What is “Optimisation”?
When is it “Premature”?

Version 1.1-beta.1

Dr. Colin Hirsch

What is “Premature Optimisation”? Why is it the “Root of all Evil”?

Version 1.1-beta.1

Dr. Colin Hirsch

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

Donald Knuth

"

premature optimization is the root of all evil

"

Donald Knuth

Optimising what?

- Time vs. space?
- Latency vs. throughput?
- Average vs. worst case?
- Performance vs. efficiency?
- One vs. few vs. many threads?

- Development time?
- Number of unit tests?
- Maintainability of code?

What is “optimisation”?

1st attempt

Making code faster!

Why do we optimise?

- Efficient code is green code
  - saves energy, money and resources
- Fast code makes for better user experience
  - higher user satisfaction, more sales
- Some code has performance requirements
  - embedded and real-time systems, …
- It’s fun and gratifying to make code faster
  - which is why we *might* have a tendency to optimise prematurely

So where’s the “evil”?

“Evil” optimisation side-effects

- Adding code and complexity
- Making code more error prone
- Getting lost in micro optimisations
- Making things slower instead of faster
- Breaking the functionality in corner cases
- Making the code less flexible and malleable
- Wasting time optimising in wrong places
- Making the code harder to read and understand
- Requiring more (unit) tests to verify correctness

What is the “evil”?

- Adding code and complexity

- Managing complexity is *a key challenge* of software development
  - Layered models
  - Divide and conquer
  - Functions and classes
  - Modules and interfaces
  - Other abstraction mechanisms
  - *Less is more* (YAGNI, DRY, minimalistic code)

Big picture: taoCONFIG vs. PEGTL

- taoCONFIG reads configuration files for applications
  - Configuration files are small and read once
    - Optimising taoCONFIG is not our priority
- PEGTL parses data according to a user's grammar
  - Some users parse a lot of data and/or a lot of times
    - PEGTL performance is always on our minds
      - But we are not implementing packrat parsing!
- Takeaway: Look at the big picture before optimising!

Hearsay based optimisation: O(n)

- PEGTL recursive-descent approach is O(n^2) or worse
- “Packrat parsing (with memoisation) is *sooo* much better with its O(n)”
- Yes, packrat parsing has better worst-case complexity, but:
  - A much higher constant factor (overhead), and:
  - has more code and uses more memory at runtime, and:
  - how many real-world grammars hit the worst case anyhow?
- Nobody has convincued us yet that the PEGTL would benefit from packrat
- Takeaway: Theoretical advantages don’t always hold in practice!

Hearsay based optimisation: virtual

- “Virtual functions are slow”
- Yes, they are slower than plain functions, but:
  - Are they slower than the alternatives?
  - How complicated are these alternatives?
- Runtime polymorphism has *some* cost!
- Virtual functions should be used where *appropriate*,
  - and [their overhead] avoided where not necessary
- Takeaway: Compare and choose wisely!

Statistics guided optimisation

- Needed to extend app to keep track of certain things
- Question was which container to use for these things
- Use cases were discussed and access patterns analysed
- Then we discovered the number of these things at any given time
  - Nearly always either 0 and 1
- In other words it doesn’t matter which data structure is used
- Takeaway: Know what you are optimising for!

Library vs. hand rolled: taoJSON

- taoJSON value class holds different types
- Initially based on union & enum & switch statements (fast!)
- Later changed to std::variant (slow?)
- That’s the opposite of optimising, but:
  - Pages of low-level code were removed (great!)
  - Performance did not suffer noticeably (good!)
- Takeaway: The standard library is often very good and/or good enough!

Hand optimised: FLC video player

- Once upon a time I had an FLC video file on my Amiga
- Found a player written in *assembly*
  - Hand-optimised read-and-decode loop (fast but … assembly!)
- But the host adapter can DMA from HDD to RAM…
- Wrote a multi-threaded player in C
  - Use CPU to decode during asynchronous DMA (faster and … easier!)
- Takeaway: Restructuring on high-level beats low-level optimisations!

Just doing our jobs: “Good code”

- Writing *appropriate* / *elegant* / *minimalistic* code
  - std::unique_ptr vs. std::shared_ptr
  - std::vector vs. std::list vs. std::deque vs. std::set
  - passing by value or by reference
- Reasoning about these choices is reasoning about structure and design!
- And these choices convey information to the reader!
- Takeaway: Not everything that *optimises* is an *optimisation*!

Just doing our jobs!

- Everything that simplifies code or reduces complexity
  - Even if it makes the code faster as side effect
- Everything that makes code more readable and maintainable
  - Even if it makes the code faster as side effect
- Making the structure of the code match the structure of the problem
  - Frequently produces *good* or at least *good enough* performance
- Most things that make code faster without increasing complexity
  - Choosing the most efficient alternative without drawbacks

Complexity vs. optimisation?

What is “optimisation”?

2nd attempt

Making code faster…

…while increasing complexity!

What about “premature”?

Premature optimisation checklist

1st attempt

- Am I optimising the right places?
  - Probably not, the profiler is your friend!
  - Is it even worth it, is the code run often enough?
- Will my change improve performance?
  - Benchmark a prototype or mockup or something!
- Am I optimising code that will survive?
  - Is the feature needed in the first place?
  - Will a higher-level improvements eliminate the code?
- Will my unit tests catch bugs introduced while optimising?
- Am I keeping in mind that the most efficient code is … no code?

Recommended approach

- First create the baseline
  - Correct code
  - Nice, simple, minimalistic, elegant, … code
  - This is usually quite fast/efficient,
    - perhaps even fast/efficient enough
- Then think about what might need optimising
  - And use the profiler and the questions to be sure
    - Low hanging fruit with local impact can be fair game

What is “premature optimisation”?

Optimising code…

…before the baseline!

baseline = correct, clean, elegant, minimalistic code

…before doing the checklist!

checklist = all the questions from two slides ago

Thank you!

Colin Hirsch — mail@cohi.at