High Performance PHP

George Schlossnagle
<george@omniti.com>

What is Performance?

- Ensuring that your application works correctly within required time constraints.

Factors Affecting Performance

- Webserver Setup
- Language Optimization
- Database Setup
- Data Setup
- Network Setup
- Application Design
- Application Implementation

Language Optimization

- Number one improvement you can make is using a compiler cache
  - Apc
  - Phpa
  - Zend Cache
- Code optimizer
  - Zend Optimizer
  - ???

Database/Data Setup

- Databases need their own tuning
  - Server Setup
  - Smart schema design
  - Building indexes/structuring queries to be efficient
- Hire a DBA, or become a DBA

Application Design

- Macro-Level Tuning
- Questions:
  - Does the application do what we want and no more?
  - Are there design changes that can improve performance without sacrificing necessary features?

Application Design

- Examples
  - Application-integrated caching
  - Reducing feature bloat

Application Implementation

- Micro-Level Tuning
- Identifying and addressing performance bottlenecks in specific code sections

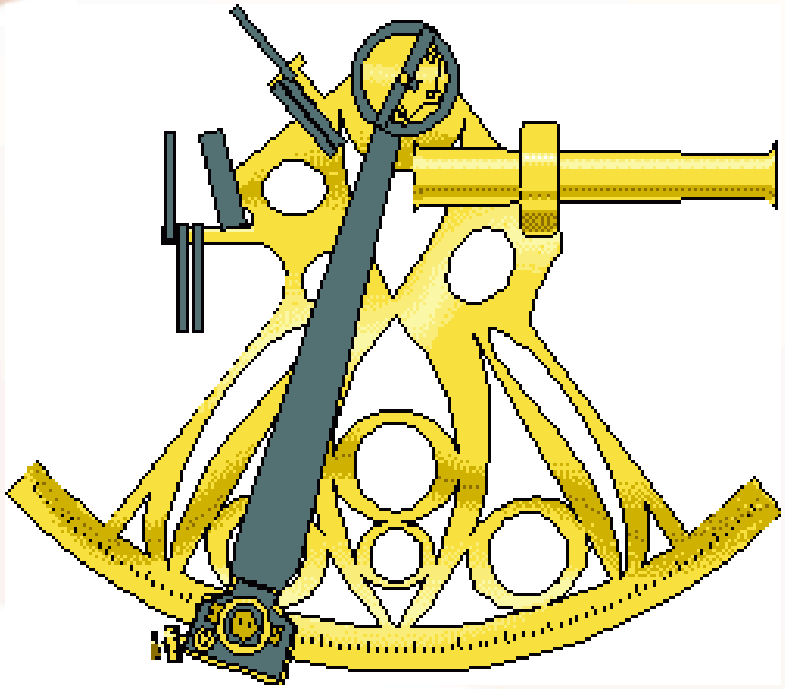
Roadmap

- Set Goals
- Profile
- Tune
- Evaluate

Setting Goals

- So many things to measure

Profiling



Profiling is about knowing where you are and where you are going.

Why Profile?

- Finding bottlenecks in a large code-base can be difficult.
- Quantitative methods are less subject to personal bias.
- To set metrics for success.

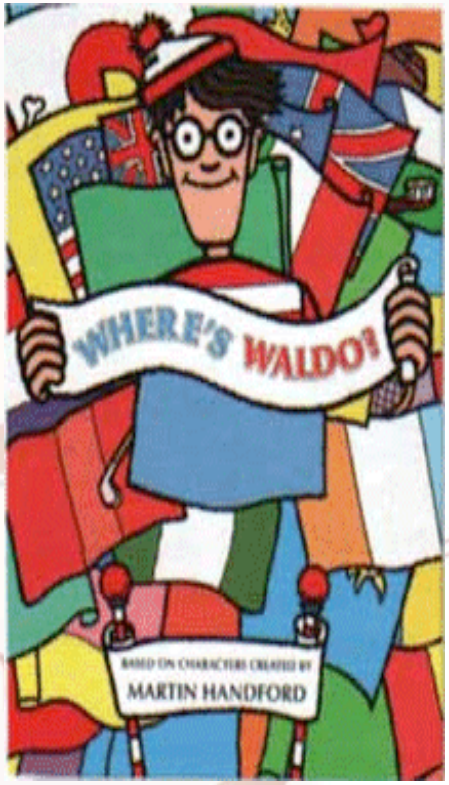
Finding Troublesome Code

- SLA Driven

  Political concerns dictate performance needs for particular scripts. This makes goal-setting easy, although problems can lie outside the symptom.
- Small Volume

  If you have fewer than 10 frequently accessed scripts/pages, or if you're troubleshooting a standalone app, skip these steps and profile all the code.
- Raw Resource Usage Driven

  A relatively fast but frequently executed script can be a bottleneck. If we have a significant number of candidates, we need to know where to focus our efforts.



Resource Measurement Method #1

Utilize Psychic Abilities

(or an innate understanding of the application)

Resource Measurement Method #2

A Bit of Science

Total Resource Utilization =

(cost per access) * (number of accesses)

Resource Measurement Attempt #1
Measuring With Apache's mod_log_config

Add "%T" to your apache logs:

```
LogFormat "%h %l %u %t \"%r\" %>s %b %T" profile
```

- Quick and Dirty
- Seconds-only accuracy is trouble, especially if most scripts average less than a second

Hot-Rod Your mod_log_config

We can add low-cost, high-resolution timers to mod_log_config.

- Apache is easy to hack
- This also allows for exposure of potential problems outside PHP
- Much less information to process than full traces

PHP-Userspace Profilers

- Benchmark_Profiler (PEAR)

Profilers that sit in the ZendEngine

- DBG (http://dd.cron.ru/dbg/)
- APD (PECL)

Generating Profiles with APD

Download from PECL

Or

Get the latest from PECL via anon cvs

```
> cd apd
> phpize; configure --enable-apd=shared
> sudo make install
```

In php.ini add

This automatically should install the 'php_apd' zend module into your

```
<PHP INSTALL PATH>/lib/php/<ZEND VERSION><-OPTIONAL_DEBUG>/
```

directory. It isn't mandatory to have it there, in fact you can install it anywhere you care. In your INI file, add the following lines:

```
zend_extension = /absolute/path/to/php_apd.so
apd.dumpdir = /absolute/path/to/trace/directory
```

Reading APD Profiles

```
pprofp <flags> <trace file>
    Sort options
    -a          Sort by alphabetic names of subroutines.
    -l          Sort by number of calls to subroutines
    -r          Sort by real time spent in subroutines.
    -R          Sort by real time spent in subroutines (inclusive of child calls).
    -s          Sort by system time spent in subroutines.
    -S          Sort by sys. time spent in subroutines (inclusive of child calls).
    -u          Sort by user time spent in subroutines.
    -U          Sort by user time spent in subroutines (inclusive of child calls).
    -v          Sort by average amount of time spent in subroutines.
    -z          Sort by user+system time spent in subroutines. (default)

    Display options
    -c          Display Real time elapsed alongside call tree.
    -i          Suppress reporting for php builtin functions
    -O <cnt>    Specifies maximum number of subroutines to display. (default 15)
    -t          Display compressed call tree.
    -T          Display uncompressed call tree.
```

Super-Simple Example

```
<?
    apd_set_pprof_trace();
    function my_print($var) {
          print $var;
    }
    function hello($var) {
          $var = ucfirst(strtolower($var);
          return “Hello $var\n”;
    }
    my_print(hello(“George”));
?>
```

Profile

```
[~/bench/php]> pprofp -T /tmp/pprof.25044
main
hello
  strtolower
  ucfirst
my_print

Trace for hello.php
Total Elapsed Time =    0.01
Total System Time  =    0.00
Total User Time    =    0.00


         Real         User        System              secs/    cumm
%Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls    call    s/call Name
--------------------------------------------------------------------
  0.0  0.01  0.01   0.00  0.00   0.00  0.00     1   0.0000    0.00 my_print
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.00 ucfirst
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.00 strtolower
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.00 hello
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.00 main
```

Float Like a Butterfly Sting Like a Bee

Profiling fast running code – Build a testing harness

omniti

Where to Profile

- In Development
  - ✓ Easiest
  - ✓ 'Real' data often not available
- On live production code
  - ✓ Often logistically difficult
  - ✓ Can uncover problems not exposed elsewhere due to data differences/concurrency levels

Example: Seven Dirty Words

Problem: A personal-page application which stores content in a database and filters their content for potentially malicious javascript is misbehaving in production.

Example: Seven Dirty Words

- First we profile the page in our test environment:

```
 Real          User        System              secs/    cumm
%Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls    call    s/call Name
-----------------------------------------------------------------------
 30.0   0.02  0.02    0.04  0.04    0.02  0.02    390    0.0002     0.00 define
 15.0   0.00  0.00    0.03  0.03    0.00  0.00     12    0.0025     0.00 ociexecute
 10.0   0.04  0.21    0.01  0.16    0.01  0.03     37    0.0005     0.01 include
  5.0   0.00  0.00    0.01  0.01    0.00  0.00     89    0.0001     0.00 ocifetch
  5.0   0.00  0.00    0.01  0.01    0.00  0.00     50    0.0002     0.00 bar
```

Example: Seven Dirty Words

Comments:

- 12 db calls seems excessive, we may be experiencing a db bottleneck
- Overall execution is fast, we should look at code running agaisnt 'live' data

Example: Seven Dirty Words

Same profile run in production:

```
       Real          User         System              secs/    cumm
%Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls    call     s/call Name
-----------------------------------------------------------------------
 99.9 20.02 20.02   9.13  9.13   0.01  0.01    54   1.1693     1.16 preg_match
  0.0  0.00  0.00   0.10  0.10   0.00  0.00    12   0.0083     0.00 ociexecute
  0.0  0.02  0.02   0.04  0.04   0.02  0.02   390   0.0002     0.00 define
  0.0  0.00  0.00   0.02  0.02   0.00  0.00    94   0.0002     0.00 ocifetch
  0.0  0.04  0.21   0.01  0.16   0.01  0.03    37   0.0005     0.01 include
```

WOAH! A quick look at the call tree shows us

```
…
preg_match    /data/code/php/util/filters.php:326 time:(166,0,505)
…
```

Example: Seven Dirty Words

This code block is:

```
function detectUnsafeHTML($html)
   {
        global $UNSAFE_HTML;
        foreach ($UNSAFE_HTML as $rule) {
            if(preg_match($rule, $html)) {
                return -1;
            }
        }
        return 0;
   }
```

And `$UNSAFE_HTML` is an array of 'unsafe' tags and or profanity that we disallow. Since we loop through the list, Since they are processed in order, we can even spot the slow ones in the loop.

Example: Seven Dirty Words

```
…
$UNSAFE_HTML[] = !<.*[^a-z]onload\s*=!is";
$UNSAFE_HTML[] = !<.*[^a-z]onunload\s*=!is";
$UNSAFE_HTML[] = !<.*[^a-z]onerror\s*=!is";
…
```

Gee, who added these? Each runs in O(N*M), where N is the number of '<'s and M is the number of '`onerror`'s, etc. Since we can't guarantee that these are even inside tags this way, we may as well change them to

```
…
$UNSAFE_HTML[] = !onload\s*=!is";
$UNSAFE_HTML[] = !onunload\s*=!is";
$UNSAFE_HTML[] = !onerror\s*=!is";
…
```

Example: Seven Dirty Words

Same profile run in production:

```
       Real          User        System              secs/    cumm
%Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls    call     s/call Name
-----------------------------------------------------------------------
  0.0   0.00  0.00   0.11  0.10   0.00  0.00     12   0.0083     0.00 ociexecute
  0.0   0.02  0.02   0.04  0.04   0.02  0.02    390   0.0002     0.00 define
  0.0   0.00  0.00   0.02  0.02   0.00  0.00     94   0.0002     0.00 ocifetch
  0.0   0.04  0.21   0.01  0.16   0.01  0.03     37   0.0005     0.01 include
  0.0   0.01  0.01   0.02  0.02   0.02  0.02     54   0.0003     0.00 preg_match
```

Much better!

Bad regular expressions often only show their true colors under certain data, so be careful. Remember forewarned is forearmed.

Interpreting Profiles

- Profiling is often viewed as black magic

Tuning

Benchmarking

Using Benchmark_Iterate

Use Builtins/Extension Functions When Available

- Builtin functions are significantly faster

Max Pain

```
include('Benchmark/Iterate.php');
function my_max($array) {
    $max = $array[1];
    foreach( $array as $el ) {
        if ( $el > $max ) {
            $max = $el;
        }
    }
    return $max;
}

$benchmark = new Benchmark_Iterate;
foreach (array(10, 100, 1000) as $size) {
  foreach (array('max') as $func ) {
    $array = gen_array($size);
    $benchmark->run(1000, $func, $array);
    $result = $benchmark->get();
    print "$func run on datasize $size: ".$result['mean']."\n";
  }
}
```

Max Pain Results

```
>php max.php
max run on datasize 10: 7.12444782257E-05
my_max run on datasize 10: 0.000111220359802

max run on datasize 100: 8.70881080627E-05
my_max run on datasize 100: 0.000425333499908

max run on datasize 1000: 0.000373518824577
my_max run on datasize 1000: 0.0121217674017
```

The built-in function is between 2 and 30 times faster, with increasing returns as our data set grows.

What is the Matrix

Multi-Dimensional Array Slices

```
function get_slice_naive($array)
{
  $retarr = array();
  foreach( $array as $el ) {
    $retarr[] = $el[0];
  }
  return $retarr;
}
```

```
function get_slice_reset($array)
{
  return array_map("reset”,$array);
}
```

Looping Constructs

- When using builtin functions, there is a substantial benefit to using builtin looping functions like array_map and array_walk
- Otherwise, most methods are about equal

Use your database wisely

- Avoid expensive SQL
- Only select the information you want, especially from 'wide tables'
- Strike a balance between application-side processing and database-side processing

Example: pagination (v1)

```
Function display_page($pageNumber, $itemsPerPage)
{
   $conn = ociconn_wrapper();
   $query = (“SELECT * from guestbook order by date”);
   $sth = Ociparse($conn, $query);
   Ociexecute($sth);
   While(ocifetch($sth) && $I < ($page_number *
   $itemsPerPage )) { $I++}
   While(ocifetchinto($sth, $result) && $itemsPerPage--) {
        $retval[] = $results;
   }
   ocifinish($sth);
   return $retval;;
}
```

Example: pagination (v2)

```
Function display_page($pageNumber, $itemsPerPage)
{
   $conn = ociconn_wrapper();
   $query = (“select * from
        (select a.*, a.rownum rowid from
               (SELECT * from guestbook order by date)
        ) where rowid between :begin and :end”);
   $sth = Ociparse($conn, $query);
   $begin = $pageNumber * $itemsPerPage;
   $end = $begin += $itemsPerPage;
   ocibindbyname($sth, “:begin”. &$begin, -1);
   ocibindbyname($sth, “:end”, &$end, -1);
   Ociexecute($sth);
   ocifetchintostatement($sth, $retval);
   ocifinish($sth);
   return $retval;
}
```

Example: Pagination (v3+)

- If we are building out an 'index' page without the message bodies (just subjects and senders), we may want to avoid the network/memory load of selecting the bodies out.
- Storing page numbers in the database (can be extremely expensive if delete load is high)
- Using a filesystem/shared-memory cache (if operating in a cluster, may require a distributed cache deletion method)

Thanks for the Memories

Memory is cheap but not free

Know your Zend Engine

How does all this profiling stuff work

Two Straightforward Methods

- Use begin_func_call/end_func_call hooks
- Wrap zend_execute/zend_execute_internal

Where To Go From Here

- If other tuning methods are not successful, we can recode portions of the app in C.

The Fast and the Frustrated
Recoding critical functions in C

Benefits:

- Speed

Costs:

- Harder to maintain
- Less portable

Inline_C

Benefits:

- As fast as an extension (almost)
- Somewhat easier to maintain

Costs:

- Security
- Flexibility

A Simple Example

```
<?
require_once("Inline_C.php");
$function1 = <<<EOF
PHP_FUNCTION(times)
{
    long i,j;
    if (zend_parse_parameters(ZEND_NUM_ARGS()  TSRMLS_CC, "ll",
   &i,&j) == FAILURE)
        return;
    RETURN_LONG(i*j);
}

EOF;

$inline = new Inline_C;
$inline->add_code($function1);
$inline->compile();
for($i=0;$i<10;$i++) {
    for($j=0; $j<10; $j++) {
        print "$i * $j = ".times($i,$j)."\n";
    }
}

?>
```

A Less Simple Example
(implementing rc4)

- Use pear's Crypt_Rc4
  - In pear.
  - Easy to use.
  - Not extremely fast.
- implement in C
  - Faster
  - More maintenance requirements

A Less Simple Example
(implementing rc4)

```
$helper = <<EOF
#define swap_byte(x,y) t = *(x); *(x) = *(y);
    *(y) = t
void _rc4(unsigned char *buffer, int buflen,
    unsigned char *key, int k_len)
{
  unsigned char x =0, y = 0, index1 = 0, index2
    = 0, i, j;
  unsigned char k[256], s[256];
  unsigned char xorIndex, t;
  char digit[5];
  char seed[256];
  short counter;
  int n;

  if(k_len&1) {
    k_len--;
    key[k_len] = '\0';
  }
  k_len /= 2;
  strcpy(digit, "AA");
  digit[4]='\0';
  for(i=0;i<k_len;i++)
  {
    digit[2] = key[1*2];
    digit[3] = key[i*2+1];
    sscanf(digit,"%x",&seed[i]);
  }
  for(counter = 0; counter < 256; counter++)
    s[counter] = counter;
  for(counter = 0; counter < 256; counter++)
  {
    index2 = (k[index1] + s[counter] + index2) %
     256;
    swap_byte(&s[counter], &s[index2]);
    index1 = (index1 + 1) % k_len;
  }
  for(counter = 0; counter < buflen; counter++)
  {
    x = (x + 1) % 256;
    y = (s[x] + y) % 256;
    swap_byte(&s[x], &s[y]);
    xorIndex = (s[x] + s[y]) % 256;
    buffer[counter] ^= s[xorIndex];
  }
}

EOF;
```

A Less Simple Example
(implementing rc4)

```
$function = <<EOF
PHP_FUNCTION(rc4)
{
    char *data, *key;
    int datalen, keylen;
    if (zend_parse_parameters(ZEND_NUM_ARGS()  TSRMLS_CC, "ss", &data,
   &datalen, &key, &keylen) == FAILURE)
        return;
     _rc4(data, datalen, key, keylen);
     RETURN_STRINGL(data, datalen, 1);

}

EOF;
```

A Less Simple Example
(implementing rc4)

```
$inline = new Inline_C;
$inline->add_code($helper);
$inline->add_code($function);
$inline->compile();

$encode = rc4("hello world", "d3adb33f");
$plaintext =    rc4($encode, "d3adb33f");
print "Plaintext: $plaintext\n";
```

SUMMARY

- THINK SMART!

  Sound application (re)design can be your biggest win.

- LOOK SMART!

  Use profiling to focus your efforts on troublesome code blocks.

- BE SMART!

  C is fast, but carries maintenance overhead. Make sure you enter these waters with prudence.

Thank You!