Do you PHP?

Apachecon 2003

November 19, 2003. Las Vegas

Rasmus Lerdorf <rasmus@php.net>

http://lerdorf.com/ap2003.pdf

Many Things

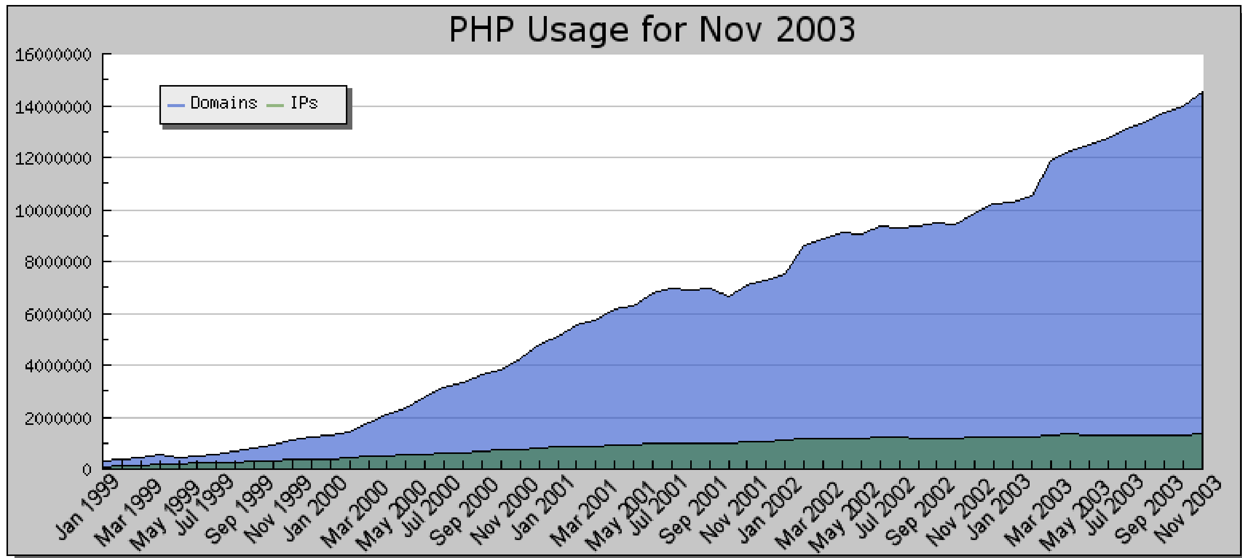
PHP is Many Things to Many People

- The BASIC of the Web
- Web Template System
- General-purpose Scripting Language
- Advanced Application Framework
- Application Server?

PHP is Big!

November 2003 Netcraft Report

- 44,946,965 Domains queried
- 14,528,748 Domains. 1,328,604 IP addresses
- PHP installed on 32% of all domains



Source: Netcraft

November 2003 Apache Module Report

- 12,220,278 Apache Servers surveyed
- 4,311,776 (52.04%) PHP
- 2,542,854 (30.69%) OpenSSL
- 2,465,141 (29.75%) mod_ssl
- 1,825,732 (22.03%) Frontpage
- 1,607,948 (19.41%) mod_perl
- 406,032 (4.90%) DAV
- 368,240 (4.44%) mod_throttle
- 330,237 (3.99%) mod_jk
- 319,777 (3.86%) mod_log_bytes
- 314,140 (3.79%) mod_bwlimited
- 249,816 (3.01%) mod_fastcgi
- 216,022 (2.61%) AuthMySQL

Source: SecuritySpace.com

Yahoo! is Big!

March 2003 Numbers

- 232 million visitors per month
- 112 million active registered users
- 2.9 million fee paying customers
- 1.9 billion average daily page requests

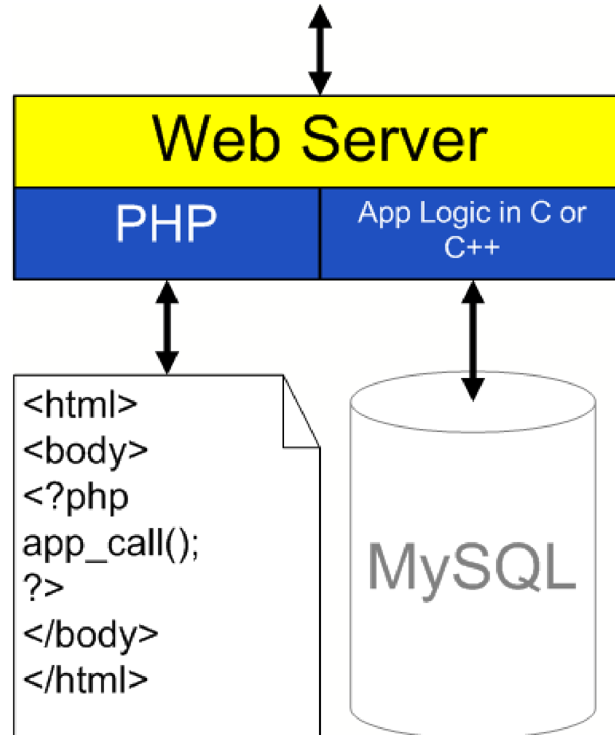
Top 25 Global Web Destinations

Property Name	Unique Audience	Rank	Active Reach %	Time Per Person
Yahoo!	120,493,514	1	48.39	01:23:11
MSN	115,588,212	2	46.42	00:50:27
AOL Time Warner	92,436,459	3	37.13	00:30:46
Microsoft	82,629,769	4	33.19	00:08:59
Google	81,443,445	5	32.71	00:17:17
eBay	49,312,550	6	19.81	01:31:47
Amazon	44,230,686	7	17.76	00:12:31
Lycos Network	44,223,690	8	17.76	00:17:01
About-Primedia	33,293,092	9	13.37	00:09:28
Wanadoo	21,587,999	10	08.67	00:23:12
CNET Networks	20,668,081	11	08.30	00:07:42
Sony	20,625,555	12	08.28	00:13:08
Walt Disney Internet Group	19,601,633	13	07.87	00:15:59
T-Online	19,037,641	14	07.65	01:07:08
Sharman Networks	17,793,954	15	07.15	00:08:31
Vivendi Universal	16,924,890	16	06.80	00:11:05
InfoSpace	16,374,618	17	06.58	00:08:13
The Gator Corporation	16,197,754	18	06.51	00:22:17
Real Networks	16,170,655	19	06.49	00:11:35
Rakuten	15,229,300	20	06.12	00:44:33
Tiscali	15,169,633	21	06.09	00:11:44
Viacom International	14,695,554	22	05.90	00:14:31
Nifty	13,896,271	23	05.58	00:27:05
AT&T	13,574,390	24	05.45	00:24:36
eUniverse Network	12,945,636	25	05.20	00:10:43

Source: Nielsen//NetRatings

The PHP Way

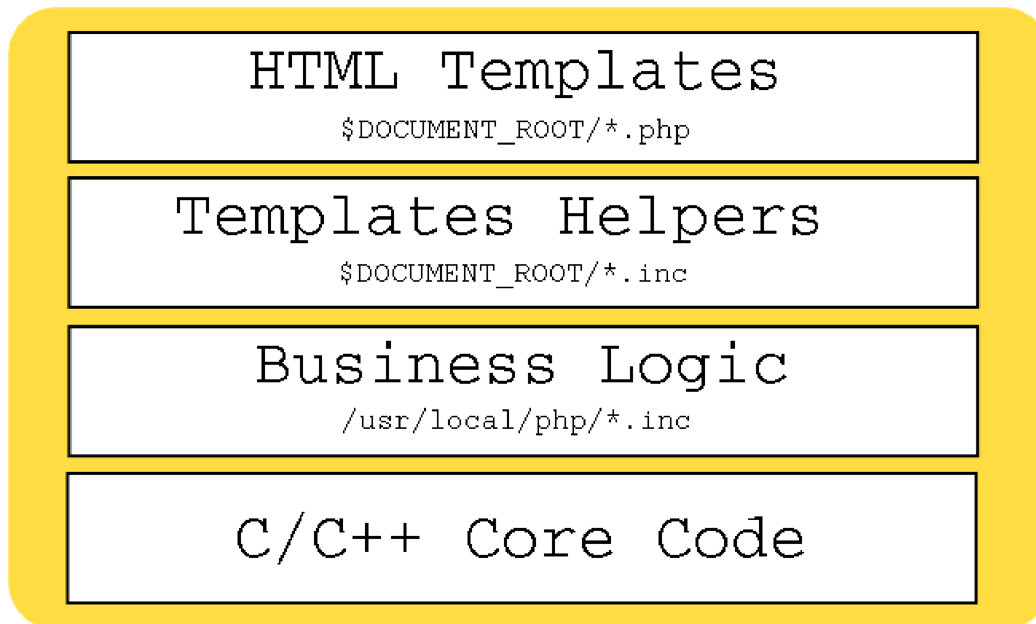
PHP was created as a framework for writing web applications in C or C++ and making it easy to expose the business logic of these applications to a powerful presentation-layer templating language.



Most people don't really use PHP this way. Over the years the templating language improved both in scope and performance to the point where entire web apps could be written in it.

App Architecture

A suggested architecture for a PHP application. The template layer should have as little business logic as possible. As you go down you have less presentation and more business logic.



Simplistic Example

In terms of a real-world example of what goes in these 4 different layers, assume we are writing a database-backed application that needs to fetch a user record containing various fields. This is a very common thing to do in a web app. Our template layer might look something like this:

php.ini

```
auto_prepend_file = "./helpers.inc"
include_path = "/usr/local/lib/php"
```

Template Layer

```
<?title('Example Page')?>
<?greeting()?>
<h1>Heading 1</h1>
  <p>
    Page content
  </p>
<h1>Heading 2</h1>
  <p>
    Yada Yada
  </p>
<h1>Heading 3</h1>
  <p>
    Page content
  </p>
<?footer()?>
```

Template Helpers

```
<?php
  include "logic.inc";
  echo '<?xml version="1.0" encoding="UTF-8"?>';
?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<?php
  $user = get_user_record($_COOKIE['user_id']);
  function greeting() {
    global $user;
    echo "Hi " .$user['first_name'] ."!<br />\n";
    if($age=birthday($user)) {
      echo "Congratulations, today is your ";
      echo "$age birthday!<br />\n";
    }
  }
  function title($title) {
    echo "<head><title>$title</title></head>\n";
    echo "<body>\n";
  }
  function footer() {
    echo "</body>\n</html>";
  }
?>
```

Business Logic

```
<?php
```

```
  function get_user_record($id) {
    mysql_connect('localhost');
    mysql_select_db('users');
    $res = mysql_query("select * from users where id='$id'");
    if(!$res) echo mysql_error();
    else $row = mysql_fetch_array($res);
    return $row;
  }
  function birthday($user) {
    if(strftime('m d')==strftime('m d',$user['bday']))
      $age = strftime('Y') - strftime('Y',$user['bday']);
      if(($age100)>10 && ($age100)<20) $ap='th';
      else switch($age%10) {
        case 1: $ap = 'st'; break;
        case 2: $ap = 'nd'; break;
        case 3: $ap = 'rd'; break;
        default:  $ap = 'th'; break;
      }
      return $age.$ap;
    else
      return false;
  }
?>
```

In this case the final layer written in C contains the mysql_* functions, and the date() function. These happen to be standard PHP functions. If birthday() is called many times on every single request and since how you figure out if it is someone's birthday is unlikely to change very often, this may be a good candidate to translate into C. Although, in this example, the birthday function is probably too simple to see much of a performance improvement. On the other hand, other than a little bit of added parameter parsing, if you compare the C version of birthday() to the PHP version, it isn't that much harder to write it in C.

C Layer

```
PHP_FUNCTION(birthday)
{
  time_t timestamp, now;
  struct tm ta1, tmbuf1, ta2, tmbuf2;
  int age;
  char ret_age[8];

  if (zend_parse_parameters(1 TSRMLS_CC, "l", &timestamp) == FAILURE)
      return;

  ta1 = php_localtime_r(&timestamp, &tmbuf1);
  time(&now);
  ta2 = php_localtime_r(&now, &tmbuf2);

  if(tmbuf1.tm_mday==tmbuf2.tm_mday && tmbuf1.tm_mon==tmbuf2.tm_mon) {
      age = tmbuf2.tm_year - tmbuf1.tm_year;
      if((age100)>10 && (age100)<19) sprintf(ret_age,"%dth",age);
      else switch(age % 10) {
          case 1: sprintf(ret_age,"%dst",age); break;
          case 2: sprintf(ret_age,"%dnd",age); break;
          case 3: sprintf(ret_age,"%drd",age); break;
          default:sprintf(ret_age,"%dth",age); break;
      }
  } else {
      RETURN_FALSE;
  }
  RETURN_STRING(ret_age,1);
}
```

Input Filtering

Security in a web application boils down to always checking any user-supplied input data.

Exploits

- readfile($filename)
- system($cmd)
- file uploads into document_root
- XSS - Cross Site Scripting hacks

Input Filter hook

```
PHP_MINIT_FUNCTION(my_input_filter)
{
    sapi_register_input_filter(my_sapi_input_filter);
    return SUCCESS;
}
```

For a complete example, see README.input_filter in the PHP 5 source distribution. For PHP4, you will have to patch your source with http://lerdorf.com/input_filter.txt

Making it all talk to each other

- C
- C++
- Shared Libraries
- ELF
- runtime linker

No Function-scoped Static Objects

```
void example() {
   static std::string str;
}
```

gcc 3.x has -fuse-cxa-atexit and your libc needs to support __cxa_atexit() in order to solve this problem. Other than that, if you can't change the code, patch PHP to not dlclose() the shared extensions on shutdown to avoid the crash. See http://lerdorf.com/dlclose.txt for the patch.

Throwing C++ exceptions

If a C++ shared library is loaded by a C runtime, like PHP, you could end up with weird __throw() problems. An annoying workaround is to link your C runtime against libstdc++.so to prevent the linker from pulling in the __throw symbol from libgcc.a.

Tuning

Some simple guidelines

- Try to limit yourself to 5 or less includes per request
- Don't go overboard on OOP. Use where appropriate.
- Same goes for Layers. Abstraction, Indirection, abstract classes.
- Everything has a cost
- Use an opcode cache
- Watch your regular expressions!
- Cache! Cache! Cache!
- If you have plenty of CPU but limited bandwidth, turn on output compression

Benchmarking

Let's have a look at how we might benchmark and subsequently tune a PHP server. We will use two machines. A client machine to run our HTTP load program (http_load from acme.com) and a P4 1.7G server with 256M of RAM. We will also be using 3 freely available opcode caches:

http://cvs.php.net/cvs.php/pear/PECL/apc

http://www.turckware.ru/en/e_mmc.htm

http://www.ioncube.com

Don't blow your io buffers!

```
#  To change this on Linux, cat a larger number
#  into /proc/sys/net/core/wmem_max in your
#  httpd startup script
SendBufferSize 65535
```

This is our benchmark script

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head><title>Simple PHP Benchmark</title></head>
<body>
<h1>String Manipulation</h1>
<p>
<?php
$str = 'This is just a silly benchmark that doesn\'t do anything useful.';
$str .= 'Here we are just uppercasing the first two characters of every word ';
$str .= 'in this long string';
$parts = explode(' ',$str);
foreach($parts as $part) {
    $new[] = strtoupper(substr($part,0,2)).substr($part,2);
}
echo implode(' ',$new);
?>
</p>
<h1>Including 2 files</h1>
<p>
<?php
include './bench_include1.inc';
include './bench_include2.inc';
?>
</p>
<h1>for-loop and calling a function many times</h1>
<p>
<?php
$a = range(1,200);
$b = range(200,1);
for($i=0; $i<200; $i++) {
    echo foo($a[$i],$b[$i]);
}
?>
</p>
<h1>Define and Instantiate an object and call some methods</h2>
<p>
<?php
class test_class {
    var $test_var1;
    var $test_var2;
    var $test_var3;
    var $test_var4;
```

```
    function test_class($arg1, $arg2) {
        echo "Constructor args: $arg1, $arg2<br />\n";
    }


    function set_var1($value) {
        $this->test_var1 = $value;
        echo "test_var1 property set to $value<br />\n";
        return true;
    }
    function set_var2($value) {
        $this->test_var2 = $value;
        echo "test_var2 property set to $value<br />\n";
        return true;
    }
    function set_var3($value) {
        $this->test_var3 = $value;
        echo "test_var3 property set to $value<br />\n";
        return true;
    }
    function set_var4($value) {
        $this->test_var4 = $value;
        echo "test_var4 property set to $value<br />\n";
        return true;
    }
    function disp() {
        echo "<pre>";
        var_dump($this);
        echo "</pre>";
    }
}
$obj = new test_class(1,'arg2');
$obj->set_var1('test1');
$obj->set_var2(123);
$obj->set_var3($a);  / Array from previous test /
$obj->set_var4(array(1,2,3,4,5,6,7,8,9));
$obj->disp();
?>
</p>
</body>
</html>
```

bench_include1.inc

```
<?php
function foo($arg1, $arg2) {
    if($arg1>$arg2) return $arg1;
    elseif($arg1<$arg2) return $arg2;
    else return 'xxx';
}
?>
```

bench_include2.inc

```
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
```

```
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
This is just a bunch of plain text in an include file.<br />
```

An http_load run

```
2500 fetches, 5 max parallel, 1.962e+07 bytes, in 15.9486 seconds
7848 mean bytes/connection
156.753 fetches/sec, 1.2302e+06 bytes/sec
msecs/connect: 1.40331 mean, 2999.44 max, 0.171 min
msecs/first-response: 29.4603 mean, 648.731 max, 6.005 min
HTTP response codes:
  code 200 -- 2500
```

Whenever you do any sort of load testing, you need look beyond just the raw numbers and have a look at what your server is actually doing. Use vmstat:

Base PHP Load

```
  procs      memory      page                    disks     faults      cpu
  r b w     avm    fre  flt  re  pi  po  fr  sr ad0 ac0   in   sy  cs us sy id
 25 0 0  149472  17572   26   0   0   0   0   0   0   0 1663 12390 4391 80 20  0
  5 0 0  149472  17532   27   0   0   0   0   0   0   0 1665 12322 4444 77 23  0
  7 0 0  149472  17492   24   0   0   0   0   0   2   0 1657 12409 4771 74 26  0
  5 0 0  149472  17452   28   0   0   0   0   0   0   0 1687 12520 4856 82 18  0
 25 0 0  149472  17412   28   0   0   0   0   0   0   0 1649 12413 4756 78 22  0
  5 0 0  149472  17372   23   0   0   0   0   0   0   0 1645 12199 4417 77 23  0
  5 0 0  149084  17332   27   0   0   0   0   0   1   0 1679 12564 4385 76 24  0
  5 0 0  149464  17304   26   0   0   0   3   0   1   0 1663 12336 4551 79 21  0
  5 0 0  149464  17264   27   0   0   0   0   0   0   0 1662 12480 4663 82 18  0
```

Load with APC cache

```
  procs      memory      page                    disks     faults      cpu
  r b w     avm    fre  flt  re  pi  po  fr  sr ad0 ac0   in   sy  cs us sy id
  5 0 0  150140  11936   29   0   0   0   0   0   0   0 1629 12203 4687 75 25  0
  5 0 0  150140  11888   28   0   0   0   0   0   0   0 1619 12007 4579 79 21  0
  5 0 0  150140  11848   28   0   0   0   0   0   0   0 1640 12305 4252 76 24  0
  5 0 0  150140  11808   25   0   0   0   0   0   1   0 1630 12006 4628 84 16  0
 25 0 0  150140  11776   27   0   0   0   0   0   0   0 1635 12304 4118 88 12  0
  5 0 0  150140  11736   26   0   0   0   0   0   0   0 1627 12059 4381 80 20  0
  5 0 0  150528  11696   22   0   0   0   0   0  11   0 1635 12178 4685 83 17  0
  5 0 0  150528  11656   26   0   0   0   0   0   0   0 1624 12027 4651 84 16  0
```

Load with IonCube cache

```
  procs      memory      page                    disks     faults      cpu
  r b w     avm    fre  flt  re  pi  po  fr  sr ad0 ac0   in   sy  cs us sy id
  5 0 0  127996  14516   28   0   0   0   0   0   0   0 1608 19343 4382 77 23  0
  5 0 0  127996  14476   25   0   0   0   0   0   0   0 1601 19416 4628 82 18  0
  8 0 0  127616  14436   23   0   0   0   0   0   1   0 1600 19181 4604 76 24  0
 26 0 0  127616  14400   24   0   0   0   1   0   1   0 1605 19439 3940 84 16  0
  5 0 0  127616  14360   26   0   0   0   0   0   0   0 1612 19416 4582 79 21  0
  5 0 0  128956  14328   25   0   0   0   0   0   1   0 1599 19336 4559 72 28  0
  6 0 0  128956  14288   26   0   0   0   0   0   0   0 1598 19390 4444 79 21  0
```

Load with Turck MMCache

```
  procs      memory      page                    disks     faults      cpu
```

```
 r b w     avm    fre  flt  re  pi  po  fr  sr ad0 ac0   in   sy   cs us sy id
 3 0 0  136696  13284   29   0   0   0   0   0   0   0 1694 14846 4748 79 21  0
 5 0 0  136696  13244   33   0   0   0   0   0   1   0 1711 14815 4472 74 26  0
 5 0 0  136308  13204   28   0   0   0   0   0   0   0 1696 14824 4771 83 17  1
 5 0 0  136688  13164   27   0   0   0   0   0   0   0 1692 14590 4880 83 17  0
 6 0 0  136688  13116   26   0   0   0   0   0   1   0 1687 14762 4010 83 17  0
 5 0 0  135696  13004   25   0   0   0   0   0   0   0 1693 14759 4840 79 21  0
 7 0 0  135696  12956   26   0   0   0   0   0   1   0 1676 14477 4643 74 25  1
```

Our benchmark test was deliberately designed to have quite a bit of PHP processing and not a whole lot of output. 7k is somewhat small for a web page. If instead we have a whole lot of output, chances are we will be io-bound instead of cpu-bound. If you are io-bound, there is little sense in optimizing at the PHP level.

Evidence of an io-bound test

```
 procs      memory      page                          disks     faults      cpu
 r b w     avm    fre  flt  re  pi  po  fr  sr ad0 ac0   in   sy   cs us sy id
 4 0 0  132860  15724 1033   0   0   0   0   0   0   0 4457  954 3704  2 25 74
 5 0 0  132860  15724 1009   0   0   0   0   0   0   0 4436  714 3597  3 24 73
 6 0 0  132860  15716  980   0   0   0   0   0   0   0 4446  925 3603  5 23 72
 2 0 0  132860  15716 1028   0   0   0   0   0   6   0 4514  720 3696  2 24 73
 3 0 0  132472  15716 1018   0   0   0   0   0   2   0 4501  946 3673  2 22 76
 4 0 0  132472  15716 1039   0   0   0   0   0   0   0 4565  737 3718  2 26 73
 3 0 0  132472  15708 1010   0   0   0   0   0   2   0 4498  938 3639  2 24 75
 2 0 0  132472  15708 1012   0   0   0   0   0   0   0 4543  730 3665  5 25 70
```

Things to try if you are io-bound

```
[php.ini]
output_handler = ob_gzhandler
```

```
[httpd.conf]
LoadModule gzip_module lib/apache/mod_gzip.so
```

Benchmarking Results

So, we have determined we are cpu-bound and we need to go faster. What can we do? Some low-hanging fruit:

include_path

```
include_path = "/usr/share/pear:."

<?php
    include './template_helpers.inc';
    include 'business_logic.inc';
?>
```

open_basedir

```
open_basedir = "/usr/share/htdocs/:/usr/share/pear/"
```

open_basedir checking adds some extra syscalls to every file operation. It can be useful, but it is rather expensive, so turn it off if you don't think you need it.

variables_order

```
variables_order = "GPC"

<?php
    echo $_SERVER['DOCUMENT_ROOT'];
    echo getenv('DOCUMENT_ROOT');
?>
```

If you never use cookies, turn those off too

Add an Opcode Cache

```
zend_extension=/usr/local/lib/php/20020429/apc.so
apc.enabled=1
apc.shm_segments=1
apc.optimization=0
apc.shm_size=30
apc.num_files_hint=10
apc.gc_ttl=10
apc.mmap_file_mask=/tmp/apc.XXXXXX
apc.filters=
```

Results

156 request/sec	Standard PHP 4.3
161 requests/sec	Fix include_path, and only populate GP
191 requests/sec	Add IonCube Accelerator on top
196 requests/sec	APC no optimizer, IPC shared mem + semaphore locking
198 requests/sec	Turck MMCache no optimizer with spinlocks
200 requests/sec	APC no optimizer, mmap mem + user space sleep locks

202 requests/sec	Turck MMCache with optimizer and spinlocks

Why Profile?

Because your assumptions of how things work behind the scenes are not always correct. By profiling your code you can identify where the bottlenecks are quantitatively.

How?

PECL to the rescue!

```
www:~> pear install apd
downloading apd-0.4p1.tgz ...
...done: 39,605 bytes
16 source files, building
running: phpize
PHP Api Version         : 20020918
Zend Module Api No      : 20020429
Zend Extension Api No   : 20021010
building in /var/tmp/pear-build-root/apd-0.4p1
running: /tmp/tmprFlAqf/apd-0.4p1/configure
running: make
apd.so copied to /tmp/tmprFlAqf/apd-0.4p1/apd.so
install ok: apd 0.4p1
```

Then in your php.ini file:

```
zend_extension = "/usr/local/lib/php/apd.so"
apd.dumpdir = /tmp
```

It isn't completely transparent. You need to tell the profiler when to start profiling. At the top of a script you want to profile, add this call:

```
<?php
apd_set_pprof_trace();
?>
```

The use the command-line tool called pprofp:

```
wwww: ~> pprofp
pprofp <flags> <trace file>
    Sort options
    -a          Sort by alphabetic names of subroutines.
    -l          Sort by number of calls to subroutines
    -m          Sort by memory used in a function call.
    -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 system 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.


% pprofp -z /tmp/pprof.48478
```

```
Trace for /home/y/share/htdocs/bench_main.php
Total Elapsed Time =    0.01
Total System Time  =    0.01
Total User Time    =    0.01


         Real         User         System              secs/    cumm
%Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls    call    s/call  Memory Usage
Name
--------------------------------------------------------------------------------------
100.0  0.00  0.00   0.01  0.01   0.01  0.01   200   0.0001    0.0001            0
foo
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
test_class->set_var2
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
test_class->set_var1
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
test_class->set_var3
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
test_class->set_var4
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
var_dump
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
test_class->disp
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
test_class->test_class
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
main
  0.0  0.00  0.00   0.00  0.00   0.00  0.00    26   0.0000    0.0000            0
strtoupper
  0.0  0.00  0.00   0.00  0.00   0.00  0.00    52   0.0000    0.0000            0
substr
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
implode
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     2   0.0000    0.0000            0
include
  0.0  0.01  0.01   0.00  0.00   0.00  0.00     2   0.0000    0.0000            0
range
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            0
explode


Trace for /home/rasmus/phpweb/index.php
Total Elapsed Time =    0.69
Total System Time  =    0.01
Total User Time    =    0.08

         Real         User         System              secs/    cumm
%Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls    call    s/call  Memory Usage
Name
--------------------------------------------------------------------------------------
 33.3  0.11  0.13   0.02  0.03   0.01  0.01     7   0.0043    0.0057       298336
require_once
 22.2  0.02  0.02   0.02  0.02   0.00  0.00   183   0.0001    0.0001       -33944
feof
 11.1  0.01  0.01   0.01  0.01   0.00  0.00     3   0.0033    0.0033       -14808
define
 11.1  0.04  0.04   0.01  0.01   0.00  0.00   182   0.0001    0.0001       112040
fgetcsv
 11.1  0.25  0.25   0.01  0.01   0.00  0.00     6   0.0017    0.0017         3768
getimagesize
 11.1  0.01  0.01   0.01  0.01   0.00  0.00    55   0.0002    0.0002         2568
sprintf
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     7   0.0000    0.0000         -136
printf
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000          136
htmlspecialchars
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000          -16
mirror_provider_url
```

```
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     7   0.0000    0.0000           112
spacer
  0.0  0.00  0.00   0.00  0.00   0.00  0.00    10   0.0000    0.0000          -552
delim
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           112
mirror_provider
  0.0  0.00  0.00   0.00  0.00   0.00  0.00    20   0.0000    0.0000          -624
print_link
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000            24
have_stats
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           -72
make_submit
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     2   0.0000    0.0000           112
strrchr
  0.0  0.08  0.08   0.00  0.00   0.00  0.00     2   0.0000    0.0000           168
filesize
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           -16
commonfooter
  0.0  0.00  0.11   0.00  0.00   0.00  0.00     2   0.0000    0.0000             0
download_link
  0.0  0.00  0.25   0.00  0.01   0.00  0.00     6   0.0000    0.0017           208
make_image
```

Stop!

One of the most important aspects of performance tuning and even programming in general is knowing when to stop. Getting your web site or product into the hands of the customer is after all the goal of all this. If you sit around and fiddle with it for months, nobody is using it.

Once you have located all the low-hanging fruit, further optimization tends to get exponentially harder for less and less gain. The time you spend on it compared to simply buying another server or two is usually not worth it. Stop and move onto the next project.

Or better yet, go home, spend some time away from these beastly computers and spend some time with your wife and/or kids

Index

Many Things 2
PHP is Big! 3
Yahoo! is Big! 4
The PHP Way 5
App Architecture 6
Simplistic Example 7
Input Filtering 9
Integrating PHP 10
Tuning 11
Benchmarking 12
Benchmarking Results 15
Benchmarking Results 17
Profiling PHP 19
Stop! 22