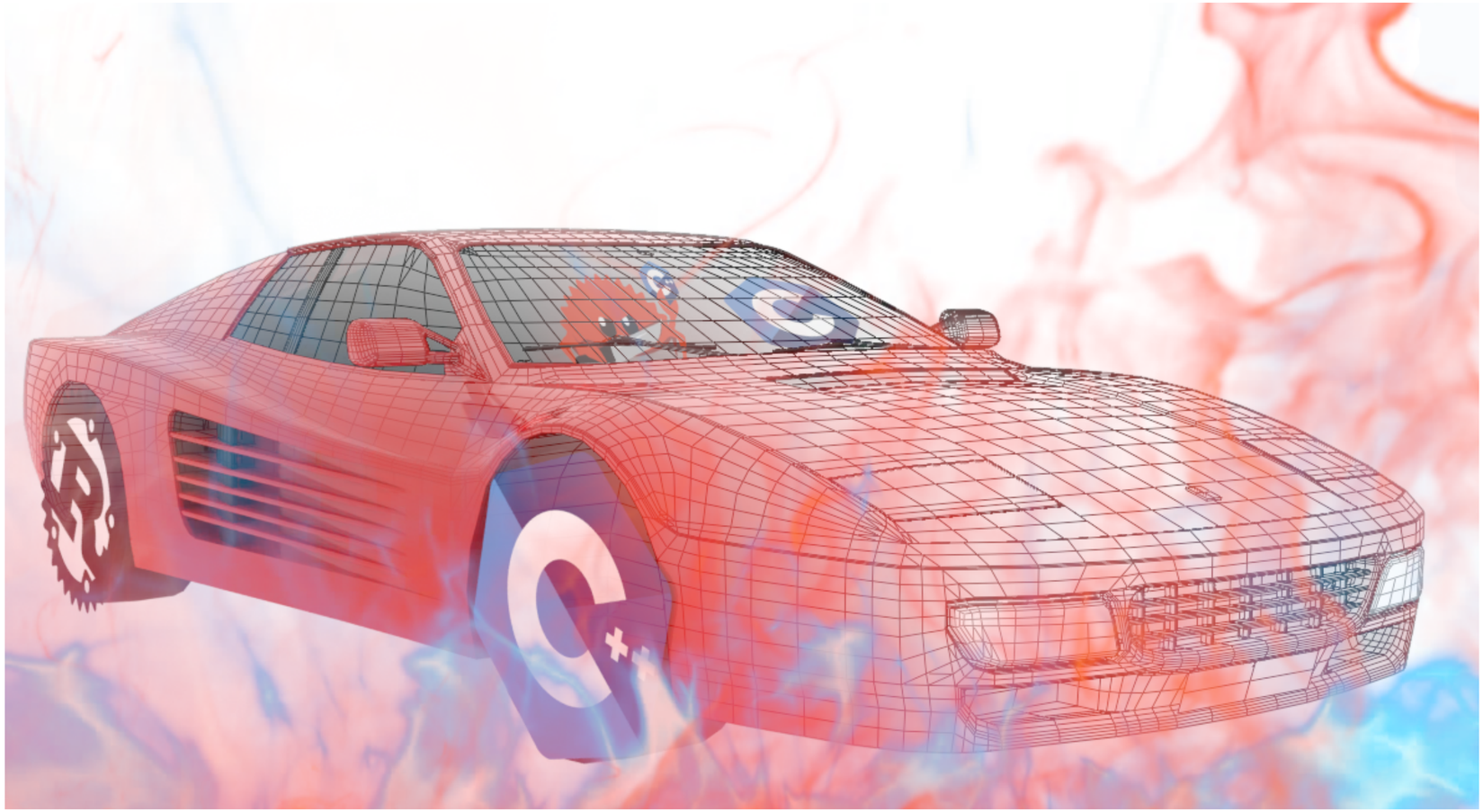
Effizientes Programmieren in C, C++ und Rust

Performance: Best Practices

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024



Summary

Modelling

- There is no single best performance criterion or performance model
- Modelling, improving and assessing performance may employ best-practices of engineering ("performance is refactoring") and (falsification, validation ...)
- Performance models should at least consider run time and intensity to describe the Neumann bottleneck
- All models are wrong, some are useful

Summary

Profiling

- Don't trust dudes (mindlessly)
- Humans tend to overestimate their ability in telling what matters. Programmers are human (after all)
- Data dominates*. Understand data to understand the impact of algorithms on performance
- Profiling is sampling
- Branch and bottleneck
- Internal measurements help to understand context-dependent behavior
- Learn to use your tools. A fool with a tool remains a fool.

Summary

Benchmarking

- Benchmarks should be relevant, reproducable, fair, verifiable and usable (use real world instances where possible)
- Prefer clock rates over IPS. None of these are reliable.
- Benchmark environments are noisy. Noise is annoying, systematic noise is dangerous.
- Data is like underwear.
- Commit no sins.
- Commit no crimes.

Statistics

- Arithmetic means are not robust. Consider geometric means.
- Without variance, we cannot assess significance
- Statistical tests are not interpreted by looking at alpha and p. Use where appropriate.
- Bonferroni correction!

Summary

Moore's Law

- We may no longer rely on hardware designers to make our programs go faster with each new hardware generation
- Moore's Law is up to programmers

Amdahl's Law

- Measure first, optimize second
- Even better: think first, measure second, optimize third (e.g., for memory layout)

Liebig's Law

- The weakest link in the "performance chain" limits overall performance

Summary

Efficient Programming

- Optimization Strategies
  - Parallelize
  - Localize
  - Pipeline
  - Branch and bound
- Learn to work with your efficient programming language of choice and keep up to date
  - E.g., participate in code challenges
  - Apply what you have learned in this lecture to your own projects, BA thesis, ...

Summary

Pitfalls and Fallacies

- Hardware designers won't (longer) fix your slow performance with new generations of technology
- Benchmarks don't remain valid indefinitely
- Peak performance rarely equals observed performance
- It is hard to fix bad memory layout "in post"
- Software engineering and efficient programming don't exclude one another (YAGNI, KISS, DRY, ...)
- The human factor matters
- Your time is more valuable than execution time (spend it wisely)

Quantitative Principles

- Take advantages of parallelism
- Exploit the principle of locality ("90% of execution time are spent in 10% of the code")
- Focus on the most common case; make no unjustified claims about what that case is

Summary

Is this code good?

Best Practices

Algorithms & Datastructures

- Asymptotic complexity vs. constant factors
- Small input vs. large input
- Worst case/avg. case/best case
- Implementation complexity
- Exploration and benchmarking

Know Your Data

- Properties of real-world data
- Invariants vs. statistical properties
- Adjust algorithms to data, adjust data to algorithms?
- Special casing

Profiling, Compilers, Manual Optimization

- Question: Where to optimize?
- Measure and verify
- Cold and hot code, bottlenecks
- Compiler limitations

Know Your Hardware

- Compute-bound, memory-bound, IO-bound
- Branch misses, cache misses, synchronization overhead
- Branchless instructions, SIMD, prefetching, ...

Best Practices

A Semi-Ordered List of Tricks and Hints

Profile first!

- Avoid syscalls and I/O
  (use buffers/accumulate)
  - *E.g., database calls in backends*
- Avoid allocations
  (use buffers/accumulate)
  - *E.g., returning lists from functions*
- Assertions!
- Use optimizations flags & hints
  - `march=native`, LTO, `always_inline`
- Use "fast paths" where possible
- Cache repeated computations
- Determine data layout first,
  code afterwards
- Don't be afraid to reorder/transform your data
  - *E.g., flat data layout is doubly important in case of frequent iteration*
- Implement your own data structures if (and only if!) necessary

Outlook

Outlook

Where is efficient programming needed?

- Algorithm Engineering
- Computer Graphics
- Operating Systems
- Embedded Systems
- Robotics
- ...

Where to go from here?

- BA
- MA
- PhD
- ...