1 | What is Data-Oriented Design?

The purpose of all programs is to transform data.

- If you don't understand the data you're working with, you don't understand the problem (and conversely, understanding it more means you understand the problem).
- Also, different data = different problems.
- Understanding the problem also relies on understanding the cost of the things you're doing (and therefore hardware as well).
- Optimizing on problems you probably don't have creates problems you do.
- More context, better solution. Data is important for this.
- Reason must prevail - pragmatism over theory.

1.1 | Three Big Lies

1.1.1 | Software is a platform.

Obviously hardware is the platform, and solutions are different for different sets of hardware. x86 vs ARM is different due to varying constraints (and therefore you need to consider low-level things). Reality is not something you're forced to deal with but the actual problem itself (pragmatism > theory).

1.1.2 | Code should be designed around your mental model of the world.

This ends up hiding data - and that's bad since trading maintenance for understanding of the data makes the problem harder to solve. Classes are similar in real life but not in code so you end up with strange monolithic and unrelated data structures. Too abstract of a solution and divorced from reality.

1.1.3 | Code is more important than data.

Code is only for transforming data. Programmers are responsible for solving data transformation problems, not just writing code. And in order to do solve these problems you need to understand the data, and idealized abstract solutions don't work since they ignore the **actual** data you're working with.

1.2 | Results of TBL

- Poor performance
- Poor concurrency
- Poor optimizability
- Poor stability
- Poor testability

2 | Example: Dictionary Lookup

2.1 | Code-First design

Something like this in memory, since we view these as being associated with each other:

Key Value

This isn't actually the case. Most of the time it's only the keys that matter since those are used for the lookup. This kills performance since now you're loading the value into the cache each time only to immediately throw it away.

2.2 | Data-First design

Have them separate, and only the keys will be in cache (yes the value will be a cold lookup that causes a cache miss, but the statistical chance is low). *Always solve for the common case first.*

3 | Cache Times

The most expensive operations are transcendentals at ~100 cycles and square roots at ~25 cycles. In comparison, L1 lookups take 3, L2 take 20+, and RAM 300+ cycles. L1 is great, L2 is decent, and RAM is painfully slow. Because of this, this means L2 cache misses are the most important events when it comes to understanding performance.

The compiler spends its time optimizing everything besides the waiting times for cache lines - because it *can't* solve them.

4 | Improvements

:collapsed: true

- Make sure the whole line is filled with data: make the most out of your reads.
  - Things don't just exist in a void most of the time so make sure to pack them together to achieve this.
- Don't put bools in structs - very low information density so cache is wasted and this likely bumps other more important things beyond the cache
- Compilers aren't amazing at optimizing reads/writes - so make sure to not perform unnecessary ones.
- "Hoist" (bring the the outside) all loop invariants (even simple ones)

Want to see information density of bools? Just dump value and compress it.

5 | Questions

- How do you avoid code duplication without STL/templates?
  - It's not that large of a problem, but ultimately you can just generate the code.
- How do you deal with variety of platforms and the situations your code will be run in?
  - Unlikely that the range if very large and that you don't knnow anything.
- What about non performance-heavy industries?
  - Performance matters everywhere, even to users.
- What about portability?
  - Optimize for a specific range.