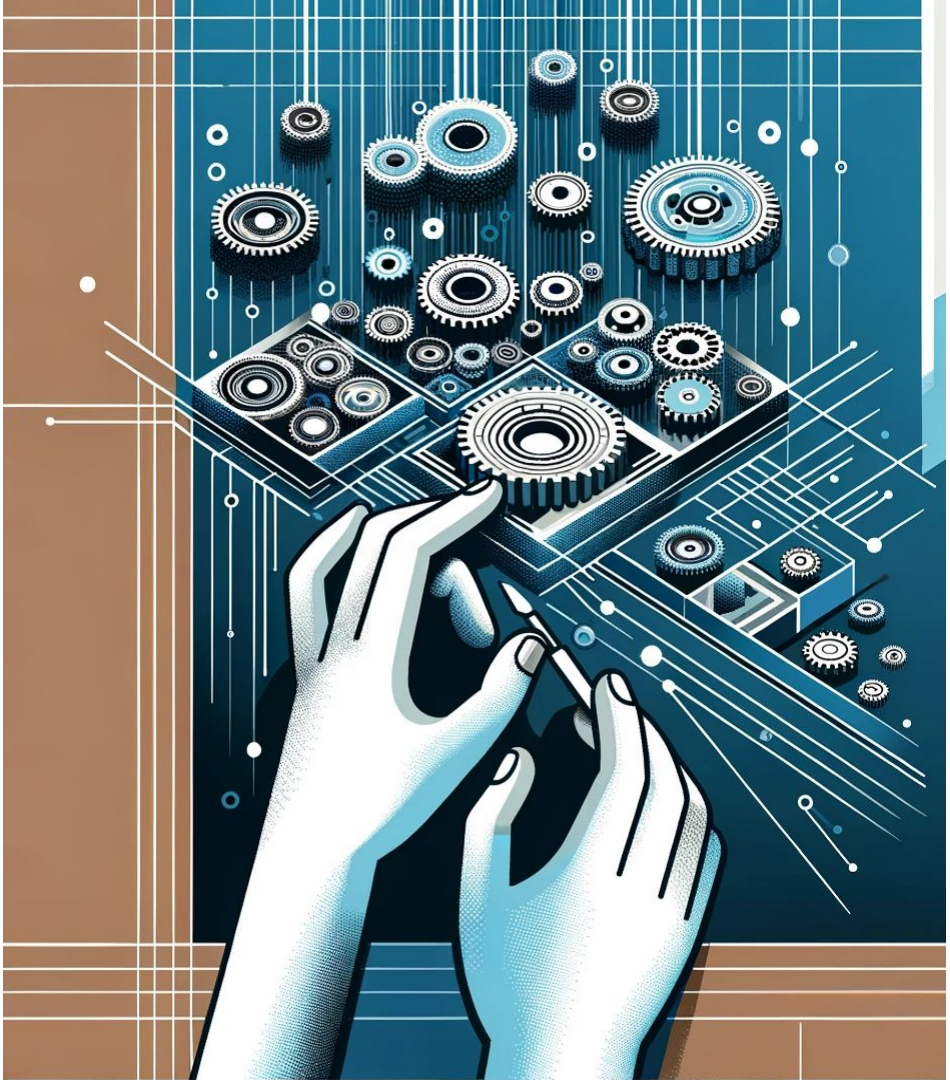


Mikrooptimalizace

Malé změny, velký dopad?

28.2.2024 v 18:00

Restaurace U Salzmannů

Salonek v 1. patře

Vstup a občerstvení zdarma

PeoplePath

We are a **global provider** of cloud-based solutions for **talent relationship management.**

Since 2002, we have empowered our clients to establish, track and develop **personal relationships** with talent throughout their entire **career lifecycle**.

We offer the most powerful and flexible **candidate engagement** technology in the marketplace.

Seattle
USA

New York
USA

Munich
Germany

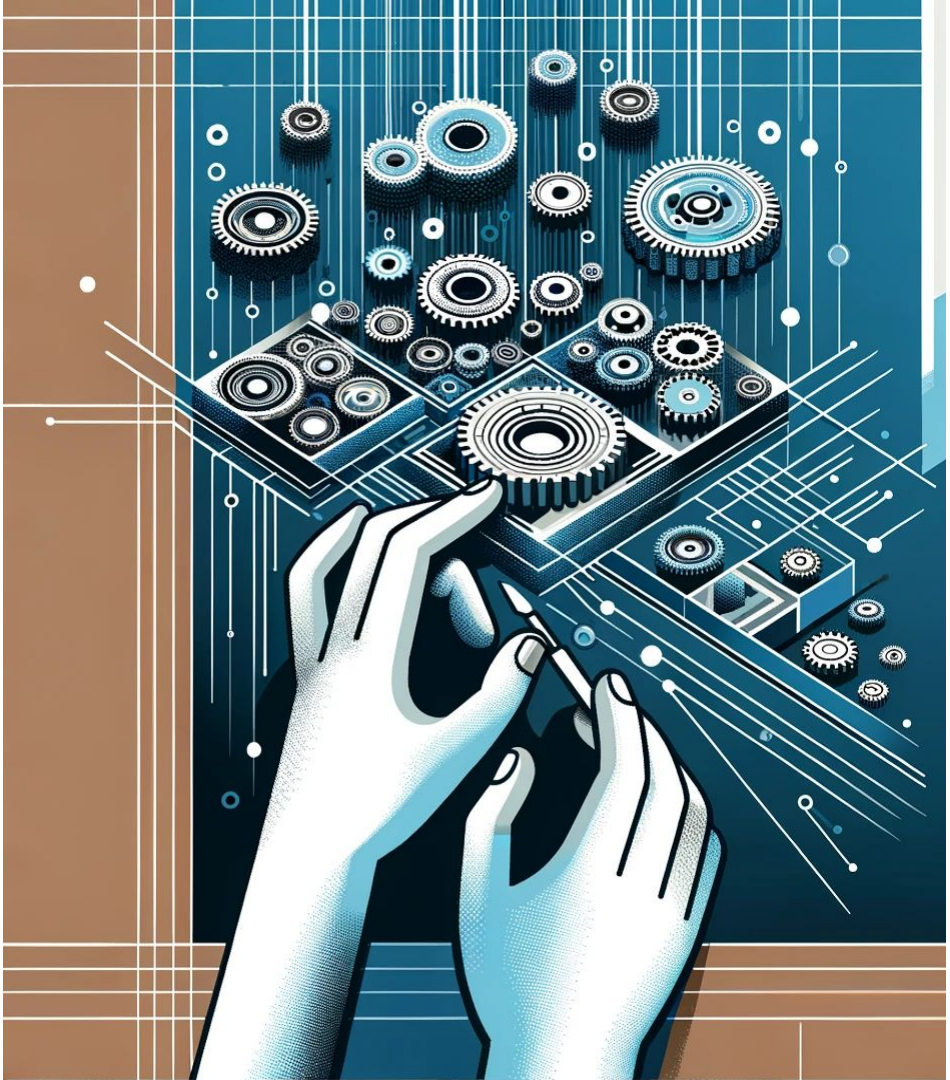
Pilsen
Czech Republic

60+
employees

100+
clients in all major sectors

250+
years of combined talent engagement experience

1,500,000+
talent profiles across clients globally



Mikrooptimalizace

Malé změny, velký dopad?

28.2.2024 v 18:00

Restaurace U Salzmannů

Salonek v 1. patře

Vstup a občerstvení zdarma

PeoplePath

Few facts about me...

- Software architect.
- +20 years experience in IT.
- Conference speaker.
- Contributor to PHP magazines.

Today’s agenda

- Micro-optimization?
- Measure, Measure, Measure!
- Micro-optimization in PHP?
- Need for speed - don’t do this at home!

Micro-optimization

Fine-tuning of small specific parts of a system.

Focus: functions, algorithms, structures.

Goal: minor/localized improvements.

Macro-optimization

Broad improvements across system components or the entire system to boost overall performance.

Focus: Modules, components, or system-wide.

Goal: Enhance overall system efficiency.

Architectural change

Fundamental modifications to address systemic issues or incorporate new technologies.

Focus: The entire system or major components.

Goal: Massively improve system performance.

Strategic focus

- Micro-optimization - critical fine-tuning.
- Macro-optimization - broad enhancements.
- Architectural changes - systemic issues.

...back to micro-optimization

What does this piece of code do?

```
for (int i = 0; i < count; i++) {
    to[i] = from[i];
}
```

Array copy

```
void copy_array(char *to, char *from, int count) {
    for (int i = 0; i < count; i++) {
        to[i] = from[i];
    }
}
```

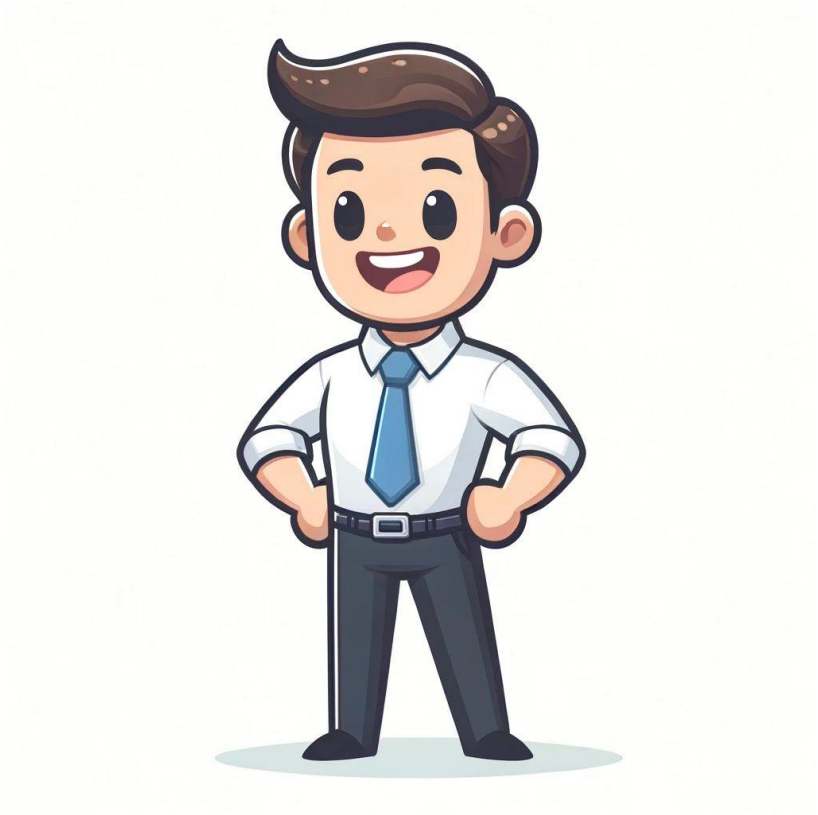
Back to the '80s

Whats happens in 80...

...stays in 80's.

That's a rule for next few minutes and slides!

Meet Duff



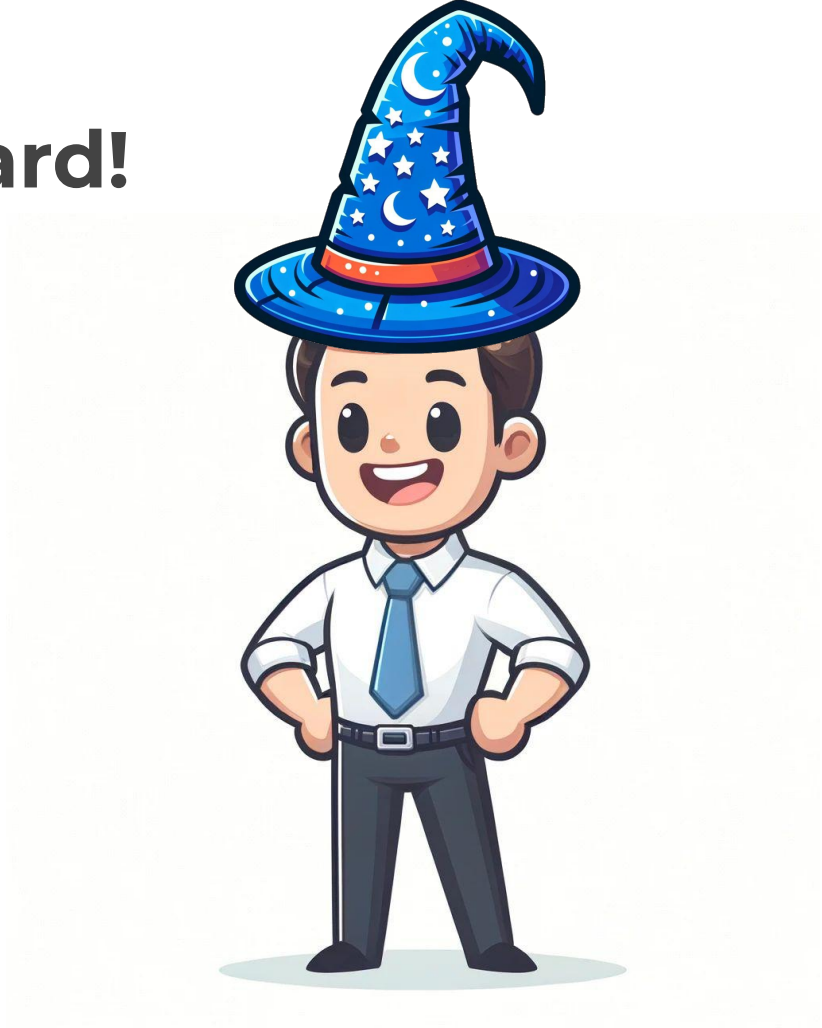
Duff has a problem...

- Working for Lucasfilm.
- Computer R&D Division (future Pixar).
- Computers are slow.

Array copy

```
void copy_array(char *to, char *from, int count) {
    for (int i = 0; i < count; i++) {
        to[i] = from[i];
    }
}
```

Duff is wizard!



Duff's device

```
int i = 0;
int n = (count + 7) / 8;

switch (count % 8) {
    do {
        case 0: to[i] = from[i]; i++;
        case 7: to[i] = from[i]; i++;
        case 6: to[i] = from[i]; i++;
        case 5: to[i] = from[i]; i++;
        case 4: to[i] = from[i]; i++;
        case 3: to[i] = from[i]; i++;
        case 2: to[i] = from[i]; i++;
        case 1: to[i] = from[i]; i++;
    } while (--n > 0);
}
```

Duff’s device

Yes, this is valid C or C++ code!

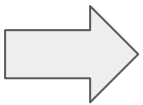
```
int i = 0;
int n = (count + 7) / 8;

switch (count % 8) {
    do {
        case 0: to[i] = from[i]; i++;
        case 7: to[i] = from[i]; i++;
        case 6: to[i] = from[i]; i++;
        case 5: to[i] = from[i]; i++;
        case 4: to[i] = from[i]; i++;
        case 3: to[i] = from[i]; i++;
        case 2: to[i] = from[i]; i++;
        case 1: to[i] = from[i]; i++;
    } while (--n > 0);
}
```

Yes, this is valid C or C++ code!

Loop unrolling

- Minimize iterations, by code duplication.
- In some situations* code on right is faster:

```
for (int i = 0; i < count; i++) {
   to[i] = from[i];
}
```



```
for (int i = 0; i < count; i += 8) {
   to[i] = from[i];
   to[i+1] = from[i+1];
   to[i+2] = from[i+2];
   to[i+3] = from[i+3];
   to[i+4] = from[i+4];
   to[i+5] = from[i+5];
   to[i+6] = from[i+6];
   to[i+7] = from[i+7];
}
```

Why switch/do combination?

```
switch (count % 8) {
    do {
        case 0: to[i] = from[i]; i++;
        case 7: to[i] = from[i]; i++;
        case 6: to[i] = from[i]; i++;
        case 5: to[i] = from[i]; i++;
        case 4: to[i] = from[i]; i++;
        case 3: to[i] = from[i]; i++;
        case 2: to[i] = from[i]; i++;
        case 1: to[i] = from[i]; i++;
    } while (--n > 0);
}
```

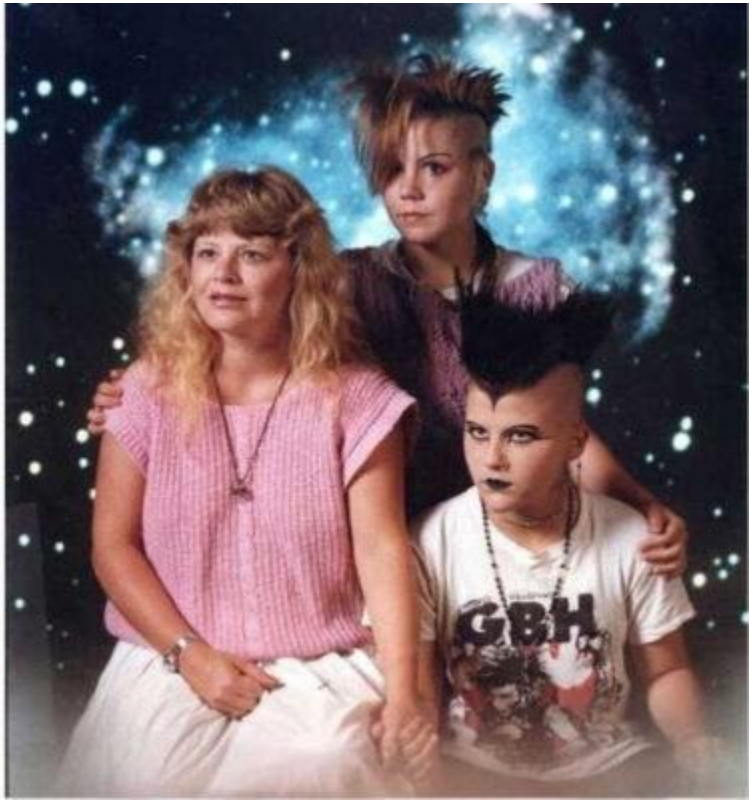
Duff initial position

He knows how:

- CPU works internally,
- compilers works internally.

He really needs to speed up something.


BACK
TO THE FUTURE


GBH

We no longer live in 80’s

- CPU:
  - much faster,
  - doing better optimalistions.
- Same for compilers.
- Duff device is no longer fastest solution.
- On contrary - can be **slower**.

Duff device and microptimalisation

- You often need knowledge of inner workings.
- They can be time/implementation specific.
- They can look bizar.

Measure, measure...

Speed measure troubles...

Benchmark how long it will take to remove column from middle of big table in MySQL.

What has biggest impact on time?

A. Number of records in table?

B. MySQL version?

C. Call from my boss on MS Teams?

What has biggest impact on time?

C. Call from my boss on MS Teams...

Speed measuring challenges

- Challenges of a very small piece of code:
  - accuracy and environment issues,
  - measurement and hardware variability
  - data set and optimizations.

Establish a proper methodology?

...reporting standards in computer science have some way to improve before they match the quality of the very best practice in the natural or social sciences.*

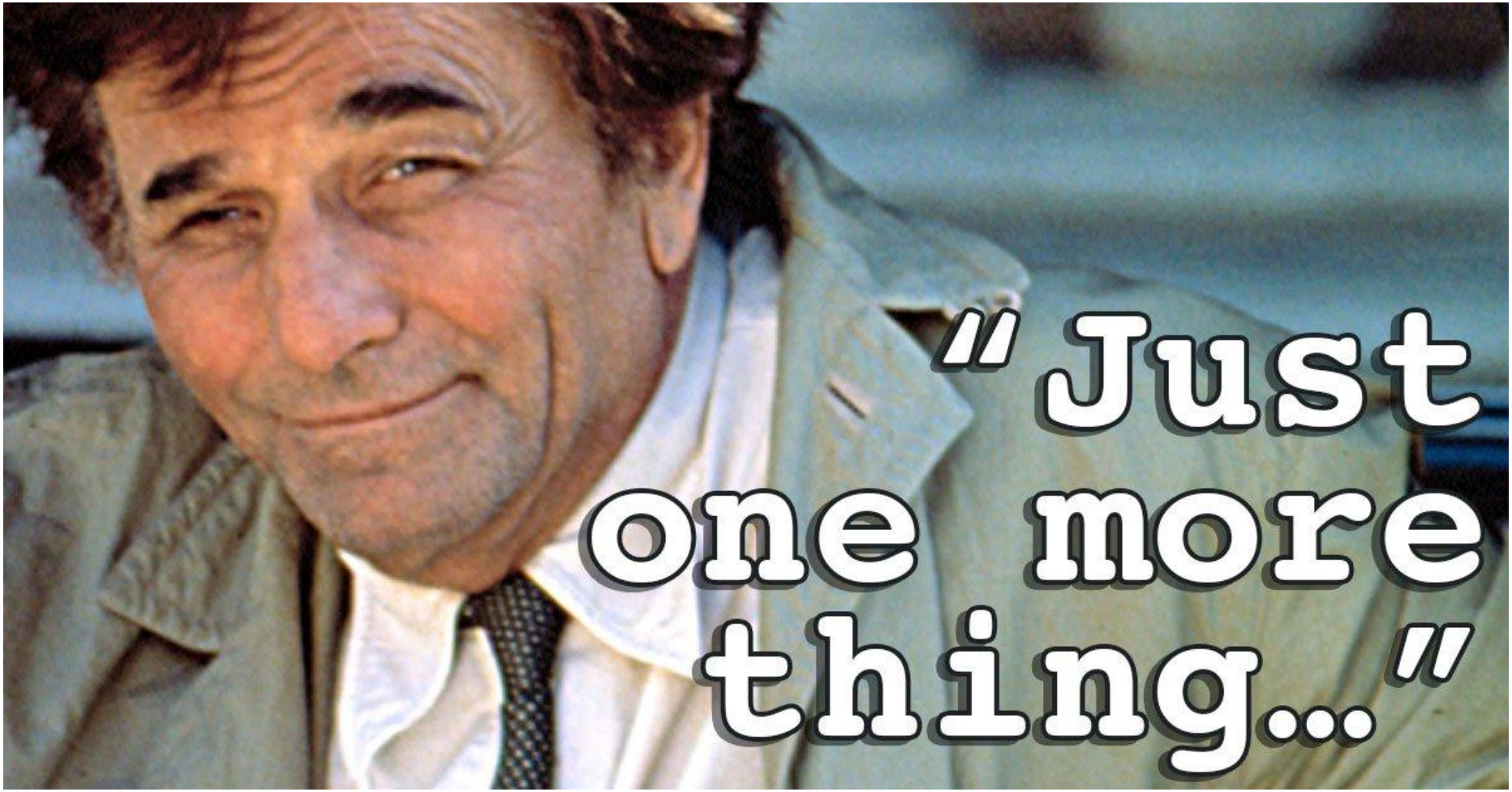
*Mytkowicz et al [2008] find measurement bias to be “significant and commonplace”

Establish a proper methodology?

- Paper: Robust Benchmarking in Noisy Environments

Not always suitable...

- Can be hard in day to day basis.
- Try at least to:
  - not reinvent the wheel (phpbench),
  - isolate your benchmark from system load,
  - measure in proper environment.


"Just one more thing..."

Think about role of optimizations...

- How this will be stored in opcache?

```php
<?php

echo strlen('hello world');
```

Think about role of optimizations...

```php
<?php

echo strlen('hello world'); // prints length of 'hello world': 11
```

“Compiled code”:

```
line      #* E  I  O op                    ext   return  operands
-------------------------------------------------------------------
    3     0  E >   ECHO                                   11
          1        > RETURN                               1
```

Think about role of optimizations...

```php
<?php

echo 11; // prints length of 'hello world': 11
```

“Compiled code”:

```
line      #* E I O op                    ext   return  operands
-------------------------------------------------------------------
    3      0  E >   ECHO                                    11
           1       > RETURN                                  1
```

Think about role of optimizations...

- Where is call to strlen?
- Where is string “hello world”?

```
line       #* E I O op                    ext   return  operands
-------------------------------------------------------------------
    3      0  E >   ECHO                                    11
           1      > RETURN                                  1
```

TL;DR good for performance, bad for microbenchmarking...

Micro-optimization in PHP

Micro-optimization in PHP

- No low level access, missing “opportunities”.
- Despite JIT still mainly using interpretation.
- No compiler with extensive optimisation.

X

- Opcache do more then caching.

Hoist loop conditions

- Limit evaluation of expansive condition.

```
for ($x = 0; $x < get_the_limit(); $x++) { ... }
```

```
$limit = get_the_limit();
for ($x = 0; $x < $limit; $x++) { ... }
```

Short-circuit evaluation

- Evaluation of expressions stops as soon as the outcome is determined.

```
if ($enabled && expansiveCondition()) { ... }
```

Early return

```php
function findValueAndExitEarly(
    $array,
    $value
) {
    foreach ($array as $item) {
        if ($item === $value) {
            return true;
        }
    }

    return false;
}
```

```php
function findValueCompleteIteration(
    $array,
    $value
) {
    $found = false;
    foreach ($array as $item) {
        if ($item === $value) {
            $found = true;
        }
    }

    return $found;
}
```

Don’t be dynamic!

- A “variable variable”
- ~ 2 times slower

```php
$my_var  = 'hello';
$$my_var = 'world';

echo $$my_var; // world
```

Don’t be dynamic!

- A variable class method.

```
$my_func = '\\my\\stuff::hello';
$my_func();
```

Eval is Evil (and slow)

- Eval, insecure, inelegant and slow.
- No opcache.
- ~ 98.6 times slower

```
eval(
    "echo 'I am insecure and ignored by opcache';"
);
```

Don’t be dynamic, even with classes!

```php
class DynamicAccess {
   private $attributes = [];
   public function __call($name, $arguments) {
       $prefix = substr($name, 0, 3);
       $property = strtolower(substr($name, 3));

       if ($prefix == 'set' && count($arguments) == 1) {
           $this->attributes[$property] = $arguments[0];
       } elseif ($prefix == 'get' && count($arguments) == 0) {
           return array_key_exists($property, $this->attributes)
              ? $this->attributes[$property] : null;
       } else {
           throw new Exception("Method {$name} is not supported.");
       }
   }
}

$obj->setName('User name');
$obj->getName();
```

Don’t be dynamic, even with classes!

```
class RegularAccess {
    private $name;

    public function setName($name) {
        $this->name = $name;
    }

    public function getName() {
        return $this->name;
    }
}

$obj->setName('User name');
$obj->getName();
```

Don’t be dynamic, even with classes!

- Dynamic methods are always slower.
- ~ 3.1 times slower
- Before PHP 7.0 it has a real impact.

Don’t be dynamic

- Dynamic stuff in PHP is always slower.
- More importantly:
  - harder to read and
  - can be a problem for static analysis.

Quotes vs apostrophes

- No difference for this:

```
$string_double = "foo";
$string_single = 'bar';
```

Quotes vs apostrophes

Quotes vs apostrophes

```
compiled vars:  !0 = $string_double, !1 = $string_single
line      #* E I O op                    ext  return  operands
-------------------------------------------------------------
    3     0  E >   ASSIGN                              !0, 'foo'
    4     1        ASSIGN                              !1, 'bar'
          2      > RETURN                               1
```

Variable interpolation vs concatenation

- There is no performance difference.

```
$foo = $bar . $none . 'hello';

$foo = "{$bar}{$none}hello";
```

Multi-Byte strings

- mb_* functions (mb_strlen, etc.)
- ~ 466.3 times slower
- Hard to ignore in globalised world.

Multi-Byte - tl;dr

- Not only in globalised world:

```php
$string = "Don't do it 🚫, it won't work 🤦";

var_dump(strlen($string));    // int(36)
var_dump(mb_strlen($string)); // int(30)
```

Compare, comparison === vs ==

- == is marginally slower (~1.06 times)
- === preventing unexpected results

```php
if ($variable === false)
```

```php
if ($variable == false)
```

declare(strict_types)

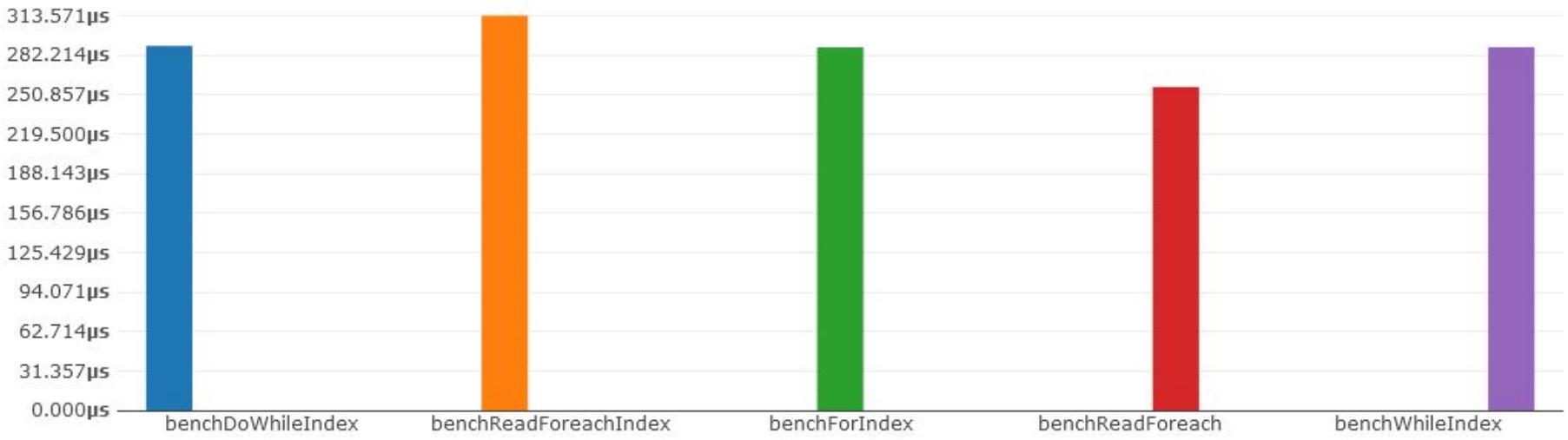
- Enforces strict type checking for function.
- No effect on performance.
- Again take into other pros/cons.

```php
<?php declare(strict_types=1);
```

Loop traversion

- Comparison between:
  - foreach (with index)
  - foreach (without index)
  - for
  - do while
  - while

Read


313.571µs
282.214µs
250.857µs
219.500µs
188.143µs
156.786µs
125.429µs
94.071µs
62.714µs
31.357µs
0.000µs
benchDoWhileIndex
benchReadForeachIndex
benchForIndex
benchReadForeach
benchWhileIndex

Loop traversion - read

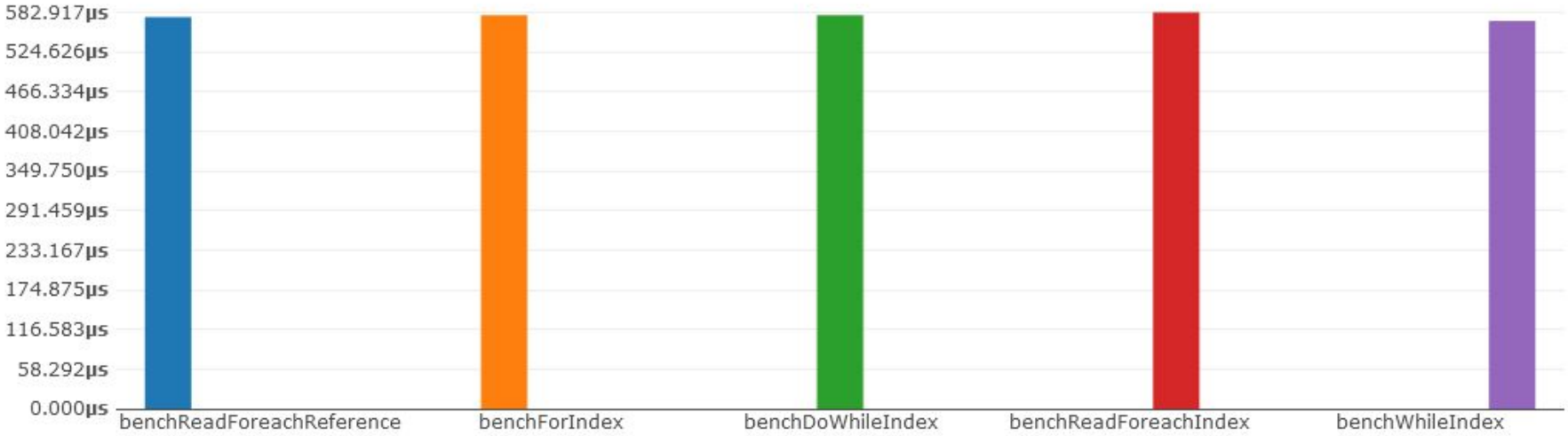
- foreach() is the most efficient.
- But only without index.

```
foreach ($array as $item) {
```

- Index version is slowest (~1.22 times slower).

```
foreach ($array as $i => $item) {
```

Loop traversion - write



Loop traversion

- Similar performance.
- Foreach with index is still slowest.

```
foreach ($array as $i => $item) {
    $array[$i] = $item . 'boo';
}
```

References

- TL;DR don’t use it!
- Unless you writing into big arrays.
- Before PHP 7.0 - terrible performance.
- Regular arrays in PHP implement COW.

COW - Copy on write

Anonymous functions

- Closures/anonymous function are slower.
- ~1.2 times slower.

```php
$parseTemplate = function($template, $data) {
    foreach ($data as $key => $value) {
        $placeholder = "{{" . $key . "}}";
        $template = str_replace($placeholder, $value, $template);
    }
    return $template;
};

$result = $parseTemplate($template, $data);
```

array_map and friends

- Call **strtoupper** on big array:
  - foreach
  - array_map + closure
  - array_map + function name

foreach

```php
foreach ($words as $word) {
    $upperWords[] = strtoupper($word);
}
```

array_map + closure

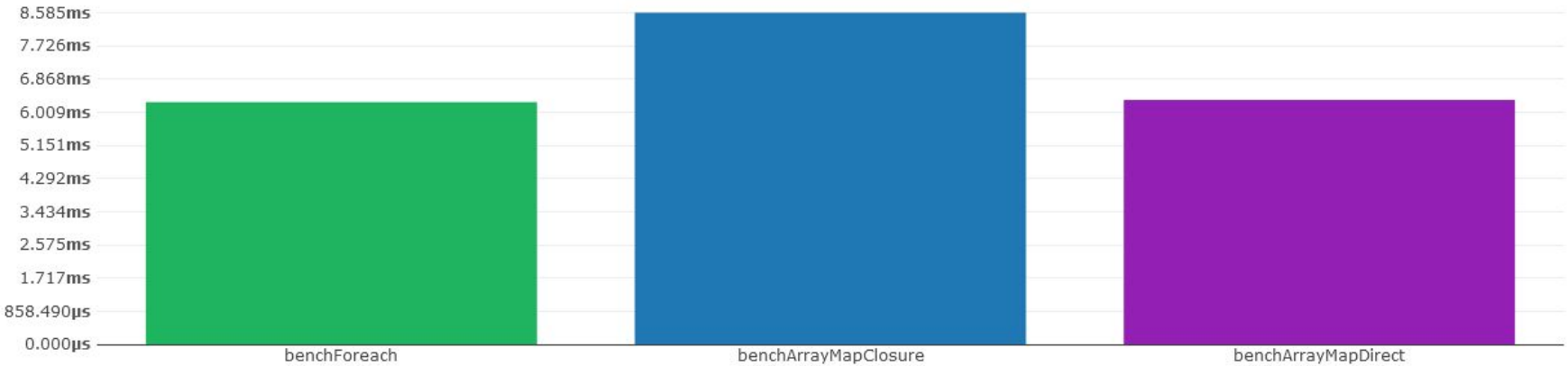
```
$upperWords = array_map(
    function($word) {
        return strtoupper($word);
    },
    $words
);
```

array_map + function name

```php
array_map('strtoupper', $words);
```

array_map and friends

- array_map + closure ~1.36 times slower



Operators beats functions?

- Closer to engine, faster code?

```php
filter_var(
    $foo,
    FILTER_SANITIZE_NUMBER_INT
);

intval($foo);

(int) $foo;
```

Operators beats functions?

- **filter_var** is ~3.65 times slower
- No difference between intval and (int).
- PHP substitutes some function calls.

No difference...

```php
$result = is_null($value);

$result = $value === null;
```

```
line      #* E I O op                                    ext  return  operands
-------------------------------------------------------------------------------
    3     0  E >   TYPE_CHECK                             2  ~2      !1
          1          ASSIGN                                         !0, ~2
    5     2          TYPE_CHECK                          2  ~4      !1
          3          ASSIGN                                         !0, ~4
    6     4        > RETURN                                         1
```

Function name resolving

- Every call in namespaced context to function check local scope and then in the main scope.

```
strlen('this is my string !'.$i);
\strlen('this is my string !'.$i);
```

Function name resolving

- Every call in namespaced context to function check local scope and then in the main scope.

```
namespace FooBar {
    strlen('this is my string !'.$i);
    \strlen('this is my string !'.$i);
}
```

Function name resolving

```php
namespace FooBar {
    function strlen() {return "suprise!";}
    echo strlen('this is my string !');
    echo \strlen('this is my string !');
}
```

```
suprise!19
```

Function name resolving

- Looking for overridden function is ~1.29 times slower.
- Potential perf gain by ~22% just by:
  - prefix calls by \
  - or by “use strlen;”

Need for speed!

Need for speed!

- I really need to speed up my code!
- I am (mostly) limited to PHP!

When things get out of control!

- I really want to have a fast calculation of:

Cosine similarity

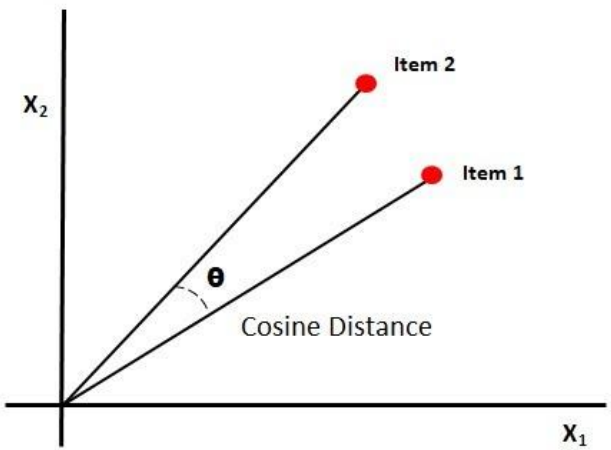
$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Cosine what??

- Measures similarity between two entities based on their features.
- Useful in recommendation systems to find similar content or preferences.

Visual analogy

- Think of entities as arrows from the same point; similarity is the cosine of the angle between them.



What’s that good for?

- LLM has short memory and limited input.
  - Complicates analysis of large texts.
- LLM embeddings
  - Compute semantic vectors for text.

LLM embeddings

- Compute the embedding for each text section.
- Compute the embedding for the user prompt.
- Compare embeddings using **cosine similarity**.
- Send the question to GPT with the associated section.

Back to cosine similarity

```php
function cosine_similarity($vec1, $vec2) {
    $dot_product = 0.0;
    $magnitude_a = 0.0;
    $magnitude_b = 0.0;

    $size = count($vec1);
    for ($i = 0; $i < $size; ++$i) {
        $dot_product += $vec1[$i] * $vec2[$i];
        $magnitude_a += $vec1[$i] * $vec1[$i];
        $magnitude_b += $vec2[$i] * $vec2[$i];
    }

    return $dot_product / (sqrt($magnitude_a) * sqrt($magnitude_b));
}
```

Back to cosine similarity

```php
function cosine_similarity(array $vec1, array $vec2): float {
    $dotProduct = function(array $arr1, array $arr2): float {
        return array_sum(
            array_map(fn($a, $b) => $a * $b, $arr1, $arr2)
        );
    };

    return $dotProduct($vec1, $vec2) /
    (
        sqrt($dotProduct($vec1, $vec1))
        * sqrt($dotProduct($vec2, $vec2))
    );
}
```

In pursuit of speed...

- Reimplement algorithm in multiple ways.
- Do the benchmark on three data sets:
  - Small (5 elements)
  - Middle (15 elements)
  - Big (35 elements)

Pure PHP

“Functional” PHP

C (FFI)


C (extension)
RUST
Assembly (SSE)
imgflip.com

C version

```c
double cosine_similarity(
    const double *array1,
    const double *array2, int size
) {
    double dot_product = 0.0;
    double magnitude_a = 0.0;
    double magnitude_b = 0.0;
    int i;

    for (i = 0; i < size; ++i) {
        dot_product += array1[i] * array2[i];
        magnitude_a += array1[i] * array1[i];
        magnitude_b += array2[i] * array2[i];
    }

    return dot_product / (sqrt(magnitude_a) * sqrt(magnitude_b));
}
```

C (extension)

```c
double cosine_similarity(
   const double *array1,
   const double *array2, int size
) {
   double dot_product = 0.0;
   double magnitude_a = 0.0;
   double magnitude_b = 0.0;
   int i;

   for (i = 0; i < size; ++i) {
       dot_product += array1[i] * array2[i];
       magnitude_a += array1[i] * array1[i];
       magnitude_b += array2[i] * array2[i];
   }

   return dot_product / (sqrt(magnitude_a) * sqrt(magnitude_b));
}
```

\+ A lot of boilerplate code

RUST

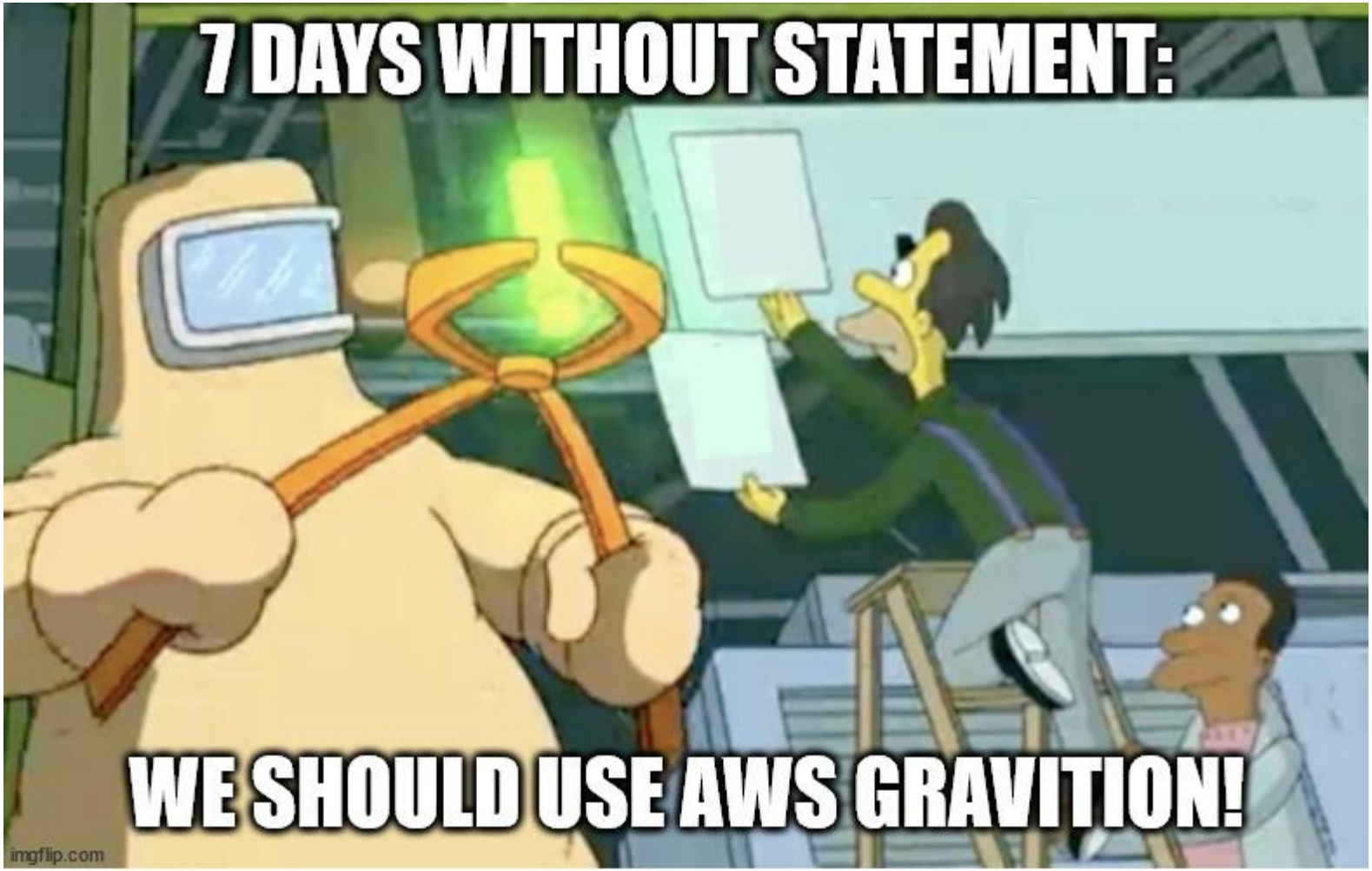
```rust
use ext_php_rs::prelude::*;
use ext_php_rs::types::Zval;

#[php_function]
pub fn cosine_similarity_rust(array1: Vec<f64>, array2: Vec<f64>) -> Result<Zval, String> {
    let mut dot_product = 0.0;
    let mut magnitude_a = 0.0;
    let mut magnitude_b = 0.0;

    for (x, y) in array1.iter().zip(array2.iter()) {
        dot_product += x * y;
        magnitude_a += x.powi(2);
        magnitude_b += y.powi(2);
    }

    let result = dot_product / (magnitude_a.sqrt() * magnitude_b.sqrt());

    let mut zval_result = Zval::new();
    zval_result.set_double(result);
    Ok(zval_result)
}
```

RUST

```rust
use ext_php_rs::prelude::*;
use ext_php_rs::types::Zval;

#[php_function]
pub fn cosine_similarity_rust(array1: Vec<f64>, array2: Vec<f64>) -> Result<Zval, String> {
    let mut dot_product = 0.0;
    let mut magnitude_a = 0.0;
    let mut magnitude_b = 0.0;

    for (x, y) in array1.iter().zip(array2.iter()) {
        dot_product += x * y;
        magnitude_a += x.powi(2);
        magnitude_b += y.powi(2);
    }

    let result = dot_product / (magnitude_a.sqrt() * magnitude_b.sqrt());

    let mut zval_result = Zval::new();
    zval_result.set_double(result);
    Ok(zval_result)
}
```

Assembly (SSE2)

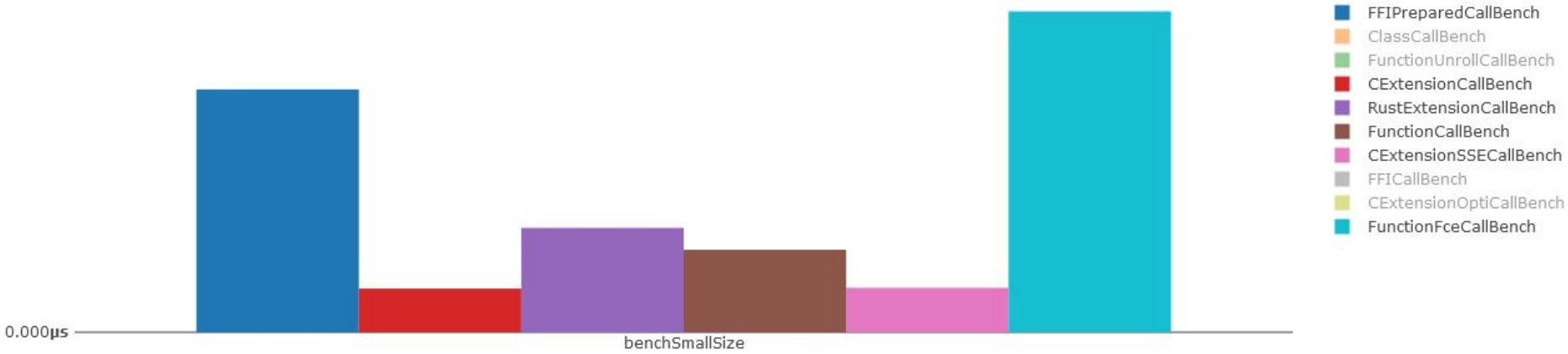
- Hand vectorisation of array calculations.
- Some instructions can work on more data at once (SIMD).
- Won’t work on ARM (eg.: M1, AWS Graviton)
- Ultimately pointless? Compiler did same?


7 DAYS WITHOUT STATEMENT:
WE SHOULD USE AWS GRAVITION!
imgflip.com

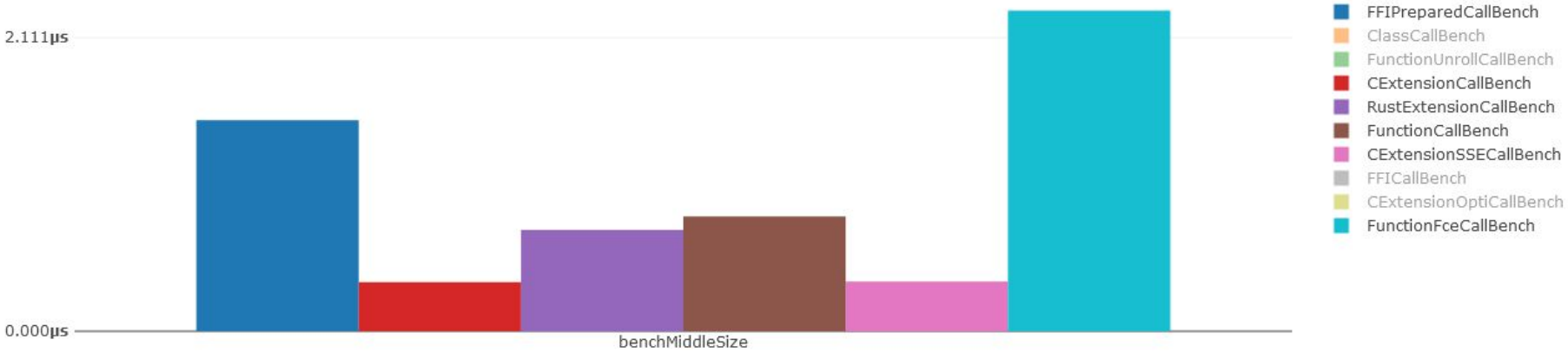
Assembly (SSE2)

```
__m128d sum_dot = _mm_setzero_pd();
__m128d sum_mag_a = _mm_setzero_pd();
__m128d sum_mag_b = _mm_setzero_pd();

int i;
for (i = 0; i <= size - 2; i += 2) {
    __m128d a = _mm_loadu_pd(&array1[i]);
    __m128d b = _mm_loadu_pd(&array2[i]);

    sum_dot = _mm_add_pd(sum_dot, _mm_mul_pd(a, b));
    sum_mag_a = _mm_add_pd(sum_mag_a, _mm_mul_pd(a, a));
    sum_mag_b = _mm_add_pd(sum_mag_b, _mm_mul_pd(b, b));
}
```

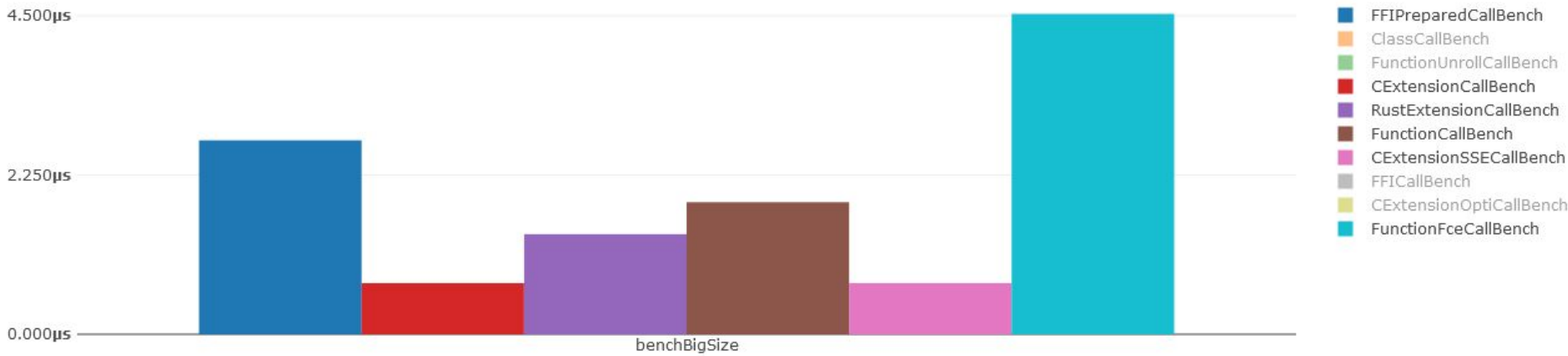
Small dataset



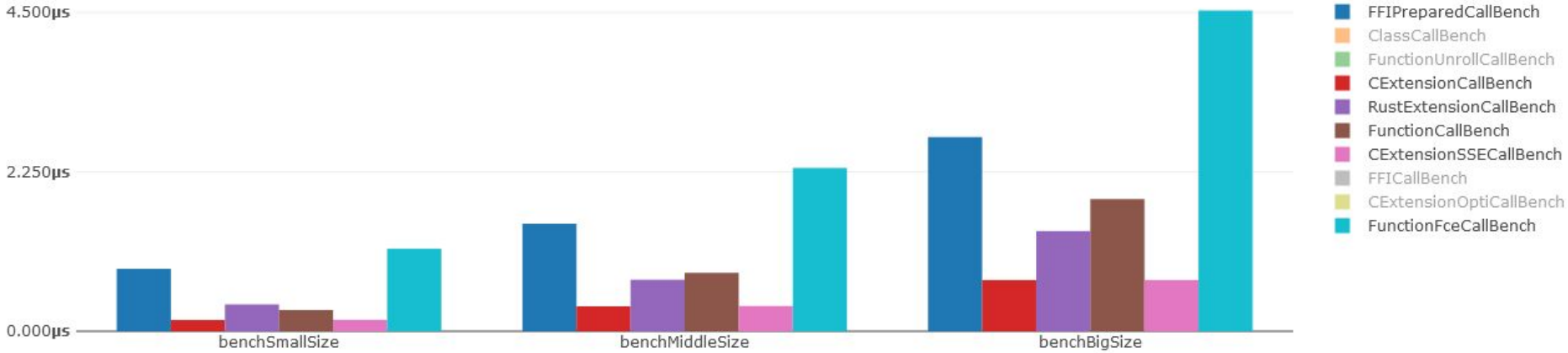
Middle size dataset



Big size dataset



Comparison


4.500µs
2.250µs
0.000µs
benchSmallSize
benchMiddleSize
benchBigSize
FFIPreparedCallBench
ClassCallBench
FunctionUnrollCallBench
CExtensionCallBench
RustExtensionCallBench
FunctionCallBench
CExtensionSSECallBench
FFICallBench
CExtensionOptiCallBench
FunctionFceCallBench

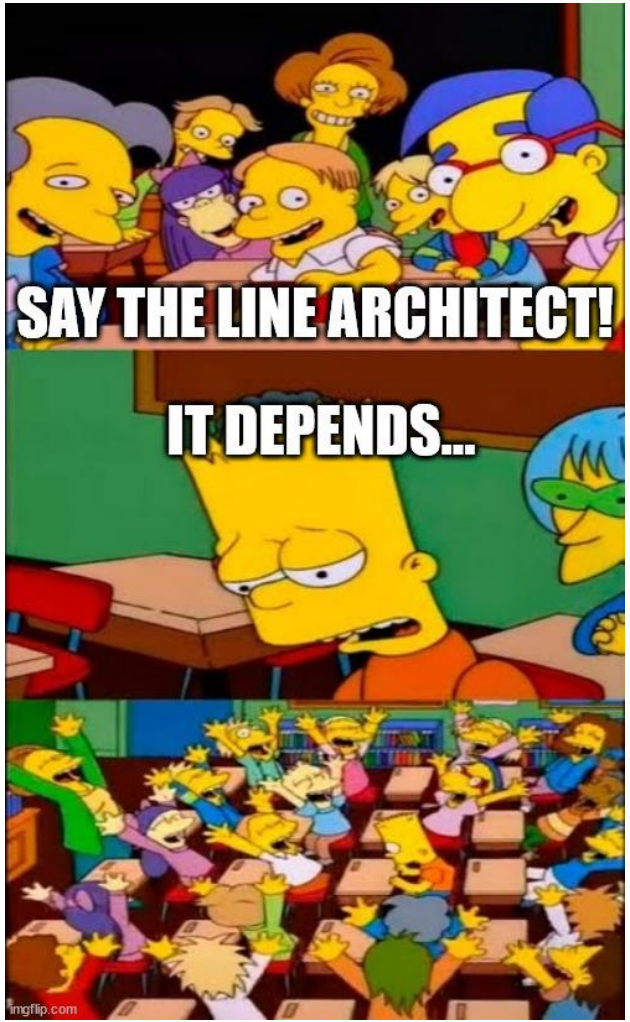
Recapitulation

- Pseudo functional version is slow (they are not identical after all).
- Small dataset - calling overhead is markant.
- With bigger dataset calling overhead diminish.

Interpretation

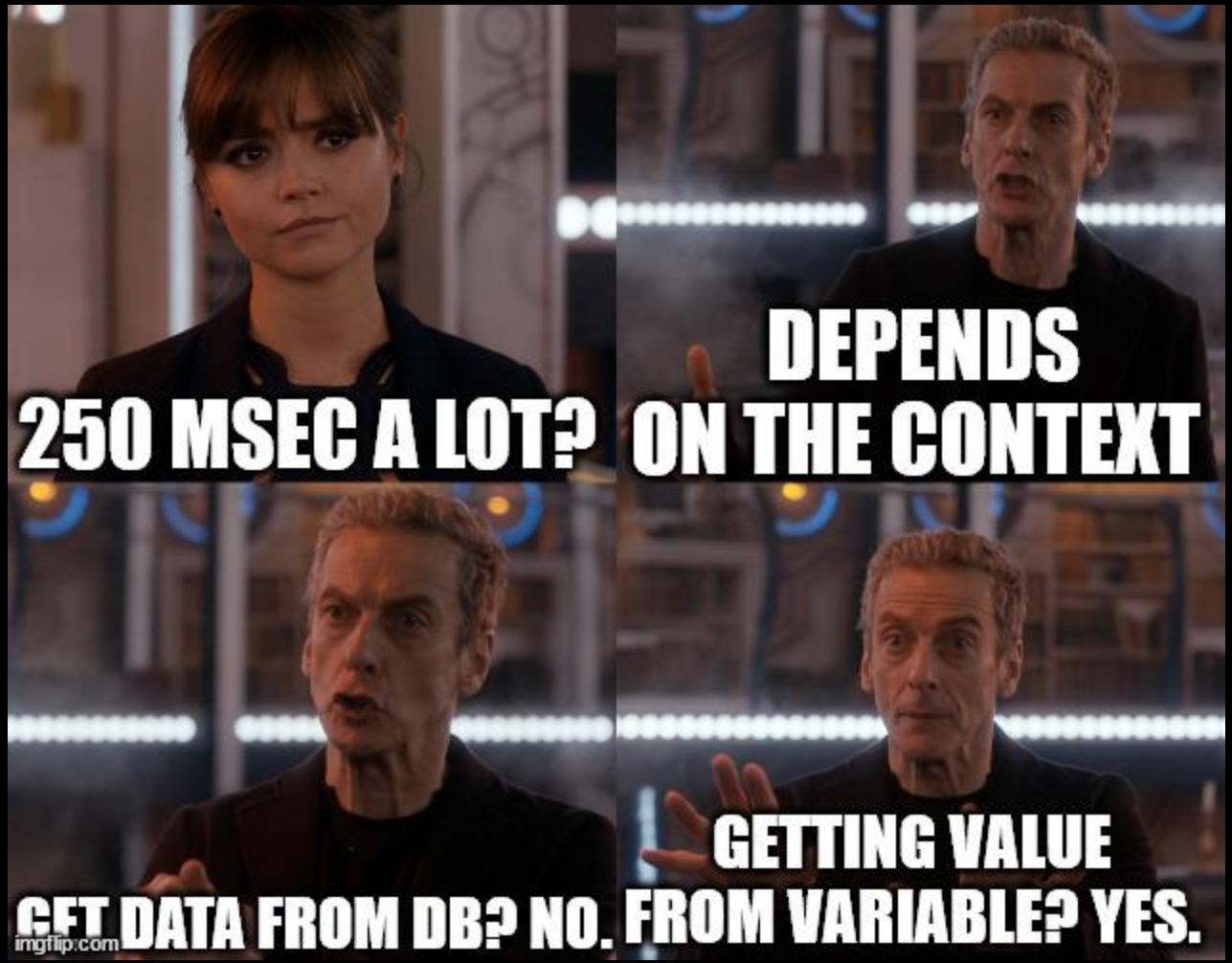
- FFI overhead is high.
- Handwritten assembly is roughly same.
- Rust has a bigger calling overhead then C.
- We are still in the micro-second range.

Is it worth it?


SAY THE LINE ARCHITECT!
IT DEPENDS...
imgflip.com

Is it worth it?

- Depends on situation.
- In some cases it can help.
- You have to properly measure.


250 MSEC A LOT?
DEPENDS
ON THE CONTEXT
GET DATA FROM DB? NO.
GETTING VALUE
FROM VARIABLE? YES.
imgflip.com

Is it worth it?

- It can be niche areas.
- Consider other areas:
  - Comprehension
  - Maintainability
- Real impact?

Is it worth it?

Linux 6.8 Network Optimizations Can Boost TCP Performance For Many Concurrent Connections By ~40%

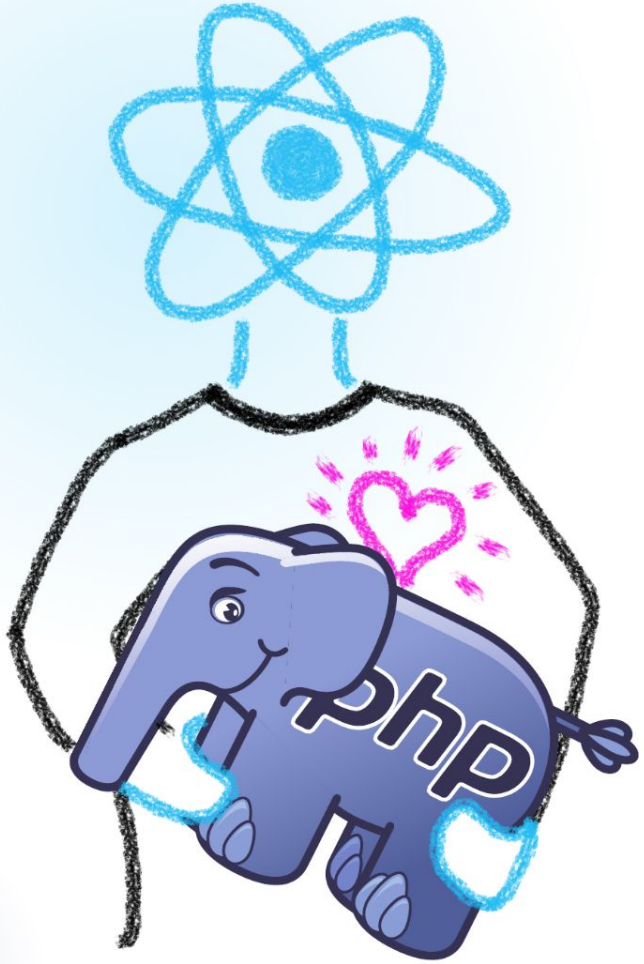
https://www.phoronix.com/news/Linux-6.8-Networking

Linux 6.8 Network Optimizations

- Network speed up by change of order definition in source code.
- Most import question:

Will we fell any difference after switching to 6.8 kernel?

Thanks for attention!



Propojení světa PHP ♡ Reactu

Restaurace u Salzmannů

27/3/2024 | 18:00

React je jeden z nejpoužívanějších frameworků na světě, pojďme se podívat na to, jakým způsobem ho efektivně využít a propojit ve stávající PHP aplikaci. Čekají nás tato témata:

- používání React komponent v PHP aplikaci
- předávání dat z PHP do Reactu synchronně i asynchronně
- pokročilá komunikace mezi komponentami

Vstup zdarma

RTsoft
informační systémy na míru