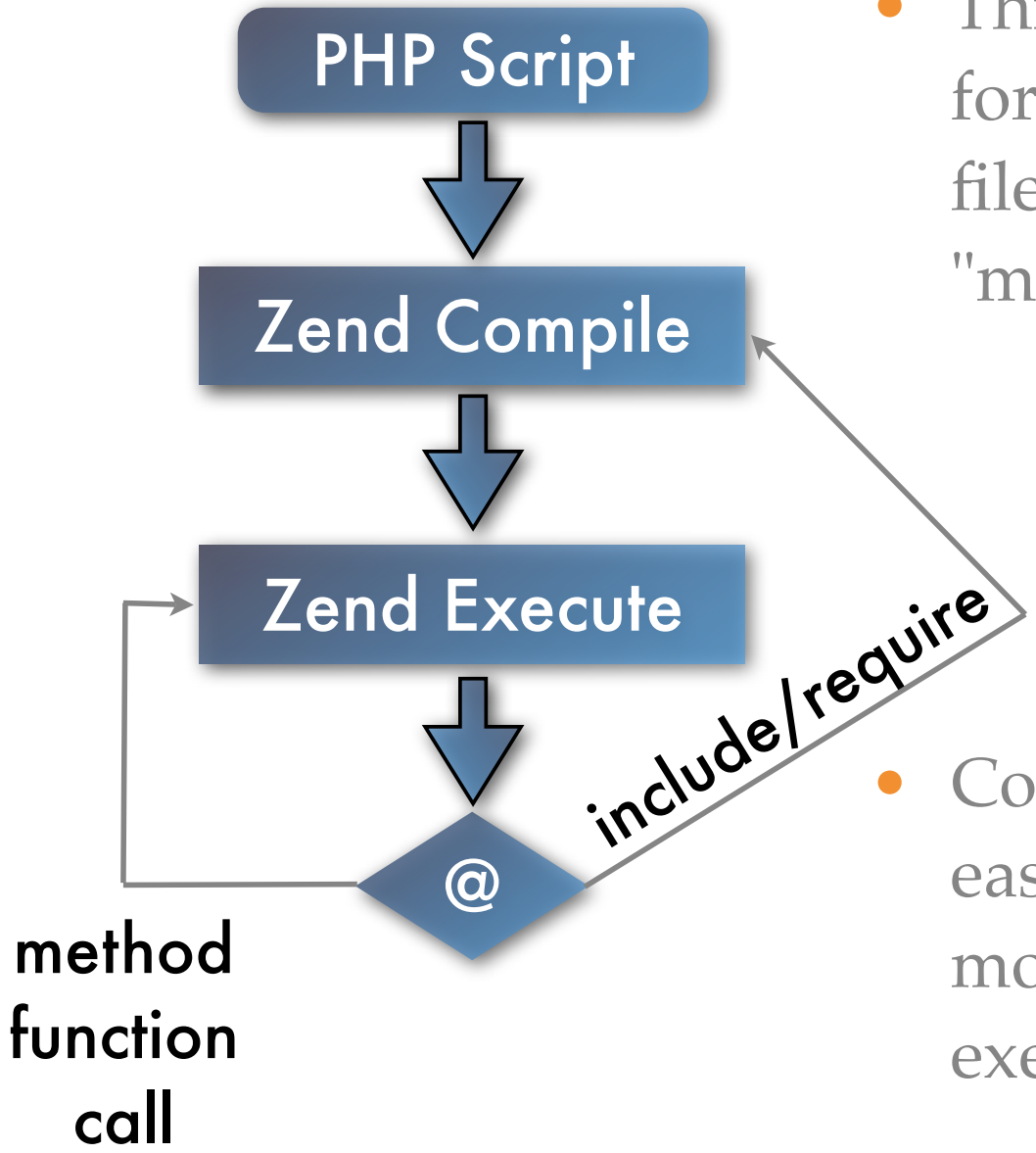
PHP & Performance

By: Ilia Alshanetsky

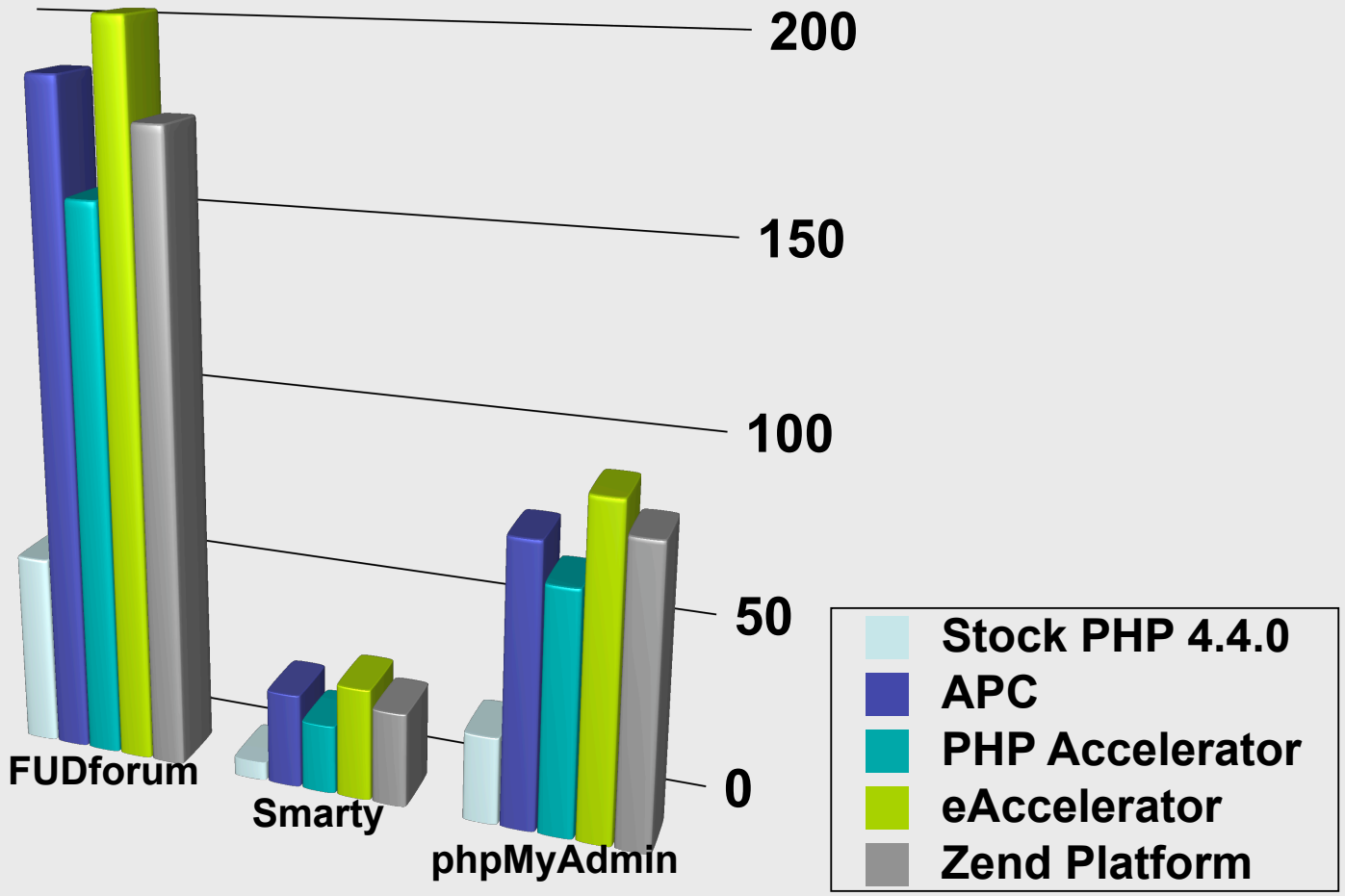


- This cycle happens for every include file, not just for the "main" script.
- Compilation can easily consume more time than execution.

Compiler/Opcode Caches

- Each PHP script is compiled only once for each revision.
- Reduced File IO, opcodes are being read from memory instead of being parsed from disk.
- Opcodes can optimized for faster execution.

Quick Comparison



Compiler Optimizations

- For absolute maximum performance, ensure that all of the software is compiled to take advantage of the available hardware.

```
export CFLAGS="-O3 -msse -mmmx -march=pentium3 \
-mcpu=pentium3 -funroll-loops -mfpmath=sse \
-fomit-frame-pointer"
```

  - Enable all compiler optimizations with `-O3`
  - Tune the code to your CPU via `-march -mcpu`
  - CPU specific features `-msse -mmmx -mfpmath=sse`
  - Drop debug data `-fomit-frame-pointer`

Web Server: File IO

- ✓ Keep `DirectoryIndex` file list as short as possible.
- ✓ Whenever possible disable `.htaccess` via `AllowOverride none`.
- ✓ Use `Options FollowSymLinks` to simplify file access process in Apache.
- ✓ If logs are unnecessary disable them.
- ✓ If logging is a must, log everything to 1 file and break it up during the analysis stage.

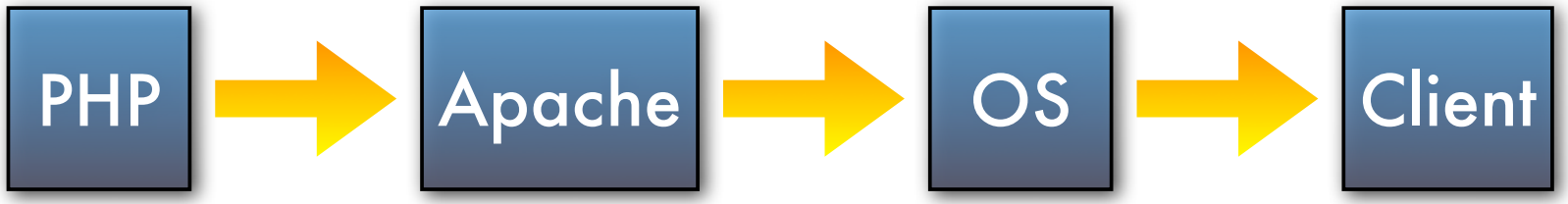
Web Server: Syscalls

- Syscall is function executed by the Kernel. The goal is to minimize the number of these calls needed to perform a request.
  - Do not enable **`ExtendedStatus`**.
  - For Deny / Allow rules use IPs rather then domains.
  - Do not enable **`HostnameLookups`**.
  - Set **`ServerToken=prod`**

Web Server: KeepAlive

- In theory **`KeepAlive`** is supposed to make things faster, however if not used carefully it can cripple the server.
  - In Apache set **`KeepAlive`** timeout, **`KeepAliveTimeout`** as low as possible. **Suggested value: 10 seconds.**
  - If the server is only serving dynamic requests, disable **`KeepAlive`** all together.

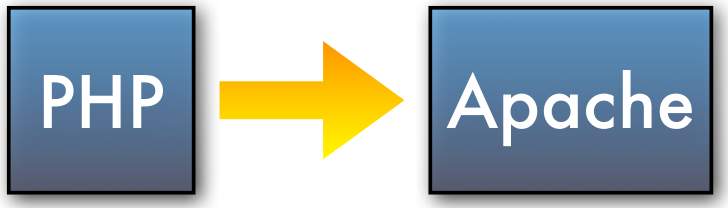
Matching Your IO Sizes



- The goal is to pass off as much work to the kernel as efficiently as possible.
- Optimizes PHP to OS Communication
- Reduces Number Of System Calls

PHP: Output Control

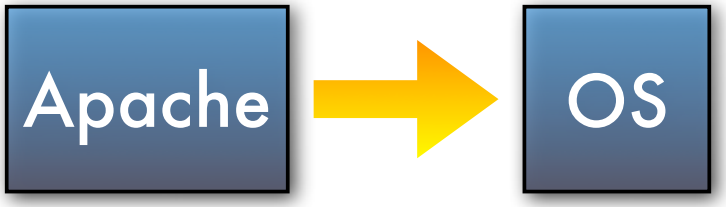
- Efficient
- Flexible



- In your script, with `ob_start()`
- Everywhere, with `output_buffering = On`
- Improves browser's rendering speed

Apache: Output Control

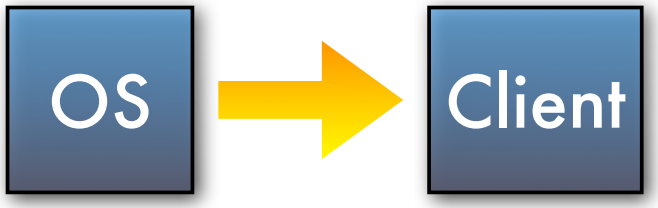
- The idea is to hand off entire page to the kernel without blocking.



- Set **SendBufferSize** = **PageSize**

OS: Output Control

OS (Linux)




```
/proc/sys/net/ipv4/tcp_wmem
4096    16384   maxcontentsize
min      default      max

/proc/sys/net/ipv4/tcp_mem
(maxcontentsize * maxclients) / pagesize
```

- **Be careful on low memory systems!**

Static Content Serving

- While Apache is great for dynamic requests, static requests can be served WAY FASTER by other web servers.
  - `lighttpd`
  - `Boa`
  - `Tux`
  - `thttpd`

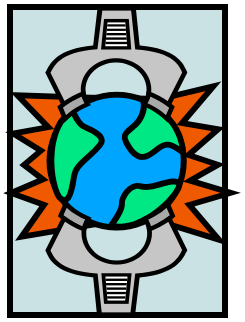
- For static requests these servers are easily 300-400% faster then Apache 1 or 2.

Less Output == Faster

- Saves server bandwidth (saves $$ too).
- Reduces server resource usage (CPU / Memory / Disk)
- Pages load faster for clients.
- Reduces network IO high traffic sites, where it is the primary bottleneck in most cases.

Content Compression

- Most browsers support content compression.
- Compressed pages are on average are 6-8 times smaller.
  - Apache 1 (`mod_gzip / mod_deflate`)
  - Apache 2 (`mod_deflate`)
  - PHP
    - From PHP configuration `zlib.output_compression=1`
    - From inside the script `ob_start("ob_gzhandler")`



* Compression will utilize 3%-5% of CPU.

Content Reduction

```
<?php
$o = array("clean" => true,
  "drop-proprietary-attributes" => true,
  "drop-font-tags" => true,
  "drop-empty-paras" => true,
  "hide-comments" => true,
  "join-classes" => true,
  "join-styles" => true
);

$tidy = tidy_parse_file("php.html", $o);
tidy_clean_repair($tidy);
echo $tidy;
?>
```

- Use a post-processor like Tidy to remove formatting, comments and CCSify the code.

Tuning PHP Configuration

- register_globals = Off **
- magic_quotes_gpc = Off
- expose_php = Off
- register_argc_argv = Off
- always_populate_raw_post_data = Off **
- session.use_trans_sid = Off **
- session.auto_start = Off **
- session.gc_divisor = 1000 or 10000

Profiling & Benchmarking

- Identify Bottlenecks
- Track Resource Usage
- Generate Call Trees
- Create Progress Tracking Data

Testing Web Servers

- Apache Bench
  - **ab** utility bundled with Apache
- Siege
  - http://www.joedog.org/JoeDog/Siege
- http_load (Excellent for latency tests)
  - http://www.acme.com/software/http_load/

Web Server Testing

```
Concurrency Level:      10
Time taken for tests:   0.265 seconds
Complete requests:      100
Failed requests:        0
Broken pipe errors:     0
Total transferred:      5077082 bytes
HTML transferred:       5061168 bytes
Requests per second:    377.36 [#/sec] (mean)
Time per request:       26.50 [ms] (mean)
Time per request:       2.65 [ms] (mean)
Transfer rate:          19158.80 [Kbytes/sec]
```

Latency Test

```
1000 fetches, 5 max parallel,
  2.9648e+07 bytes,
  in 0.813035 seconds
29648 mean bytes/connection
1229.96 fetches/sec,
3.64658e+07 bytes/sec
msecs/connect:
   0.463202 mean, 12.082 max, 0.045 min
msecs/first-response:
   3.12969 mean,  50.783 max, 0.811 min
HTTP response codes:
  code 200 -- 1000
```

1 msec = 0.0001 seconds

Profiling PHP Code

- APD (Pure profiler)
  - http://pecl.php.net/apd
- XDebug (Profiler & Debugger)
  - http://xdebug.org/
- DBG (Profiler & Debugger)
  - http://dd.cron.ru/dbg/

Profiling with APD

- Installation Steps
  - pecl install apd
  - Modify php.ini

```
zend_extension=apd.so
```

Process repeated for
every function/method call

Generating A Trace

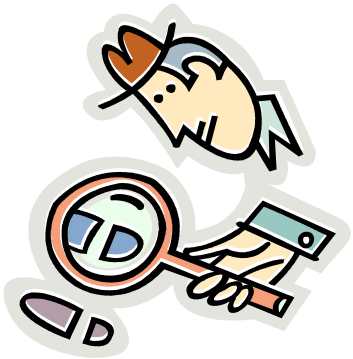
- Profiling of a script starts from the point when the `apd_set_pprof_trace()` function is called.
  - All code executed prior, will not be profiled.

```
$parts = preg_split("!\s!", "a b c");
function test(&$var) {
        $var = base64_encode(trim($var));
}
apd_set_pprof_trace();
array_walk($parts, 'test');
```

- Use the `auto_prepend_file` php.ini setting to activate profiling for an entire application.

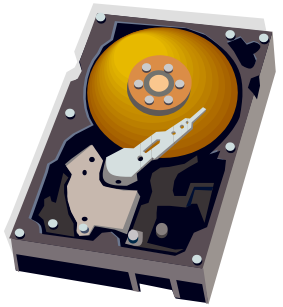
Interpreting the Results

```
 Real    User         System         secs          cumm.
%Time   (excl/cumm)  (excl/cumm)   (excl/cumm)    Calls  call    s/call  Name
-------------------------------------------------------------------------------

82.4    0.00 0.00  0.00 0.00     0.00 0.00         1  0.0007   0.0007  apd_set_pprof_trace

10.2    0.00 0.00  0.00 0.00     0.00 0.00         3  0.0000   0.0000  trim

4.3     0.00 0.00  0.00 0.00     0.00 0.00         3  0.0000   0.0000  base64_encode

1.9     0.00 0.00  0.00 0.00     0.00 0.00         3  0.0000   0.0000  test

0.6     0.00 0.00  0.00 0.00     0.00 0.00         1  0.0000   0.0001  array_walk

0.6     0.00 0.00  0.00 0.00     0.00 0.00         1  0.0000   0.0008  main
```



Drive Tuning

- Hard-drive is in most cases the slowest part of the system, yet all the data eventually comes from it.
- By adjust the drive configuration parameters you can help your OS get the most out of it.



Drive Tuning Parameters

- Use the **`hdparm`** utility to adjust settings.
  - **`-c1`** - set IDE 32-bit I/O setting
  - **`-d1`** - enable DMA
  - **`-u1`** - enable IRQ unmasking
  - **`-m16`** - turn on multicount
  - **`-X 34|66|100|133`** - transfer mode

Validating Changes

Benchmark the affect of the changes using:

```
hdparm -tT /dev/[drive]
```

RAM Disk

- One way to accelerate File IO operations is by moving the files and directories to a RAM disk.
- On Linux this is extremely simple to do using via **tmpfs**.

```
# Speed Up /tmp Directory
mount --bind -ttmpfs /tmp /tmp

# Accelerate Scripts Directory
mount --bind -ttmpfs /home/webroot /home/webroot
```

Session Storage

- PHP's session extension by default stores each session inside a separate file.
  - Many files in one directory reduce access speed.
    - Assign each user their own session directory
    - Split sessions into multiple directories
      session.save_path = "N;/path"

Session Storage Alternatives

- File system is slow, lets use memory
  - **mm** - native shared memory storage
  - **apc** - use APC's store/fetch/delete
  - **memcache** - memory storage daemon

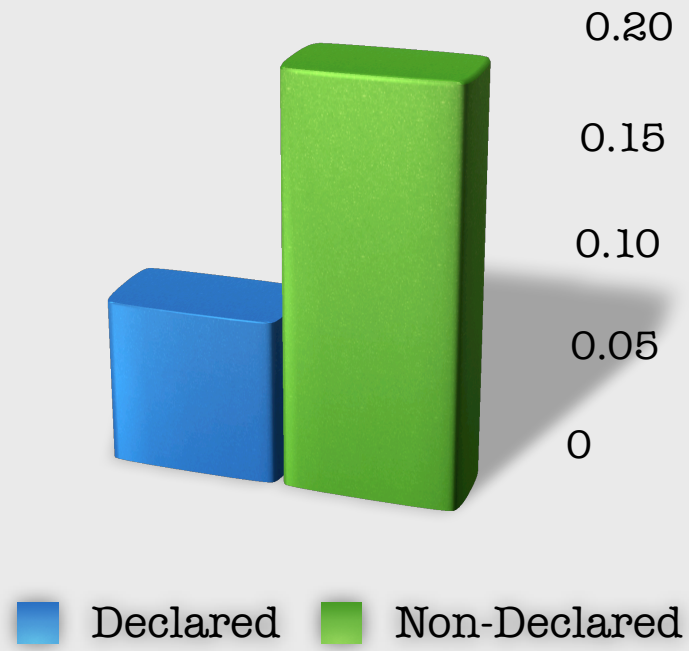
Now let's tune PHP code

OOP Tips

- Always declare your static methods!
  - Cleaner & Faster code

```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";
```

Speed Comparison



Use Class Constants

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

Avoid Magic

- Magic methods such as **__get()** / **__set()**
- Magic loading functions such as **__autoload()**
- Dynamic methods via **__call()**

require_once() is once too many

```
<?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
```

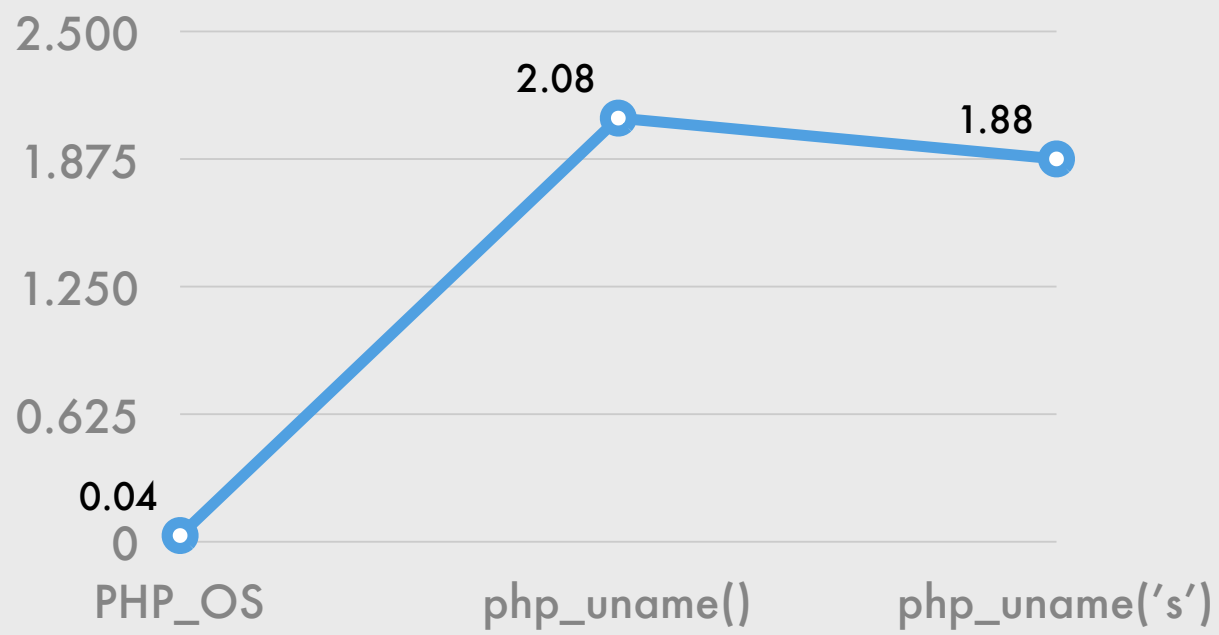
- If you absolutely cannot avoid require_once and include_once use full paths.

- In PHP 5.2>= this will allow PHP to avoid opening the file twice.

Avoid Pointless Function Calls

- php_version()
  - ✓ PHP_VERSION constant
- php_uname('s')
  - ✓ PHP_OS constant
- php_sapi_name()
  - ✓ PHP_SAPI constant

Quick Comparison



Fastest Win32 Detection in the West

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

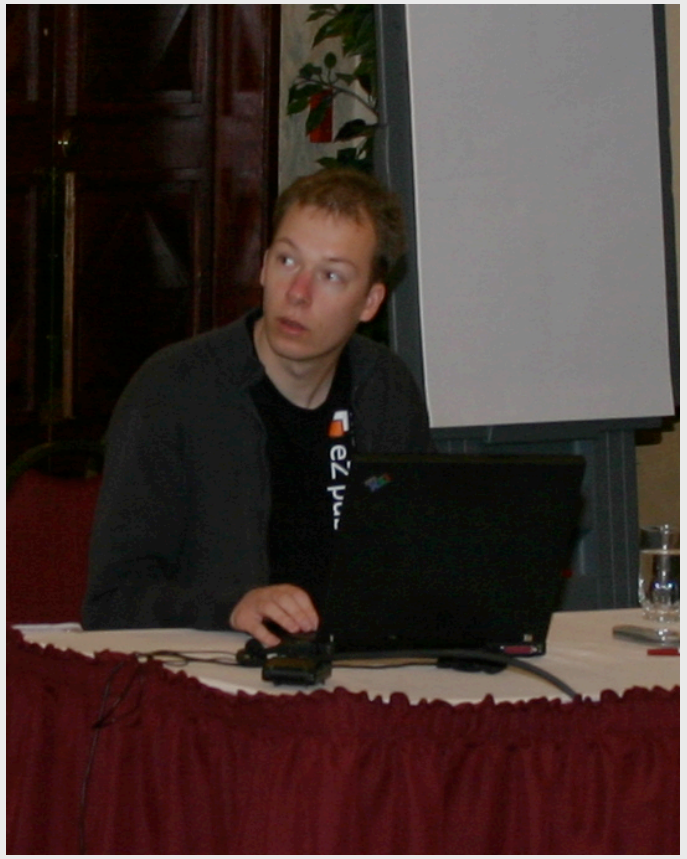
- Does not use functions
- Does not care about WinXP, WinNT, Windows, Windows98, NT 5.0, etc...
- Always available

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 Slowdowns

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

```
$text = preg_replace(
'"/(\n|\t|\r\n|\s)+/"', ' ', $origtext );
```

Use non-capturing patterns

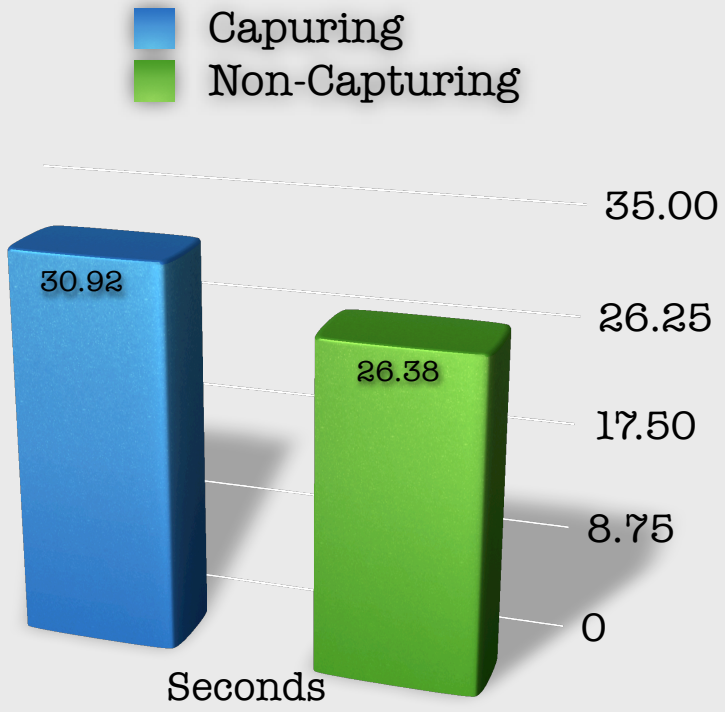
- Placing ?: at the start of a sub-pattern makes it non-capturing.

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

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

```
$text = preg_replace(
'"/(?:\n|\t|\r\n|\s)+/"', ' ', $origtext );
```

End Result



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

If Possible Avoid Regex

```php
<?php
// Slow
if (preg_match("!^foo_!i", "FoO_")) { }
// Much faster
if (!strncasecmp("foo_", "FoO_", 4)) { }

// Slow
if (preg_match("![a8f9]!", "sometext")) { }
// Faster
if (strpbrk("a8f9", "sometext")) { }

// Slow
if (preg_match("!string!i", "text")) {}
// Faster
if (stripos("text", "string") !== false) {}
```

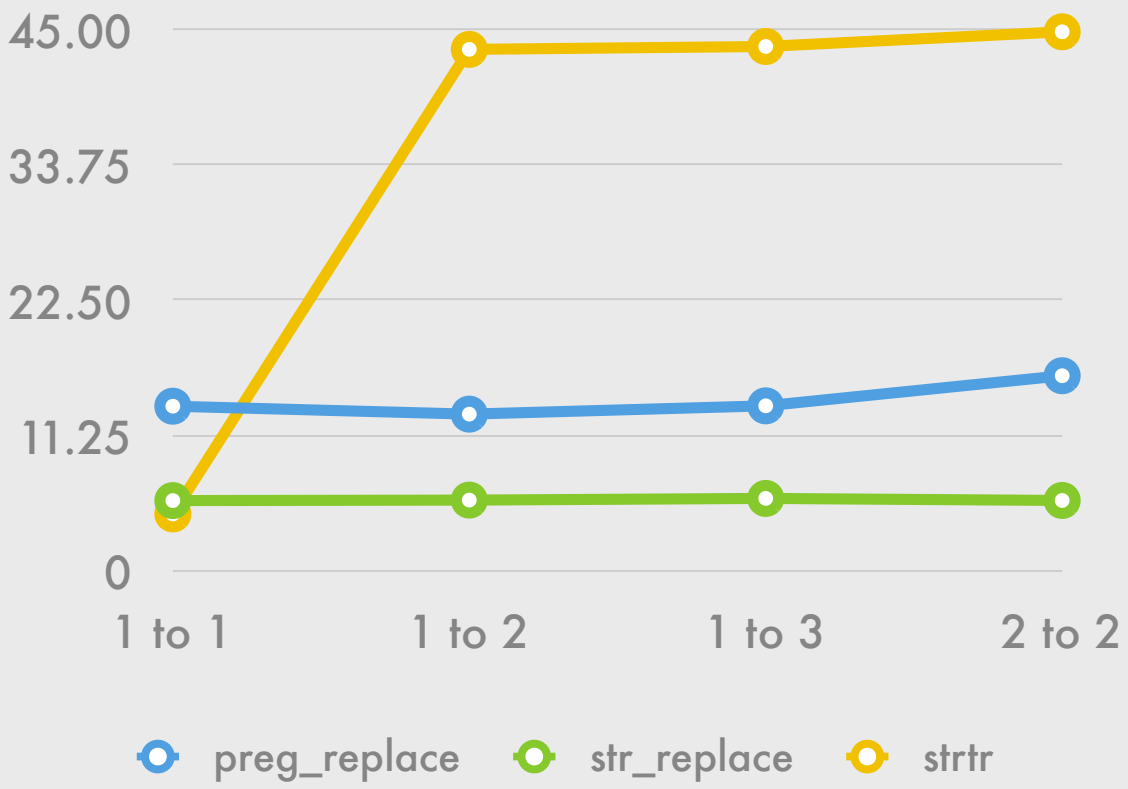
More Regex Avoidance

```
$text = preg_replace( "/\n/", "\\n", $text);
```

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

```
$text = str_replace( "/\n/", "\\n", $text);
```

Speed Comparison



Use strtr() Properly!

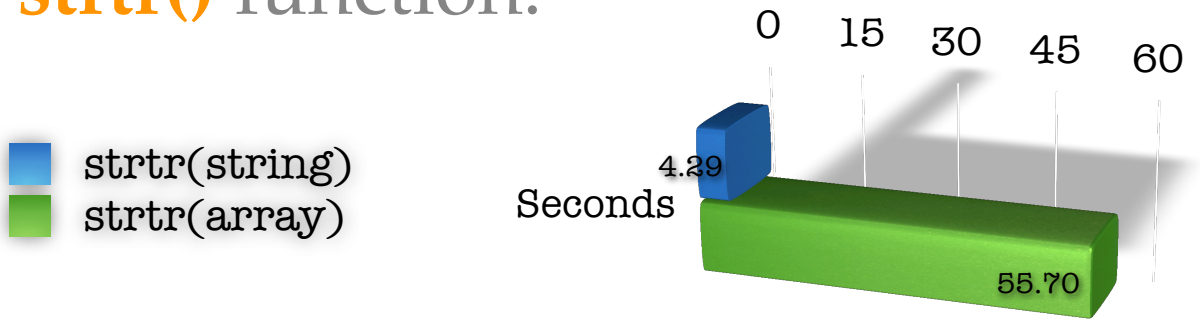
```
$rep = array( '-' => '*', '.' => '*' );

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

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

Use Strings!

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

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



Don't Replace When you

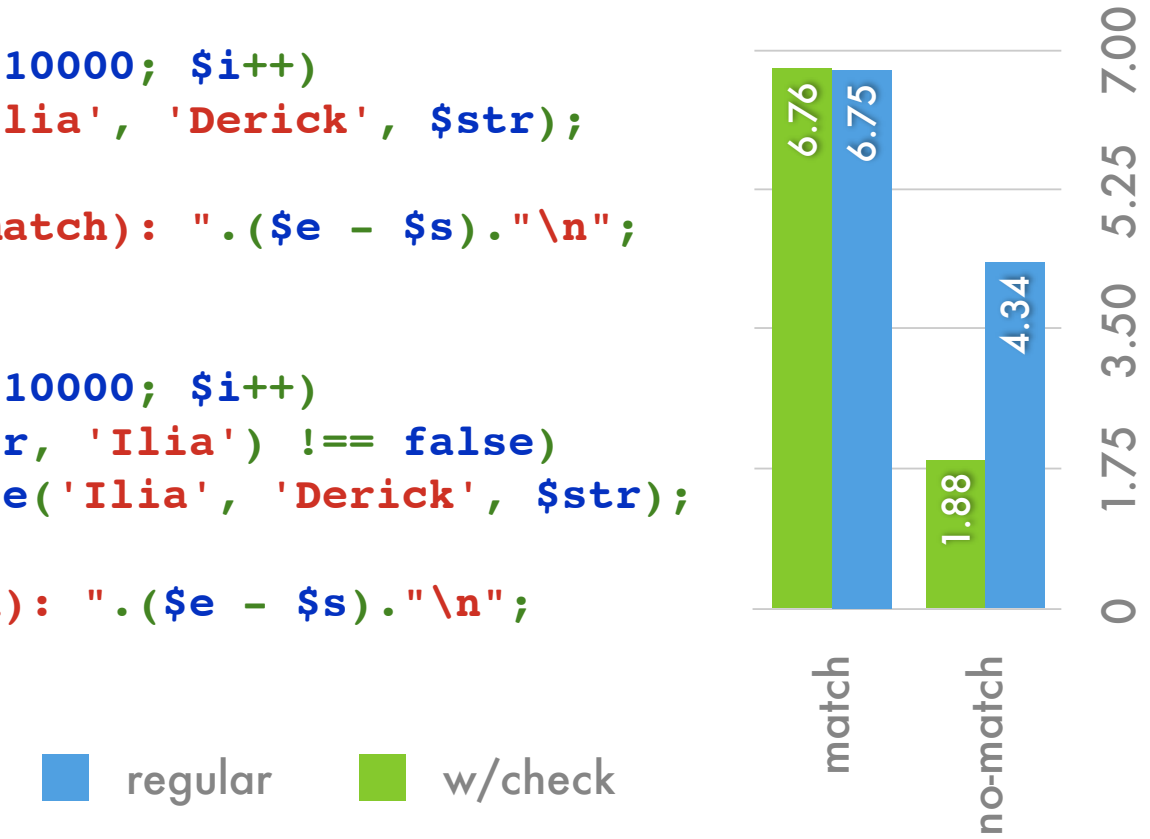
- Any replacement operation requires memory, if only to store the "modified" result.

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

Test Scenario

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

```php
$s = microtime(1);
for ($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 character in PHP's alphabet.

```
@action();
```

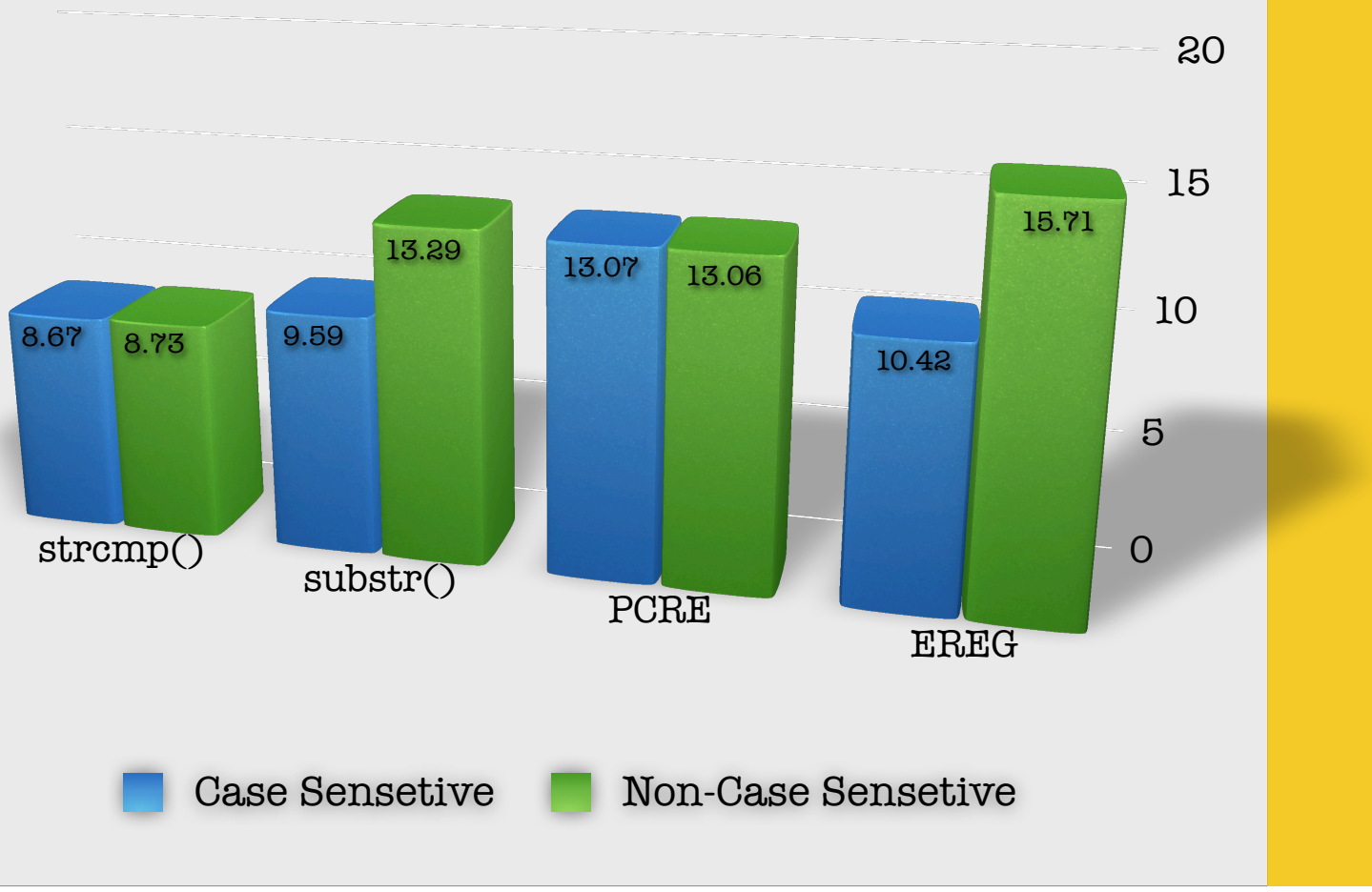
- This seemingly innocuous operator actually performs fairly intensive operations in the background.

```
$old = ini_set("error_reporting", 0);
action();
ini_set("error_reporting", $old);
```

Better String Comparison

```php
<?php
// The Good
if (!strncmp(PHP_OS, 'WIN', 3)) {
if (!strncasecmp(PHP_OS, 'WIN', 3)) {

// The Bad
if (substr(PHP_OS, 0, 3) == 'WIN') {
if (strtolower(substr(PHP_OS, 0, 3))) == 'win') {

// And The Ugly
if (preg_match('!^WIN!', PHP_OS)) {
if (preg_match('!^WIN!i', PHP_OS)) {
```

Quick Benchmark



Comparing 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) != 'text')

/* == */

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

Don't Mis-use Constants

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

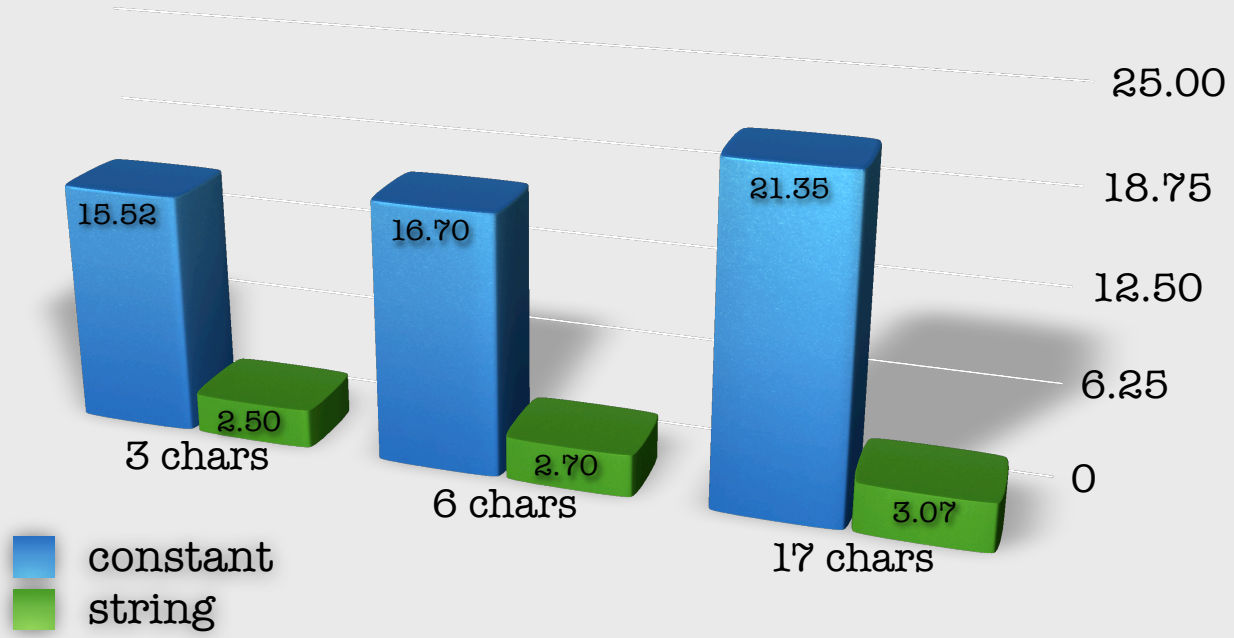
```
$foo = array("bar"=>0);
$foo[bar] = 1;
```

Why is this bad?

- 1 strtolower
- 2 hash lookups
- E_NOTICE error message generated
- temporary string being created on the fly.

Performance Check

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



700% difference on average!!!

Fix "harmless" error messages

- Each error results in:
  - Generation of a complex error string
  - Output to stdout/stderr
  - Potential write to a file or syslog
  - In pre-5.2 releases may leak memory in some rare instances.

Simplify for() loop

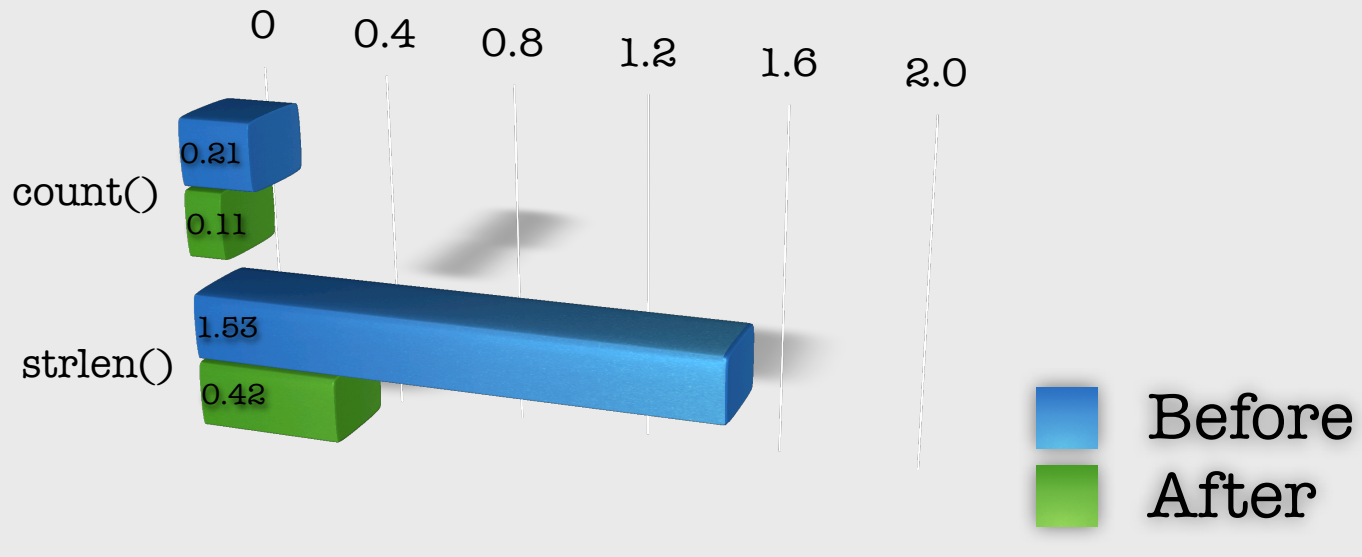
Avoid function calls within **for()** loop control blocks.

```
<?php
for ( $i = 1; $i < sizeof($array); $i++ ) {}

for ( $i = 0; $i < count($array); $i++ ) {}

for ( $i = 0; $i < strlen($string); $i++ ) {}
```

Otherwise function is called for every loop iteration.

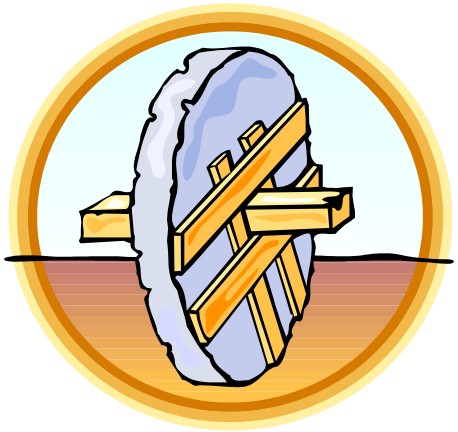
Speed Check

```
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++){}
```



Don't Re-invent the Wheel

- It is surprising how frequently people try to re-invent the wheel.
- Now a days PHP has
  - ✓ 2,700 functions
  - ✓ 80 core extensions
  - ✓ 154 PECL extensions



- Chances are what you need already exists!

Use Full File Paths

- While it is convenient(??) to do **require "foo.php"** and have it work, internally it leads to significant overhead.

- Whenever possible you should use full paths, that require no resolution in PHP.

The internals of file ops.

```
stat64("./b.php", {st_mode=S_IFREG|0644, st_size=6, ...}) = 0
getcwd("/tmp", 4096)                                    = 5
lstat64("/tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=18008, ...}) = 0
lstat64("/tmp/b.php", {st_mode=S_IFREG|0644, st_size=6, ...}) = 0
open("/tmp/b.php", O_RDONLY)
```

The issue can be further exasperated by the use of include_path.

Reference Tricks

References can be used to simply & accelerate access to multi-dimensional arrays.

```
$a['b']['c'] = array();
// slow 2 extra hash lookups per access
for($i = 0; $i < 5; $i++)
        $a['b']['c'][$i] = $i;

// much faster reference based approach
$ref =& $a['b']['c'];
for($i = 0; $i < 5; $i++)
        $ref[$i] = $i;
```

Optimization Myths

- Removing comments makes code faster
- Using ‘ is faster than “
- Passing things by-reference makes code faster
- Objects make code faster
- Ternary **? :** is faster then **if () { } else {}**

Caching is the recognition and exploitation of the fact that most "dynamic" data does not change every time you request it.

Caching Approaches

- Complete page content caching
- Content pre-generation
- On-Demand caching
- Partial page caching
- SQL query caching
- Browser caching

"Blog" Scenario

```
<?php
require "/blog.inc";
load_header();
$qry = mysql_query("SELECT  * FROM blog_entries
ORDER BY date DESC LIMIT 15");
while ($entry = mysql_fetch_assoc($qry)) {
  show_blog_entry($entry);
}
load_footer();
```

Cache Handler

```
function cache() {
  list(,$dt) = mysql_fetch_row(mysql_query("SELECT
  MAX(date) FROM blog_entries"));
  if (filemtime("/blog/cache/index.html") > $dt) {
        // cache hit
        readfile("/blog/cache/index.html");
        exit;
  }
  init_cache();
}
```

```
function init_cache() {
  ob_start();
  register_shutdown_function('write_cache');
}

function write_cache() {
  echo $data = ob_get_clean();
  $fname = "/blog/cache/index.html";
  $tmp_fname = tempnam("/blog/cache", "blog");
  file_put_contents($tmp_fname, $data);
  rename($tmp_fname, $fname);
}
```

Cache Placement

```
<?php
require "/blog.inc";
require "/cache.inc";
cache();
load_header();
$qry = mysql_query("SELECT  * FROM blog_entries
ORDER BY date DESC LIMIT 15");
while ($entry = mysql_fetch_assoc($qry)) {
  show_blog_entry($entry);
}
load_footer();
```

Quick benchmark

	Before	After
Page Size	31,117 bytes	31,117 bytes
Latency	8.73 msec.	4.62 msec.
Req./Second	451	665

Serendipity Test

	Before	After
Page Size	25,853 bytes	25,853 bytes
Latency	199.3 msec.	42.37 msec.
Req./Second	24.7	112.2

Compressed Page Cache

```
if (!empty($_SERVER["HTTP_ACCEPT_ENCODING"])
&& strpos($_SERVER["HTTP_ACCEPT_ENCODING"],
'gzip') !== false) {
  header('Content-Encoding: gzip');
  readfile("/tmp/index.html.gz");
} else {
  readfile(" /blog/cache/index.html");
}
```

```
function write_cache() {
  echo $data = ob_get_clean();
  $fname = "/blog/cache/index.html";
  $tmp_fname = tempnam("/blog/cache", "blog");
  file_put_contents($tmp_fname, $data);
  rename($tmp_fname, $fname);
  $tmp_name = tempnam("/blog/cache", "blog");
  copy("/tmp/index.html",
    "compress.zlib://" . $tmp_name);
  rename($tmp_name, "/tmp/index.html.gz");
}
```

Content Pre-generation

- Simplifies caching code
  - No real-time work
  - Entire cache can be easily rebuilt
  - Can be used to bypass PHP entirely
  - Predictable operation

Let's Avoid PHP

- Concept
  - Make product pages be .html pages
  - Generate .html pages via PHP
  - Static pages are served by the web server, skipping the PHP overhead.

```
function content_generate($s, $e)
{
  while ($s <= $e) {
     $url = "http://store.com/product.php?id=".$s;
     $file = "/cache/product_".$s;

     $data = file_get_contents($url);
     file_put_contents($file, $data);

     copy($file, "compress.zlib://" . $file . ".gz");
     $s++;
  }
}
```

Livedocs Test

	Before	After
Page Size	30,896 bytes	30,896 bytes
Latency	611.2 msec.	2.37 msec.
Req./Second	5.29	1284.6

Pre-Generation Problems

- Disk utilization
  - 2 files per-page X number of pages
- Generates pages no one may visit
- Generating all pages at once may take too long.

On Demand Caching

- Advantages
  - Generate what you need, when you need it.
  - Simple to setup
  - Has all the speed advantages of pre-generation.

404 Handler

- Concept
  - Make all links lead to .html pages
  - When page does not exist, have PHP 404 handler be triggered.
  - The handler script generates the missing page
  - All future visitors access static content.

ErrorDocument 404 /generate_html.php

```
// sample URL http://site.com/products/123.html

$id = (int) basename($_SERVER['REDIRECT_URL']);

if (!$id || $id < 0) { exit; }

$url = "http://store.com/product.php?id=".$id;
$file = "./products/".$id."".html";

$data = file_get_contents($url);
file_put_contents($file, $data, LOCK_EX);
copy($file, "compress.zlib://" . $file . ".gz");
```

CACHING QUANDARY

My pages always have an ever changing content and cannot be cached in their entirety.

*Dynamic Content

**Semi-Dynamic Content

APC to the Rescue!

Alternative PHP Cache (APC) is an open source opcode cache. It happens to provide handy PHP variable storage mechanism we can use.

APC API

- apc_store([key], [variable], [time to live])
  - key - cache entry identifier
  - variable - PHP variable to store
  - time to live - for long to store the data
- [variable] = apc_fetch([key])
- apc_delete([key])

```
if (!($promo = apc_fetch('promo'))) {
  $promo = fetch_all_promos();
  apc_store('promo', $promo, 3600); // 1 hour
}

if (!($deals = apc_fetch('deal'))) {
  $deals = fetch_all_deals();
  apc_store('promo', $promo, 600); // 10 mins
}

if (!($final_tmpl = apc_fetch('main_template'))) {
  $final_tmpl = generate_front_page();
  apc_store('main_template', $final_tmpl, 86400);
}                              // 1 day
```

```
// fetch one promo & deal to display
$data[0] = $promo[array_rand($promo)];
$data[1] = $deals[array_rand($deals)];
// get the always dynamic sections
$data[2] = get_shopping_cart();
$data[3] = generate_greeting_msg();
// in the finalized page substitute semi-static content
echo str_replace(
  array(
    '%%deals%%', '%%promo%%',
    '%%cart%%', '%%greeting%%'
  ), $data, $final_tmpl);
```

SQL Caching

- In most applications the primary bottleneck can often be traced to "database work".
- Caching of SQL can drastically reduce the load caused by unavoidable, complex queries.

Cache Tables

```
CREATE TABLE search_cache
(
  skey       CHAR(32) PRIMARY KEY,
  rec_id     INT NOT NULL,
  age        INT UNSIGNED,

  INDEX(skey),
  INDEX(age)
);
```

Basic Search Cache

```
$key = md5("user supplied search query");
$tm = time(); $max_age = $tm - 3600 * 4; // 4h

$qry = "SELECT count(*) FROM search_cache
WHERE key='{$key}' AND age>".$max_age;

if (!$pdo->query($qry)->fetchColumn()) {
  // cache generation code is here
}
// Standard join against search_cache table to get
// the relevant records
```

Cache Generation Code

```
// clear old cache entries
$pdo->exec("DELETE FROM search_cache
      WHERE key='{$key}'
      AND age<=".$max_age);

$qry  = "INSERT INTO search_cache";
$qry .= " (key,age,rec_id) ";
$qry .= " SELECT '{$key}', {$tm}, rec_id";
$qry .= " FROM [search qry] LIMIT 1000";

$pdo->exec($qry);
```

Browser Cache

- Advantages
  - Reduces data sent to virtually zero
  - Requires next to no server resources
- Disadvantages
  - Flimsy control over cache age
  - Not guaranteed to work

How to implement?

```
// Time to Cache
header("Expires: ".date(DATE_RFC822));

// Based on modification date
header("Last-Modified: ".date(DATE_RFC822));

// Content hash
header("Etag: ".md5_file("./page_cache"));
```

Most applications will end up using databases for information storage. Improper use of this resource can lead to significant and continually increasing performance loss.

Check Your Queries

```
EXPLAIN select * from users where login LIKE '%ilia%';
+----------+------+---------------+------+---------+------+-------+------------+
| table    | type | possible_keys | key  | key_len | ref  | rows  | Extra      |
+----------+------+---------------+------+---------+------+-------+------------+
| mm_users | ALL  | NULL          | NULL |    NULL | NULL | 27506 | where used |
+----------+------+---------------+------+---------+------+-------+------------+
```

Most databases offers tools for analyzing query execution.

```
EXPLAIN select * from users where login LIKE 'ilia%';
+----------+-------+---------------+-------+---------+------+------+------------+
| table    | type  | possible_keys | key   | key_len | ref  | rows | Extra      |
+----------+-------+---------------+-------+---------+------+------+------------+
| mm_users | range | login         | login |      50 | NULL |    2 | where used |
+----------+-------+---------------+-------+---------+------+------+------------+
```

Bitwise Option Packing

Rather then creating a column for every Boolean option, you can pack 32 of them into a single integer field.

```
CREATE TABLE users (
        is_active INT,
        is_banned INT,
        is_admin INT,
        ...
);
```

```
CREATE TABLE users (
        user_opt INT,
        ...
);

user_opt & 1 // active
user_opt & 2 // banned
user_opt & 4 // admin
```

KISS = Performance

- The simpler the code, the faster it runs, it really is that simple.
  - Syntactic sugar.
  - Unnecessary wrappers.
  - Wrapping one liners in functions.
  - OO for the sake of OO.

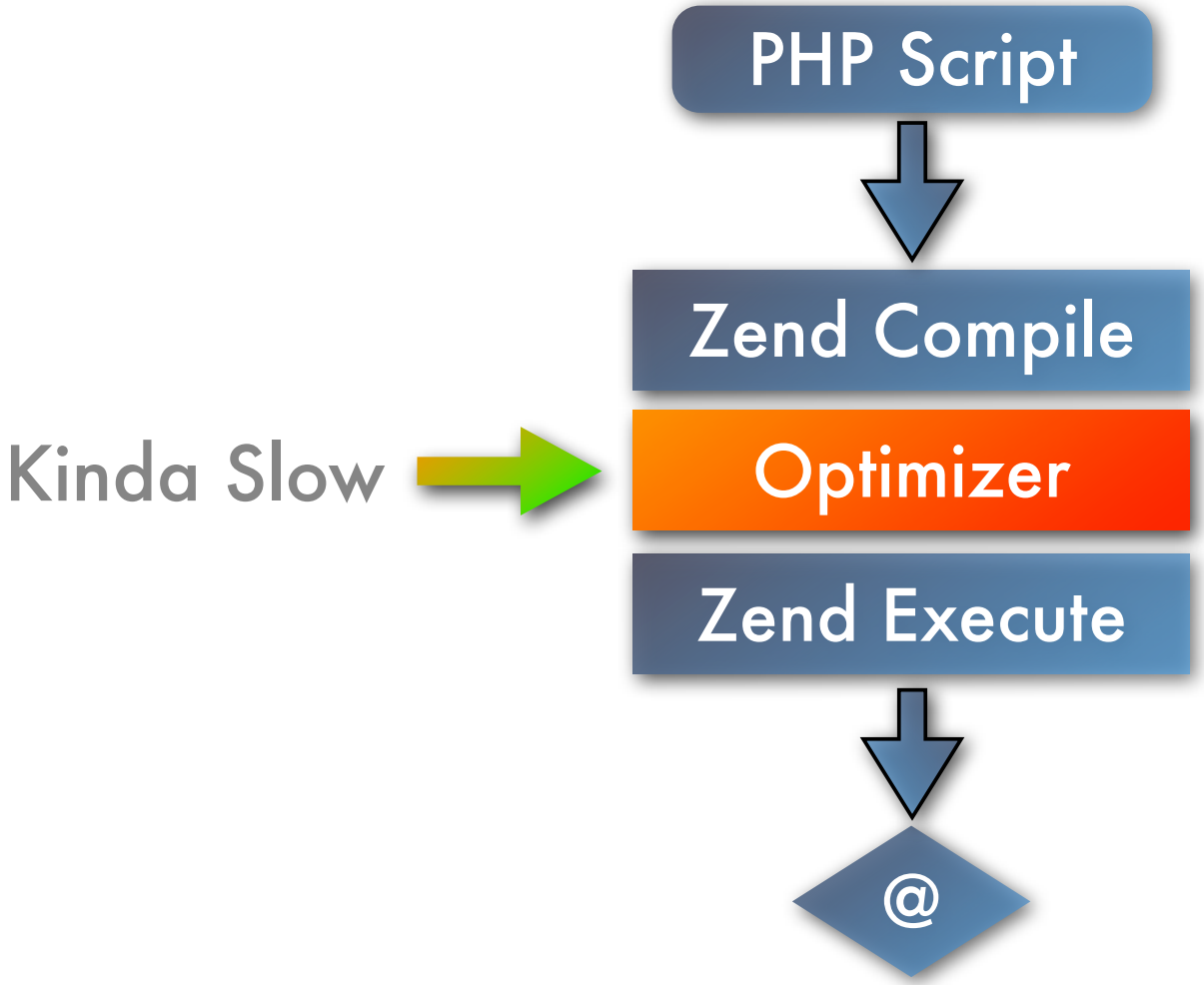
Thank You For Listening!

- These slides
  - http://www.ilia.ws/
- APC
  - http://pecl.php.net/apc
- XDebug
  - http://www.xdebug.org/

Optimizer (Why?)

- The opcodes generated by Zend Engine are often inefficient.
- Some operations can be avoided
- A lot of temporary vars are not necessary.
- Every compiler needs an optimizer ;-)

Optimizer (How?)



What Can It Do?

- opt. heredoc
- print to echo
- GLOBALS[foo] to foo
- inline known constants
- eliminate NOP
- resolve partial file paths.
- optionally inline define() calls
- 60+ function calls with static values resolved.
- Much more...

Any other ideas?