D Perf Patterns

S.Rohe Perf Meetup Munich
1.2.2017

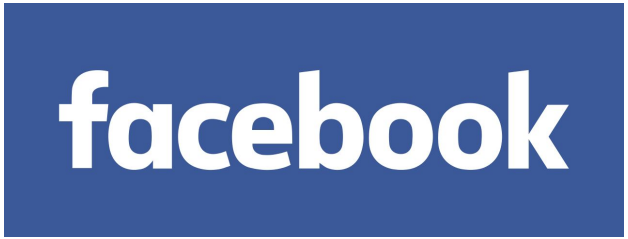
Agenda

- Motivation
- Boundaries
- DLang
- Examples
- Summary

Motivation


sociomantic
from dunnhumby


ebay


facebook

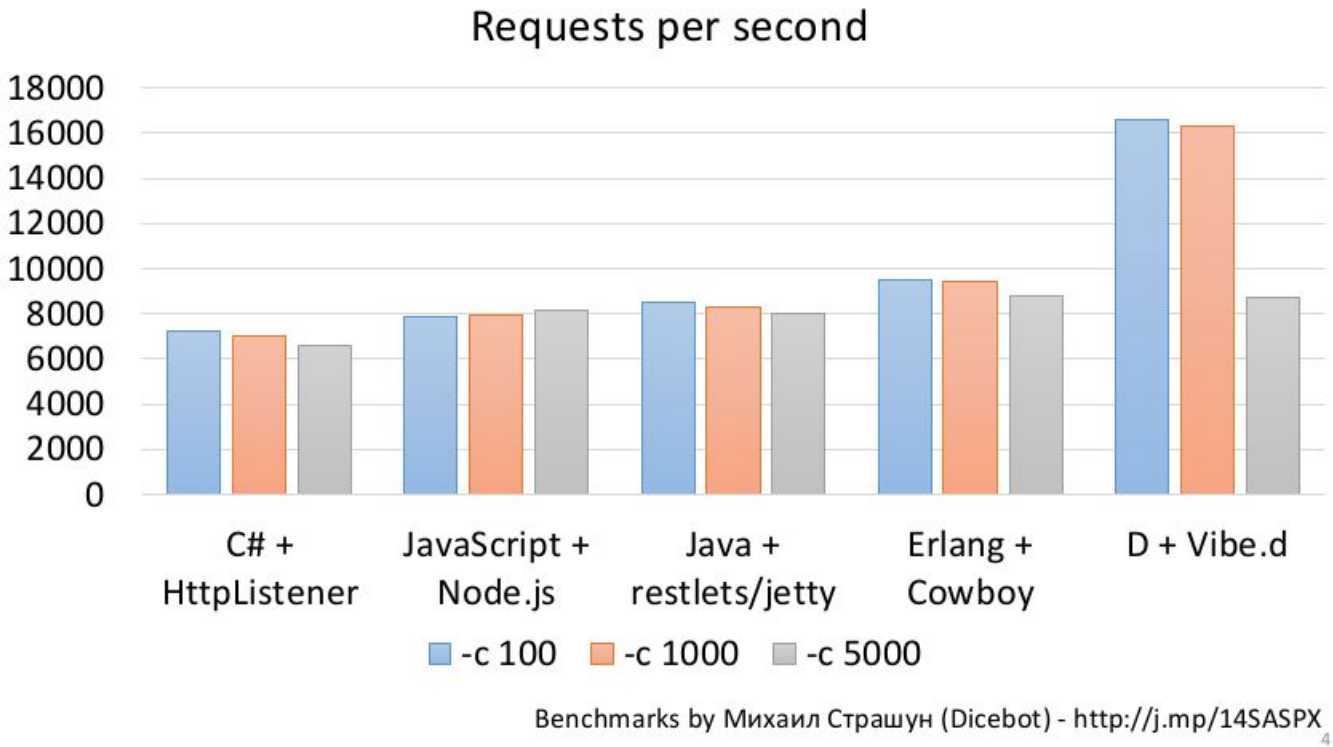

REMEDY


WEKA.IO


AdRoll

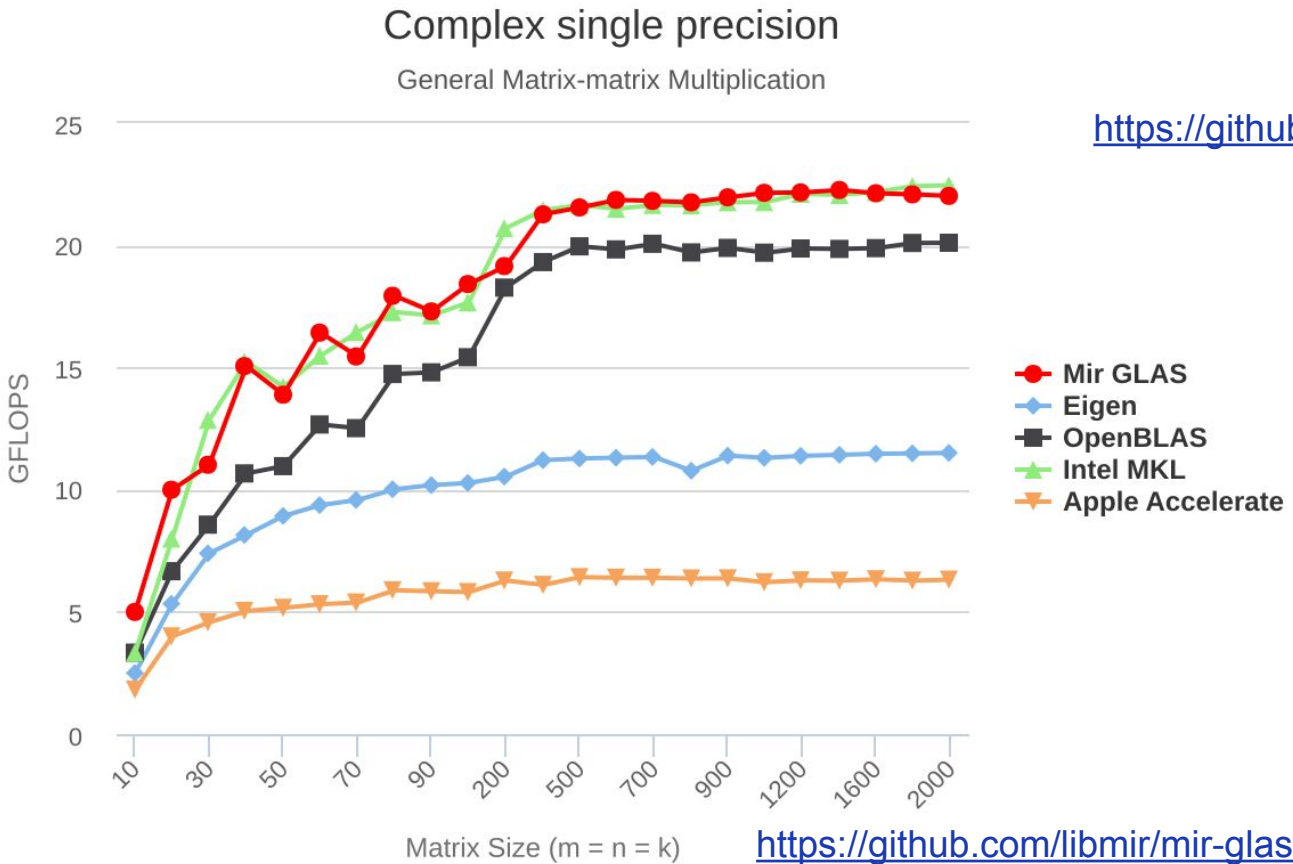

funkwerk
aktiengesellschaft

Motivation (vibe.d - D WebServer)



https://github.com/rejectedsoftware/vibe.d

Motivation (libmir - Matrix Operations)



https://github.com/libmir/mir-cpuid

https://github.com/libmir/mir-glas

Boundaries

just single core

Performance difficult to define (run time performance, memory consumption, parallel performance, ...)

not particular D specific

no proper measurements for following examples

DLang

D is a systems programming language with C-like syntax and static typing. It combines efficiency, control and modeling power with safety and programmer productivity. [dlang.org]

since 2001 D1; since 2011 D2

designed by C++ Compiler Guru and C++ Template Guru

fast compilation time, static if / static foreach

http://tour.dlang.org/

How to use Prime Numbers?

program needs prime numbers

- compute them so fast, that caching makes no sense
- cache them on first calculation
- read in a precalculated prime.txt file
- compute them during compilation

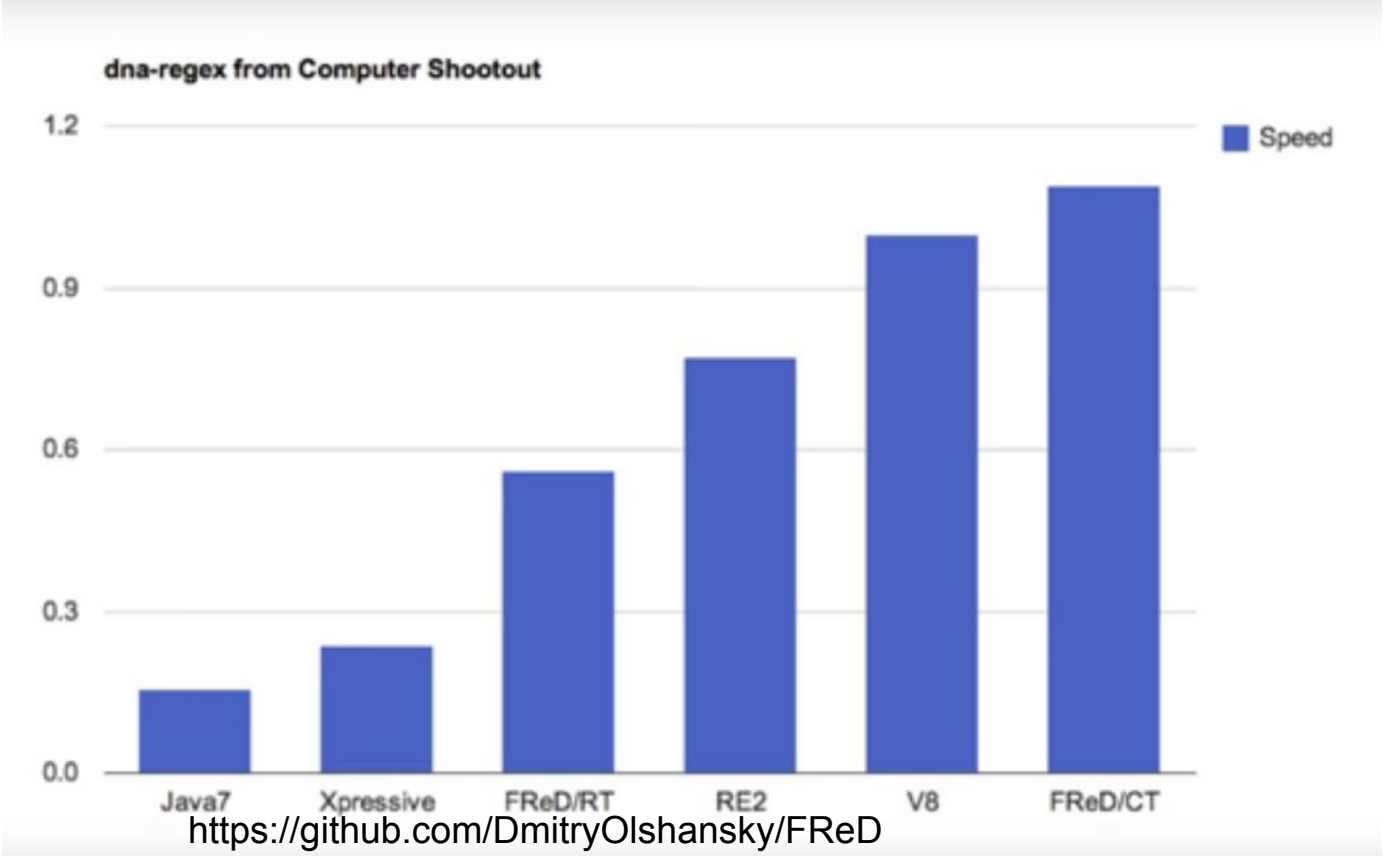
DLang - CT-Primes

Example: primes.d

```
#!/usr/bin/env rdmd
import std.algorithm, std.range, std.stdio;

uint[] primes(uint max) {
  return iota(1, max).filter!isPrime.array;
}

void main() {
  enum a = primes(10000);
  stdout.writefln("compile time primes %s", a);
}
```

```
// Algorithm from
https://en.wikipedia.org/wiki/Primality_test
bool isPrime(uint n) {
  if (n <= 3) return n > 1;
  if (n % 2 == 0 || n % 3 == 0) return false;
  uint i = 5;
  while (i*i <= n) {
        if (n % i == 0 || n % (i + 2) == 0) return false;
    i += 6;
  }
  return true;
}

unittest {
 assert(2.isPrime);
 assert(11.isPrime);
 assert(!12.isPrime);
 assert(97.isPrime);
}
```

Motivation (regex)



https://github.com/DmitryOlshansky/FReD

Examples - Regular Expressions

Example: regex.d

```
#!/usr/bin/env rdmd
import std.regex, std.stdio;

void main() {
	auto r = ctRegex!`\d\d.\d\d.\d\d\d\d`;
	auto r2 = regex(`\d\d.\d\d.\d\d\d\d`);
	string long_text = "this is a long string with some date like 01.02.2003 in it.";

	stdout.writefln("compile time match?: %s", long_text.match(r));
	stdout.writefln("run time match?: %s", long_text.match(r2));
}
```

Pattern 1: Compile Time Evaluation

- regular expressions
- ⅓, ½, sqrt(2), simple calculations
- cpuid
  - array length based on cache sizes
- static for - loop unrolling
- schema validation DSL
- define own grammars / DSLs: https://github.com/PhilippeSigaud/Pegged
- ...
- “calculate” algorithms at compile time

Sort-n

- special sort for a fixed set of members
- sort requires partitioning datasets into smaller partitions
- for large partitions this makes sense
- for smaller partitions:
  - splitting is not efficient anymore
  - do the n comparisons and swaps by hand is faster

Example: sortn.d

Sort-3

```
static if (n == 3)
    {
        s.conditionalSwap!(less, Range,
                           0,1,
                           1,2,
                           0,1);
    }
```

Sort-4

```
static if (n == 4)
    {
        s.conditionalSwap!(less, Range,
                           0,1, 2,3, // 2 in parallel
                           0,2, 1,3, // 2 in parallel
                           1,2);
    }
```

Memory is slow

Size and Speeds

- L1: 3-4 cycles, 32 kb
  - instruction
  - data
- L2: 8-14 cycles, 256kb
- L3: tens of cycles, few MB, often shared
- Cache line: 64 bytes

http://de.slideshare.net/AndreiAlexandrescu2/dconf-2016-bitpacking-like-a-madman-by-amaury-sechet

Struct Padding

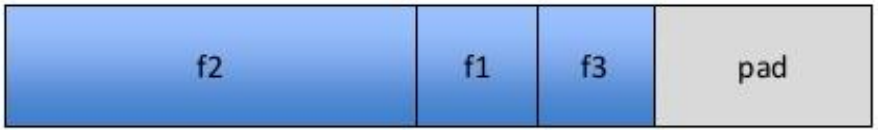
```
struct S {
    bool f1;
    uint f2;
    bool f3;
}
```

12 bytes, 6 wasted

http://de.slideshare.net/AndreiAlexandrescu2/dconf-2016-bitpacking-like-a-madman-by-amaury-sechet

Struct Padding

```
struct S {
    uint f2;
    bool f1;
    bool f3;
}
```



8 bytes, 2 wasted

http://de.slideshare.net/AndreiAlexandrescu2/dconf-2016-bitpacking-like-a-madman-by-amaury-sechet

Struct Packing

Example: packing.d

```
import std.bitmanip;

struct S {
   mixin(bitfield!(
      uint, "f1", 30,
      bool, "f2", 1,
      bool, "f3", 1,
   ));
}
```

4 bytes, 0 wasted

- f1 is now 30 bits instead of 32 bits
  - Now about 1B max
- Fields aren't atomic anymore
- bitfield does all the magic

http://de.slideshare.net/AndreiAlexandrescu2/dconf-2016-bitpacking-like-a-madman-by-amaury-sechet

Class/Struct Inheritance

- Classes are on the heap, require pointer
- Structs are on stack, no pointer
- Structs cannot be inherited, normally

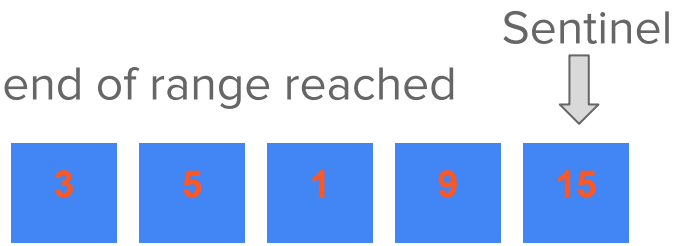
Example: struct.d

Pattern 2: Reduce Data

- less data, more efficient usage of the cache line
- less abstractions, less data

Partition I

- Idea of a Sentinel for find
- with sentinel does not need to check if end of range reached
- find with sentinel requires:
  - random access range
  - moving elements



https://www.youtube.com/watch?v=AxnotgLql0k

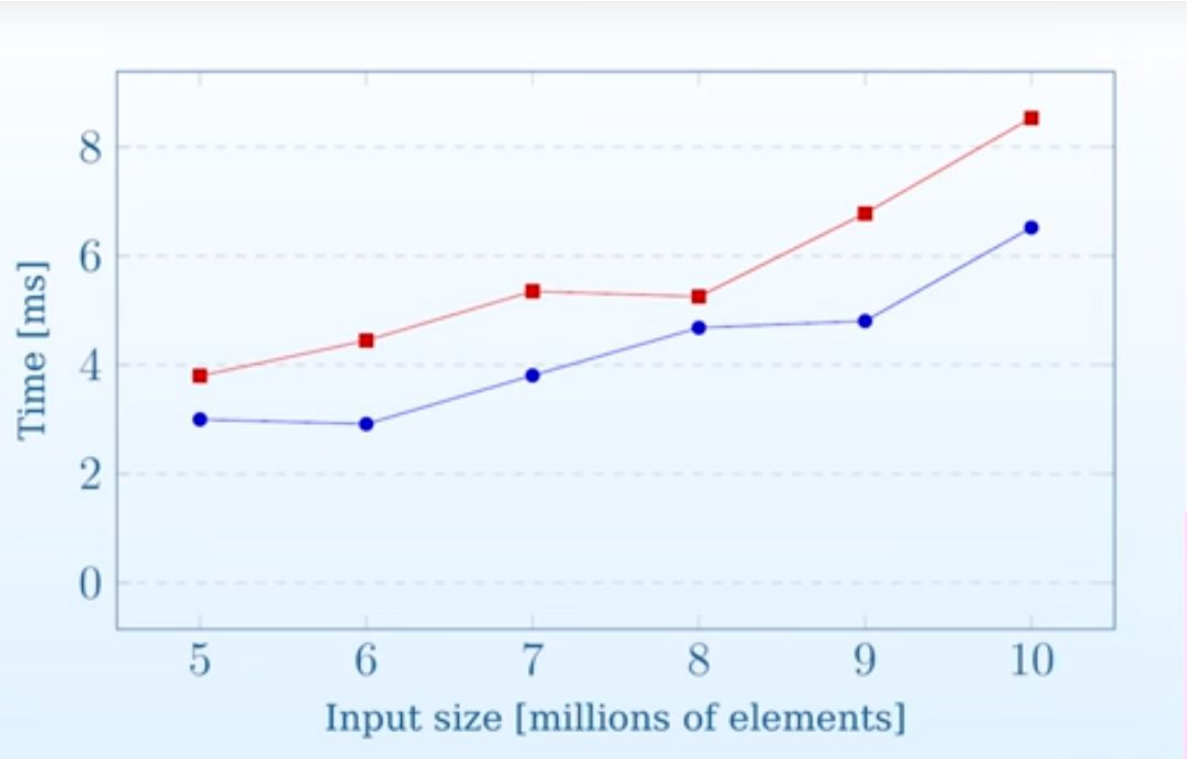
Partition II - Design By Introspection

static if to the rescue

```
Range find(R, E)(R r, E e)
if (isInputRange!R && is(typeof(r.front == e) : bool))
  static if (isRandomAccessRange!R && hasSlicing!R) {
    static if (is(typeof(
      () nothrow { r[0] = r[0]; }
    ))) {
      ... sentinel implementation ...
    } else {
      ... indexed implementation ...
    }
  } else {
    ... conservative implementation ...
  }
}
```

https://www.youtube.com/watch?v=AxnotgLql0k

Partition III - Find with Sentinel



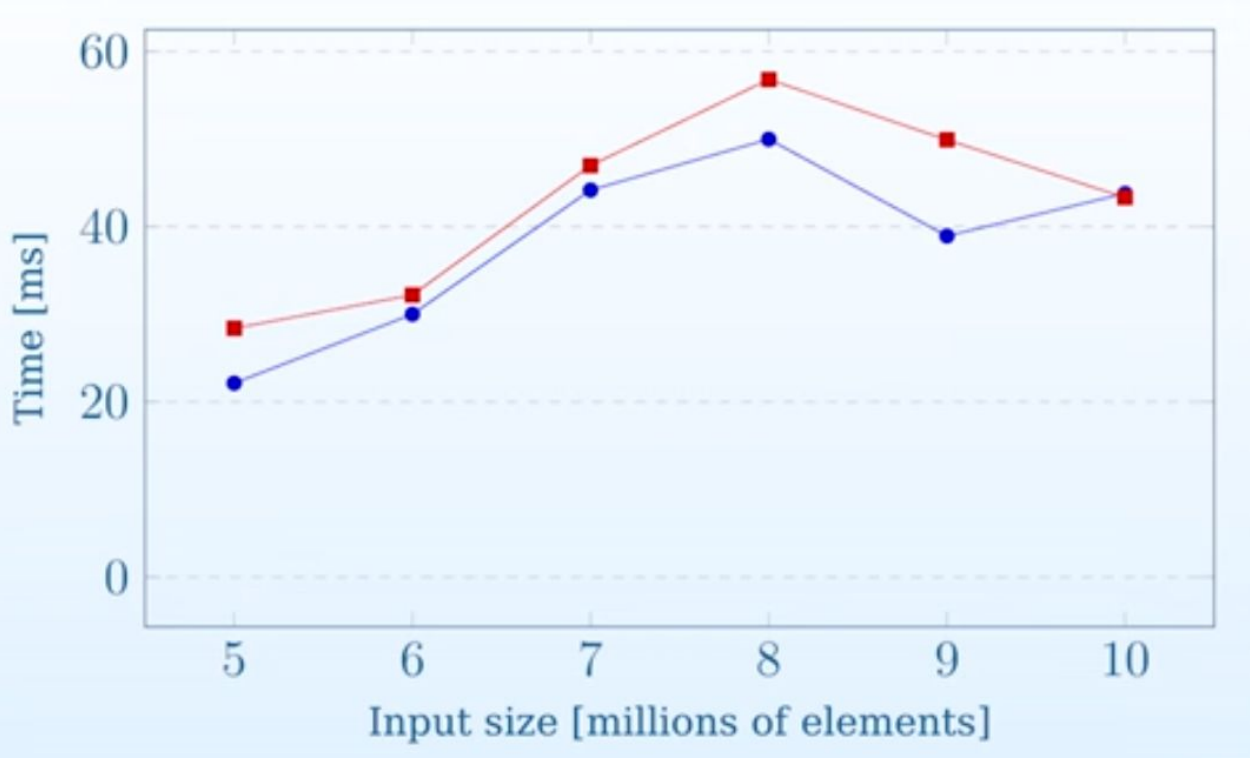
https://www.youtube.com/watch?v=AxnotgLql0k

Partition IV

- partition could be also done via sentinel
- (quick)sort performance relies heavily on partition
- Algorithm:
  - Plant sentinels at both ends
  - Create a “vacency” in the range
  - Break swapAt into two assignments
  - An assignment fills a vacency and leaves another one

https://www.youtube.com/watch?v=AxnotgLql0k

Partition V - with sentinel (blue)



https://www.youtube.com/watch?v=AxnotgLql0k

Pattern 3: Design By Introspection

- every function could provide multiple implementations
  - a conservative one
  - an optimized one
- implementation choosen at compile time based on the properties of a parameter types

Performance Patterns

- Doing as little work as possible
  - lazy log.warn
- Doing as much as possible at Compile Time
  - sqrt(2) at compile time, regex at compile time, design by introspection
- Optimize to the current CPU; cache line sizes, instructions, ...
  - preload the amount of data that fits your cache sizes
  - Vectorization - unfolding for loops based on cache sizes
- Reduce data dependency
  - sortn
- Reduce size of data
  - int8 instead of int64, bitpacking
- Reduce abstractions
  - struct inheritance


For D Lovers
Next Meetup
DLang and the Cloud - 14.02.2017 @ celonis
• DLang and the Cloud
• D-Compiler usage within the Cloud
• Heroku with D

References

Amaury Sechet - Bit Packing like a Madman

https://www.youtube.com/watch?v=95O_y9fu6qk

Alexandrescu - Writing Fast Code - how to optimize atoi

Part 1: https://www.youtube.com/watch?v=vrfYLlR8X8k

Part 2: https://www.youtube.com/watch?v=9tvbz8CSI8M

Alexandrescu - Fastware - ACCU 2016 Keynote

https://www.youtube.com/watch?v=AxnotgLql0k