High Performance PHP

By Ilia Alshanetsky

OOP Tweaks

Always declare your statics!

Dynamic methods accessed statically are 50%+ slower.

```
protected static function loadExternalFile( $file )
    {
        if ( file_exists( $file ) )
        {
            require( $file );
        }
        else
        {
            throw new ezcBaseFileNotFoundException( $file );
        }
    }
```

Quick Static Benchmark

```php
<?php
class bench {
    public function a() { return 1; }
    public static function b() { return 1; }
}

$s = microtime(1);
for ($i = 0; $i < 100000; $i++) bench::a();
$e = microtime(1);
echo "Dynamic Static Method: ".($e - $s)."\n";

$s = microtime(1);
for ($i = 0; $i < 100000; $i++) bench::b();
$e = microtime(1);
echo "Declared Static Method: ".($e - $s)."\n";
```

Conclusion?

Declaring static methods as was done in eZComponents gives us a 60% speed boost.

Use Class Constants!

```
class ezcInputForm
{
    const VALID = 0;


    const INVALID = 1;

    const DEF_NO_ARRAY                          = 1;
    const DEF_EMPTY                             = 2;
    const DEF_ELEMENT_NO_DEFINITION_ELEMENT = 3;
    const DEF_NOT_REQUIRED_OR_OPTIONAL          = 5;
    const DEF_WRONG_FLAGS_TYPE                  = 6;
    const DEF_UNSUPPORTED_FILTER                = 7;
    const DEF_FIELD_NAME_BROKEN                 = 8;
}
```

Advantages

* Parsed at compile time, no execution overhead.
* Faster lookups due to a smaller hash.
* “Namespacing” & shorter hash names.
* Cleaner code speeds up debugging ;-)

require_once() is once too many!

```
ilia@ilappy:~/ezcomponents-1.1rc1 {692}$ grep -nrI \
"require_once\((\| \)" * | grep -v tests \
| grep -v docs | grep -v "\*"

Database/src/test.php:228:   require_once( $path );

ilia@ilappy:~/ez/ezcomponents-1.1rc1 {693}$
```

Why does eZComponents not use a seemingly helpful require_once() and instead insists on using require() + manual duplication checks?

What happens in the background?

```
<?php
require_once "./a.php";
require_once "./a.php";
```

```
lstat64("/tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=7368, ...}) = 0
lstat64("/tmp/a.php", {st_mode=S_IFREG|0644, st_size=6, ...}) = 0
open("/tmp/a.php", O_RDONLY)                = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=6, ...}) = 0
fstat64(3, {st_mode=S_IFREG|0644, st_size=6, ...}) = 0

lstat64("/tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=7368, ...}) = 0
lstat64("/tmp/a.php", {st_mode=S_IFREG|0644, st_size=6, ...}) = 0
open("/tmp/a.php", O_RDONLY)                = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=6, ...}) = 0
fstat64(3, {st_mode=S_IFREG|0644, st_size=6, ...}) = 0
```

The “ONCE” Problem

- Require/Include Once constructs open file on each call!
  - Fixed in PHP 5.2/6.0 for full paths
  - Fixed if using CVS version of APC
- Increased File IO → Drop in Speed

Avoid Pointless Function Calls

Base/src/base.php:308

```
case self::DEP_PHP_VERSION:
        $phpVersion = phpversion();
        if ( version_compare( $phpVersion, $value, '>=' ) )
```

Archive/src/archive.php:436

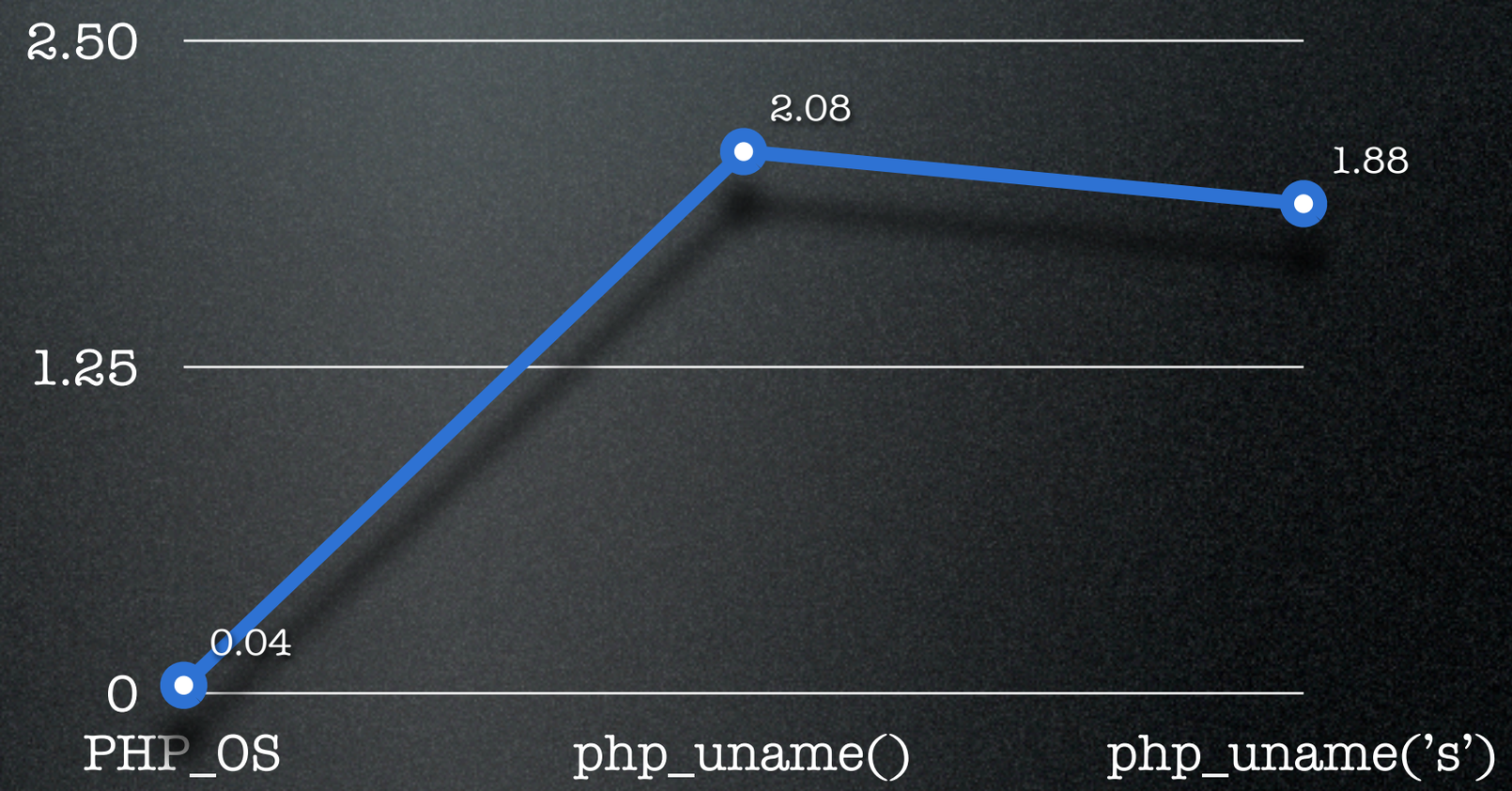
```
$isWindows = ( substr( php_uname('s'), 0, 7 ) == 'Windows' )?
true:false;
```

Mail/src/parser/parser.php:95

```
$uname = php_uname();
if ( strtoupper( substr( $uname, 0, 3 ) == "WIN" ) )
```

Use Native Constants

- php_uname('s') == PHP_OS
- php_version() == PHP_VERSION
- php_sapi_name() == PHP_SAPI



Fastest Win32 Detection in the West!

```
$isWindows =
  DIRECTORY_SEPARATOR == '\\';
```

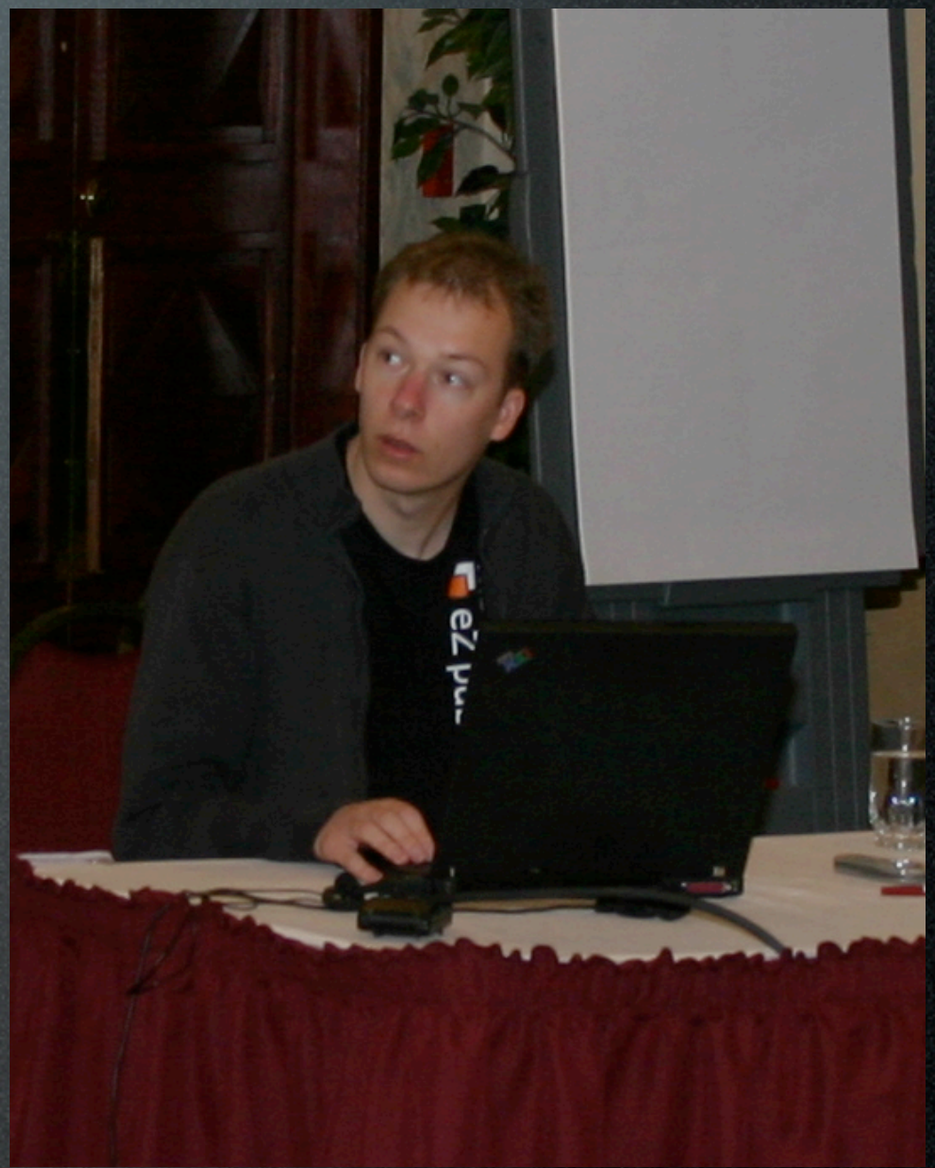
- Does not use functions
- Does not care about WinXP, WinNT, Windows, Windows98, NT 5.0, etc...
- Always available

What time is it?

Rather then calling time(), time() and time() again, use

$_SERVER ['REQUEST_TIME']

Provides a timestamp, with a second precision, without any function calls.



PCRE's Slowdowns

Mail/src/tools.php:318

```
$text = preg_replace( '/=\?([^?]+)\?/', '=?iso-8859-1?', $origtext );
```

Template/src/functions/string_functions.php:185

```
self::functionCall( "preg_replace",
   array( self::constant('"/(\n|\t|\r\n|\s)+/"'),
   self::value(" "), "%string") )
```

Non-Capturing Patterns

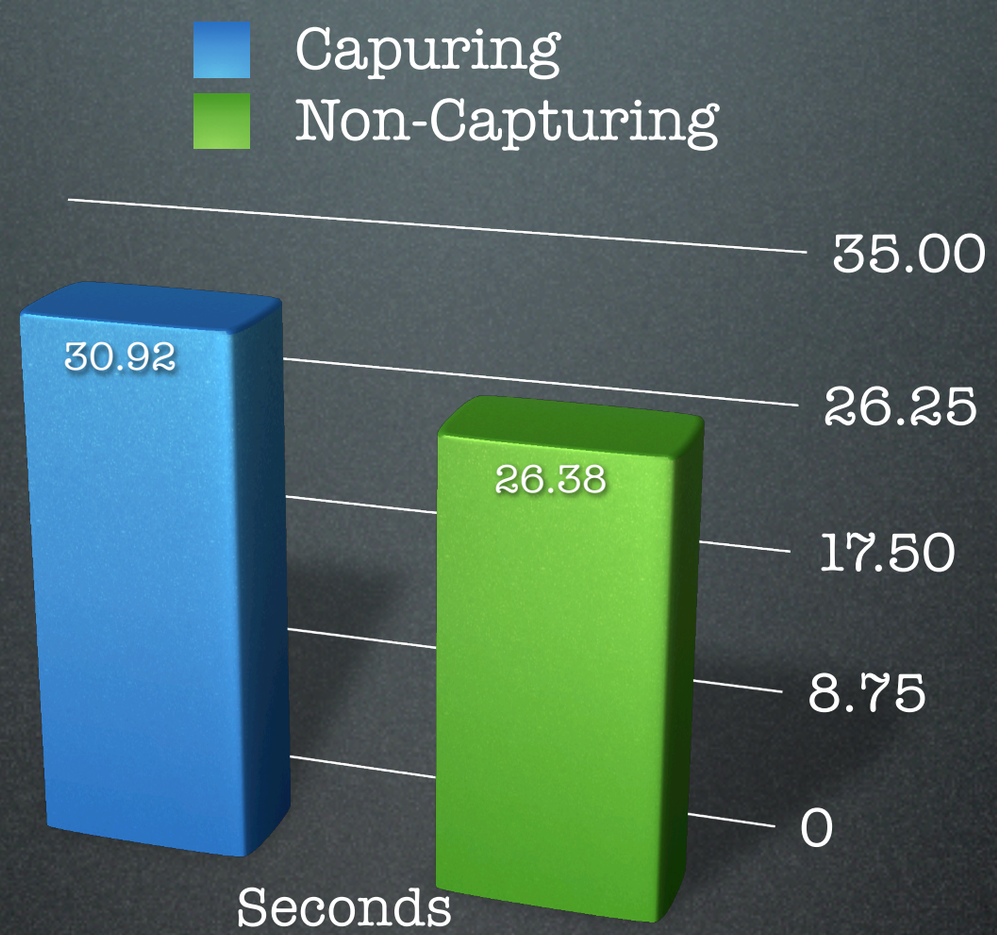
Placing ?: at the start of a sub-pattern makes it non-capturing.

```
$text = preg_replace( '/=\?(?:[^?]+)\?/', '=?iso-8859-1?', $origtext );
```

This means PHP/PCRE does not need to allocate memory to store the matched content block.

```
self::functionCall( "preg_replace",
   array( self::constant('"/(?:\n|\t|\r\n|\s)+/"'),
   self::value(" "), "%string") )
```

End Result?



A 15% performance improvement, with a 2 character change.

Avoid Regex if Possible

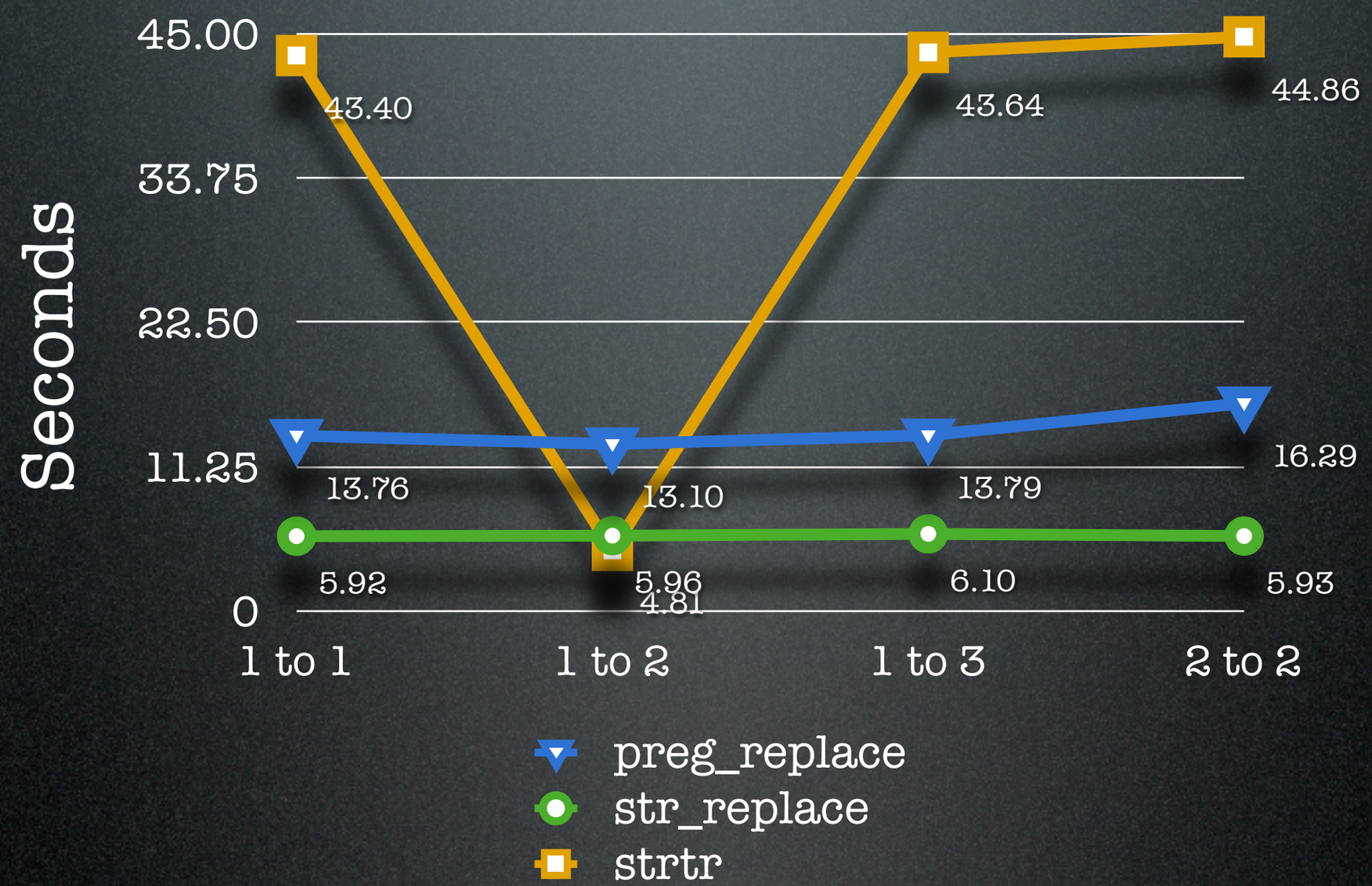
Template/src/parsers/ast_to_php/implementations/php_generator.php:287
Template/src/parsers/ast_to_php/implementations/php_generator.php:336

```
$this->write( '"'. addcslashes(
  preg_replace( "/\n/", "\\n", $type->value), '"' ) . '"');
```

In this case it would be simpler and to mention faster
to use a regular str_replace()

```
$this->write( '"'. addcslashes(
  str_replace( "\n", "\\n", $type->value), '"' ) . '"');
```

Speed Comparison



Use strtr() properly

While browsing Cache/src/storage/file.php, I found the following code:

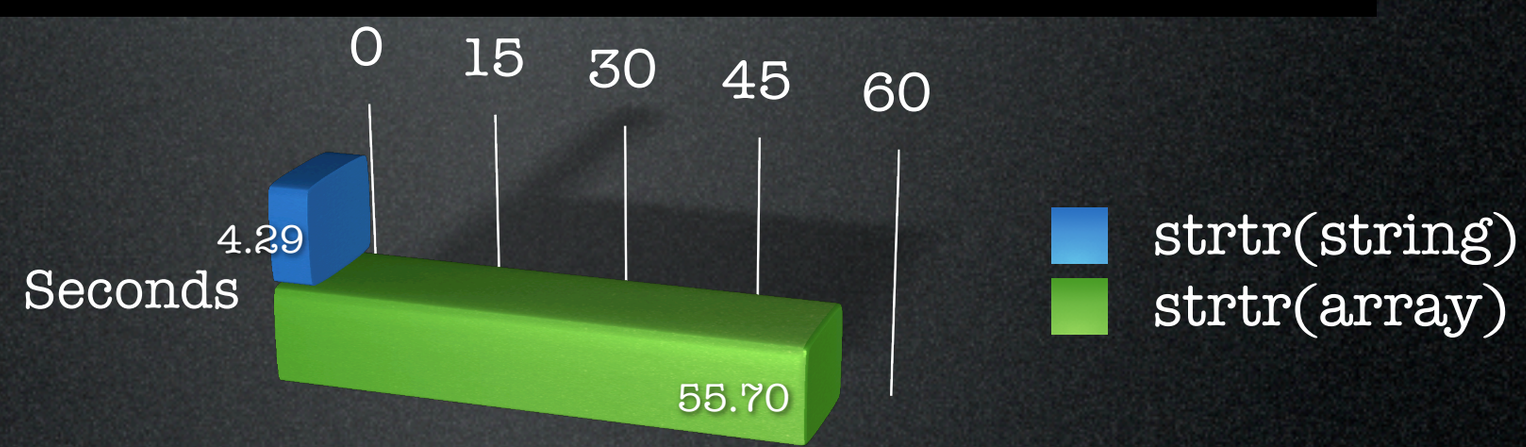
```
if ( sizeof( $globArr ) > 1 )
{
   $glob = $globArr[0]  . "-" . strtr( $globArr[1], array
   ( '-' => '*', '.' => '*' ) );
}
else
{
   $glob = strtr( $globArr[0], array( '-' => '*', '.' => '*' ) );
}
```

Any ideas on how we can make this code 10 times faster?

Use strings!

Elimination of array operations speeds up the code and simplifies the internal work in strtr() function.

```
if ( sizeof( $globArr ) > 1 )
{
   $glob = $globArr[0]  . "-" . strtr( $globArr[1], '-.','**');
}
else
{
   $glob = strtr( $globArr[0], '-.','**');
}
```



Don’t Replace When you don’t have to!

Any replacement operation requires memory, if only to store the “modified” result.

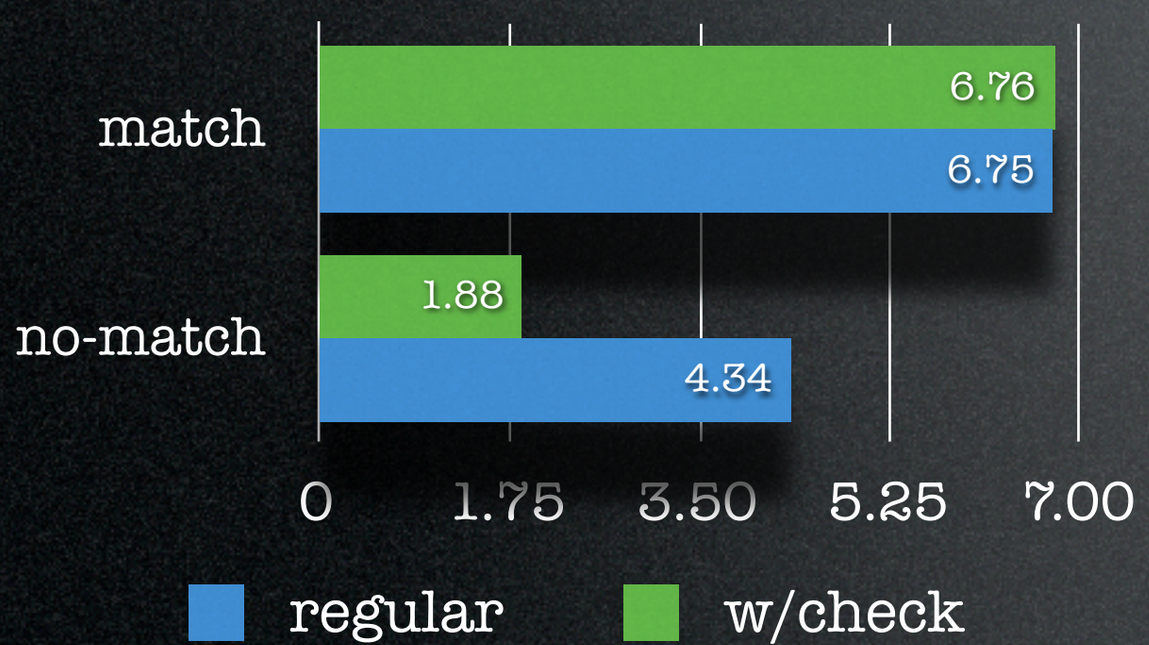
A quick strpos() to determine if any replacement is actually needed can save memory and improve performance!

Test Scenario

```
$s = microtime(1);
for ($i = 0; $i < 10000; $i++)
    str_replace('ZendStudio', 'ezPublish', $str);
$e = microtime(1);
echo "non-check (no-match): ".($e - $s)."\n";

$s = microtime(1);
for ($i = 0; $i < 10000; $i++)
    if (strpos($str, 'ZendStudio') !== false)
        str_replace('ZendStudio', 'ezPublish', $str);
$e = microtime(1);
echo "check (no-match): ".($e - $s)."\n";
```

$str is a PHP 5.2 news files, roughly 95kb in size.




```
$s = microtime(1);
for ($i = 0; $i < 10000; $i++)
    str_replace('Ilia', 'Derick', $str);
$e = microtime(1);
echo "non-check (match): ".($e - $s)."\n";

$s = microtime(1);
for ($i = 0; $i < 10000; $i++)
    if (strpos($str, 'Ilia') !== false)
        str_replace('Ilia', 'Derick', $str);
$e = microtime(1);
echo "check (match): ".($e - $s)."\n";
```

@ operator is evil!

The error blocking operator, is the most expensive letter in PHP's alphabet.

```
@action();
```

This seemingly innocuous operator actually performs fairly intensive operations in the background.

Fortunately, for ezComponents users, it is used only a few dozen times.

To @ or not to @?

```
$s = microtime(1);
for ($i = 0; $i < 100000; $i++) @phpversion();
$e = microtime(1);
echo "Error Block: ".($e - $s)."\n";
```

3X speed difference !!

VS

```
$s = microtime(1);
for ($i = 0; $i < 100000; $i++) {
    error_reporting(0);
    for ($j = 0; $j < 5; $j++)
        phpversion();
    error_reporting(E_ALL);
}
$e = microtime(1);
echo "Normal: ".($e - $s)."\n";
```

```
$s = microtime(1);
for ($i = 0; $i < 100000; $i++) phpversion();
$e = microtime(1);
echo "Normal: ".($e - $s)."\n";
```

```
$s = microtime(1);
for ($i = 0; $i < 100000; $i++)
    for ($j = 0; $j < 5; $j++) @phpversion();
$e = microtime(1);
echo "Error Block: ".($e - $s)."\n";
```

With 5 iterations changing error mode manually is even faster!

Comparing Strings

The good

```
if (!strncmp(PHP_OS, 'WIN', 3)) {
```

```
if (!strncasecmp(PHP_OS, 'WIN', 3)) {
```

The bad

```
if (substr(php_uname('s'), 0, 3) == 'WIN') {
```

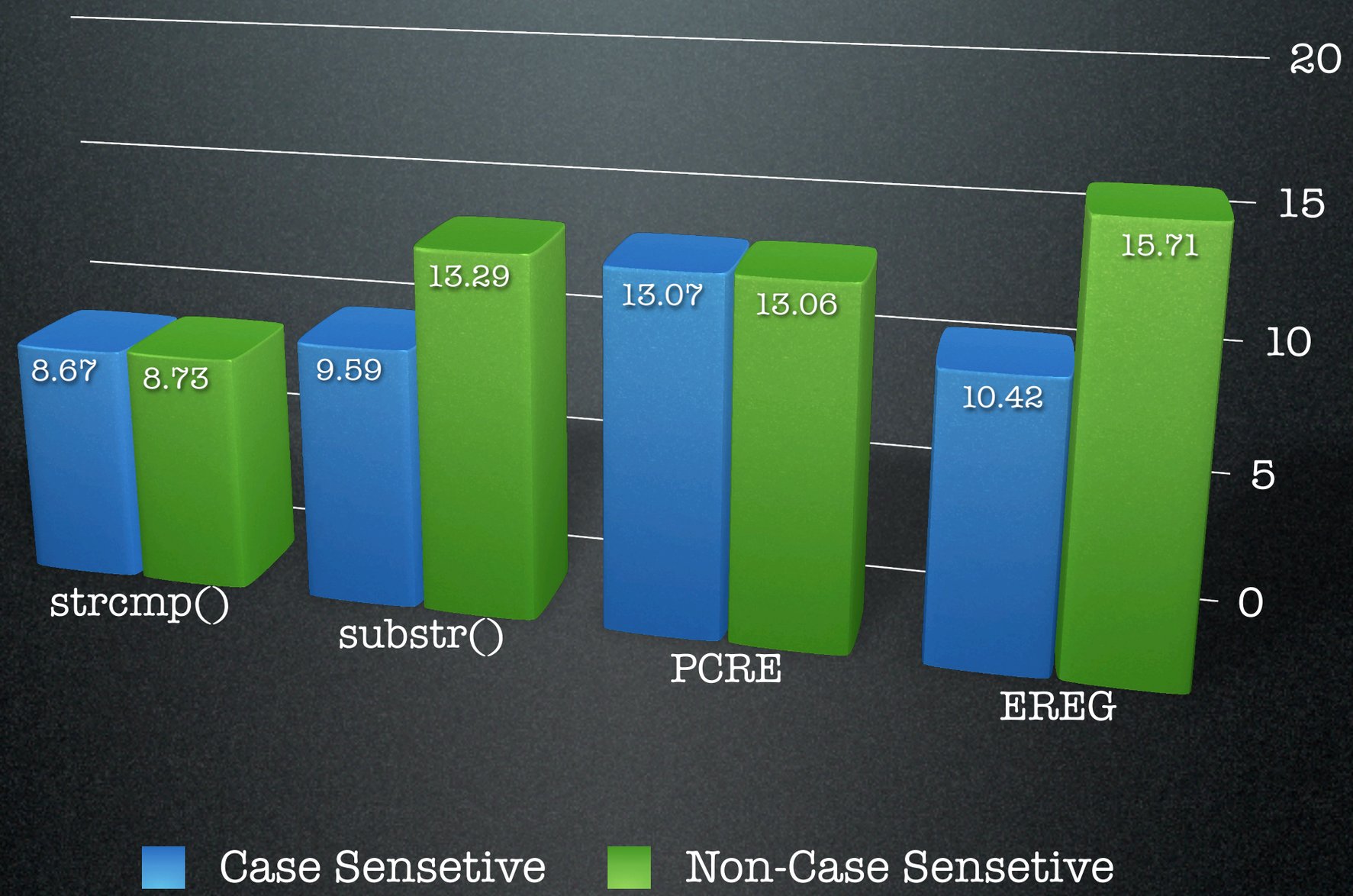
```
if (strtolower ( substr(php_uname('s'), 0, 3)))  == 'win') {
```

And the ugly

```
if (preg_match('!^WIN!', php_uname('s'))) {
```

```
if (preg_match('!^WIN!i', php_uname('s'))) {
```

Quick Benchmark



Compare from an offset

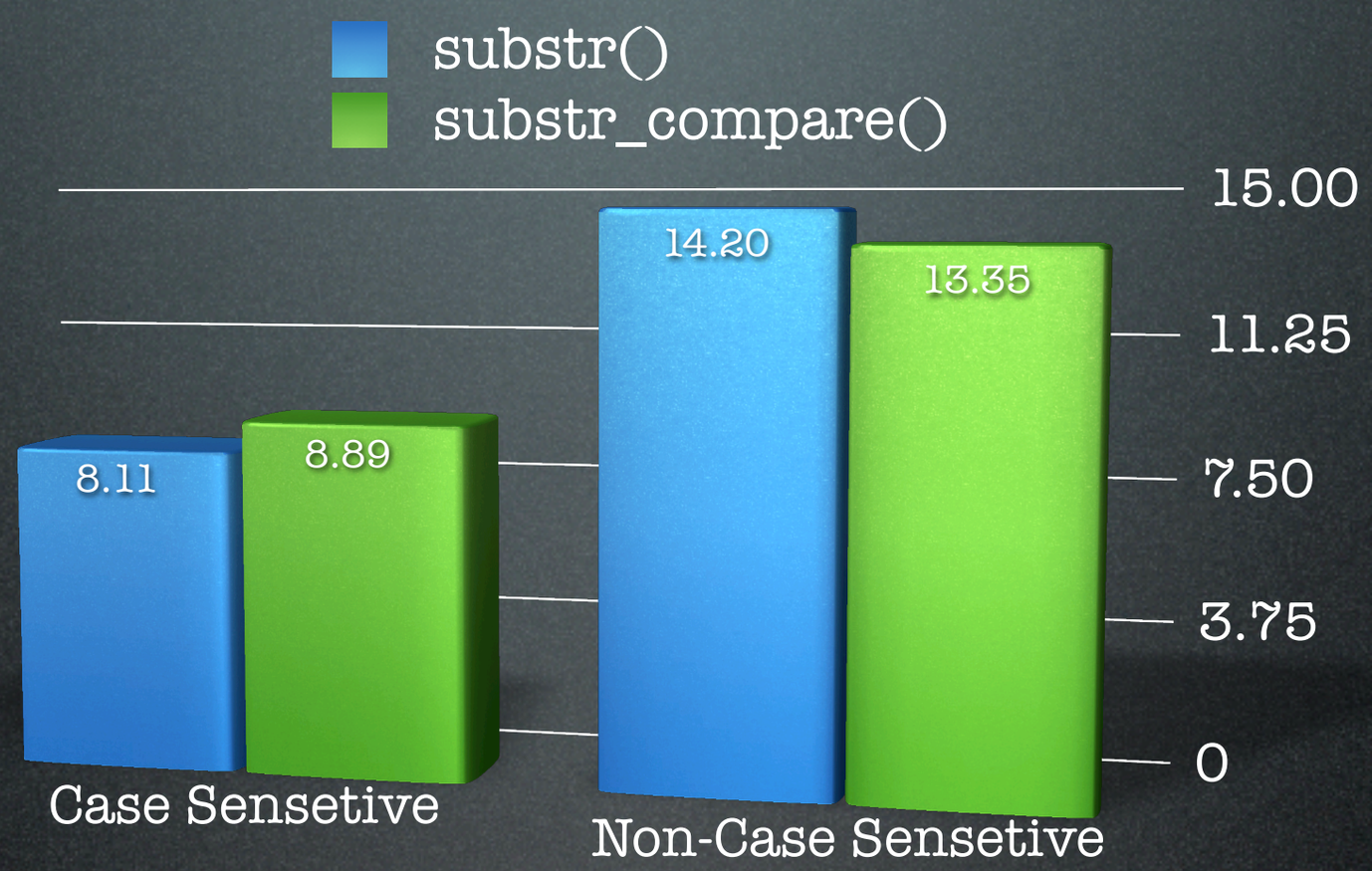
As of PHP 5, you don't need to substr() string segments from non-start position to compare them thanks to substr_compare().

```
if ( substr( $class, -15 ) != 'OperatorTstNode' )

/* == */

if ( substr_compare($class, 'OperatorTstNode', -15) )
```

But is it faster?



Unfortunately, in most cases, the answer is NO.

Unless:

* Comparing case-insensitively
* Comparing large strings

PHP "Wackiness"

Which would you think be faster

```
implode( ', ', array_value( $_SERVER ) );
```

or perhaps?

```
implode( ', ', $_SERVER );
```

Surprise!!

Unless you are using PHP-CVS the longer, implode() + array_values() is actually faster then direct implode().

constants != strings

One of my biggest pet-peeves in PHP is this kind of nonsense:

```
$foo = array("bar"=>0);
$foo[bar] = 1;
```

Fortunately, ezComponents does not use it ;-)

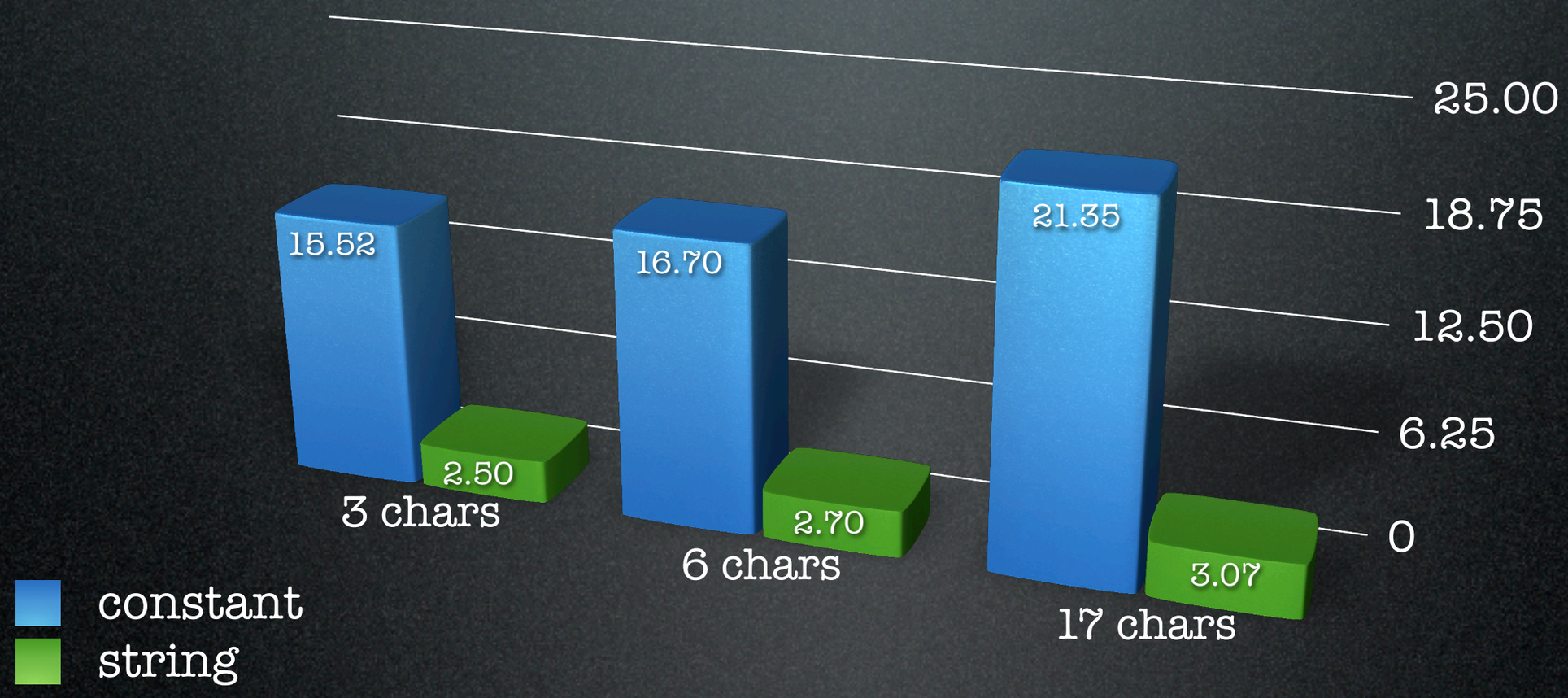
Why is it bad?

A whole slew of pointless operations:

* 2 hash lookups
* tolower on the constant name
* E_NOTICE about an undefined constant
* temporary string creation

Quick Benchmark

```
$foo[bar] = 1; /* vs */ $foo['bar'] = 1;
```



700% difference on average!!!

Simplify for() loop

If speed is of the essence don't do

ConsoleTools/src/table.php:515

```
for ( $i = 1; $i < sizeof( $rowParts ); $i++ )
```

or this

ConsoleTools/src/input.php:690

```
for ( $i = 0;  $i < count( $this->rows ); $i++ )
```

or that

Mail/src/tools.php:181

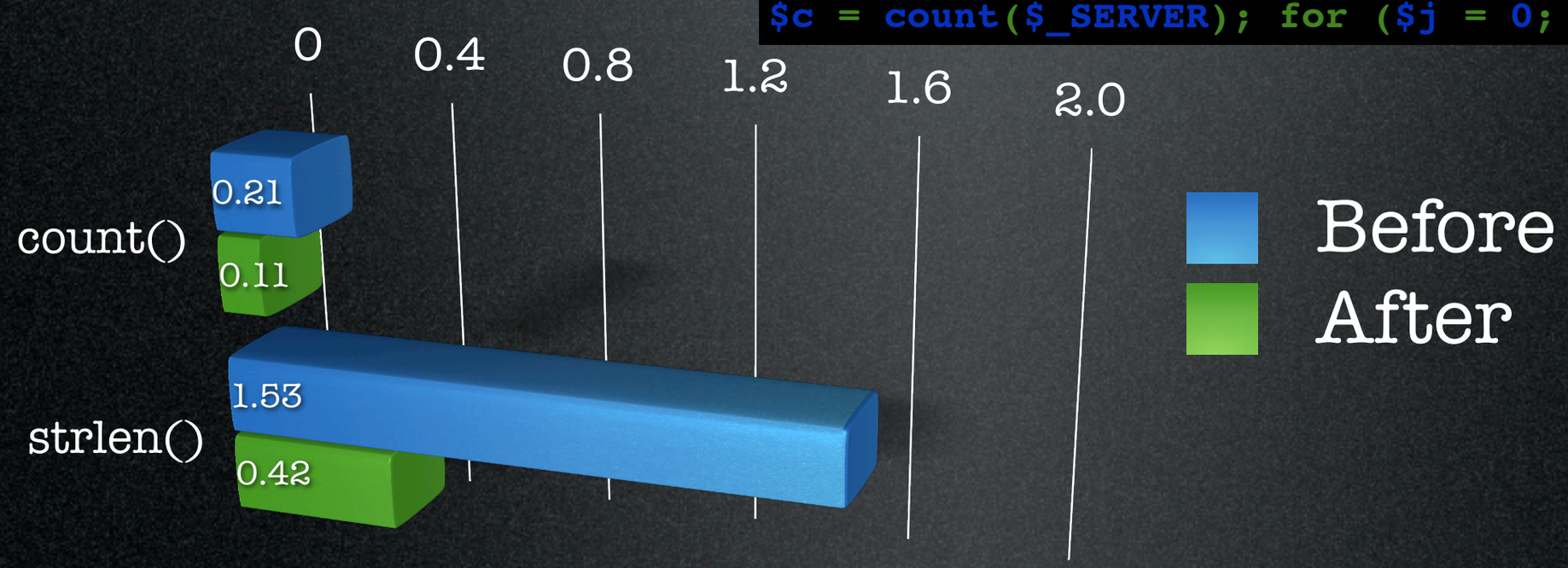
```
for( $i = 0; $i < strlen( $addresses ); $i++ )
```

Simplify for() loop

By taking out the function out of the for() you save exactly 1 function call per iteration.

```
for ($j = 0; $j < strlen('foo'); $j++) {}
/* vs */
$c = strlen('foo'); for ($j = 0; $j < $c; $j++) {}
```

```
for ($j = 0; $j < count($_SERVER); $j++) {}
/* vs */
$c = count($_SERVER); for ($j = 0; $j < $c; $j++) {}
```



Shorter != Faster

For reasons yet to be determined, people think that shorter code translates to faster code. More often then not, it is simply WRONG!

Let's Compare

Database/src/sqlabstraction/query_update.php (170 - 179)

```
if ( $setString === null )
{
  $setString = "{$key} = {$value}";
}
else
{
  $setString .= ", {$key} = {$value}";
}
```

```
if ($setString) {
    $setString .= ', ';
}
$setString .= "{$key} = {$value}";
```

```
$setString .= ($setString ? ', ' : ''). "{$key} = {$value}";
```

Speed Difference

Let's try again

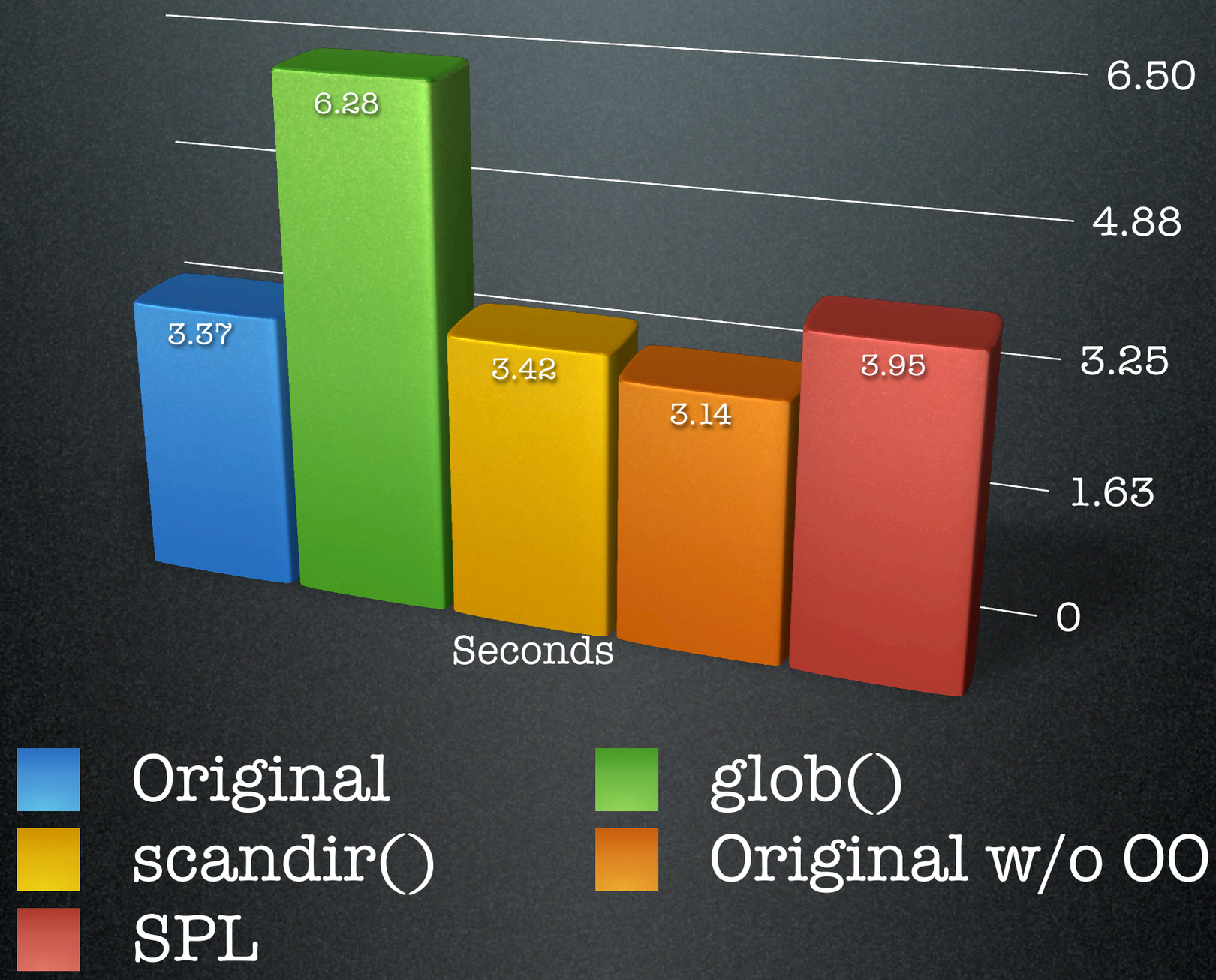
Paraphrased from File/src/file.php

```
/* original */
$d = dir(".");
while (($entry = $d->read() ) !== false) {
    if ( $entry == '.' || $entry == '..' ) {
      continue;
    }
}

/* versus */
glob("./*");

/* versus */
scandir(".") /* includes . and .. */
```

Results


6.50
4.88
3.25
1.63
0
3.37
6.28
3.42
3.14
3.95
Seconds
Original
glob()
scandir()
Original w/o OO
SPL

Strings & Variables

```
$a = "{$key} {$value}";

$a = "$key $value";

$a = $key.' '.$value;

$a = <<<H
$key $value
H;

$a = <<<H
{$key} {$value}
H;
```

eZ Comp.

Longer is still faster ;-)

Recursion & Performance

```
function rdir($dir)
{
   $d = opendir($dir);
   while ($f = readdir($d)) {
      if ($f == '.' || $f == '..')
         continue;

      if (is_dir($dir . '/' . $f)) {
         rdir($dir . '/' . $f);
         continue;
      }
   }
   closedir($d);
}
```

```
function stackdir($v)
{
   $dirs = array();
   do {
      $d = opendir($v);
      while ($f = readdir($d)) {
         if ($f == '.' || $f == '..')
            continue;

         if (is_dir($v . '/' . $f)) {
            $dirs[] = $v . '/' . $f;
            continue;
         }
      }
      closedir($d);
   } while ($v = array_pop($dirs));
}
```

eZ Components

```
new RecursiveIteratorIterator(new RecursiveDirectoryIterator())
```

Recursion & Performance

Thank you for listening

Slides available at: http://www.ilia.ws/