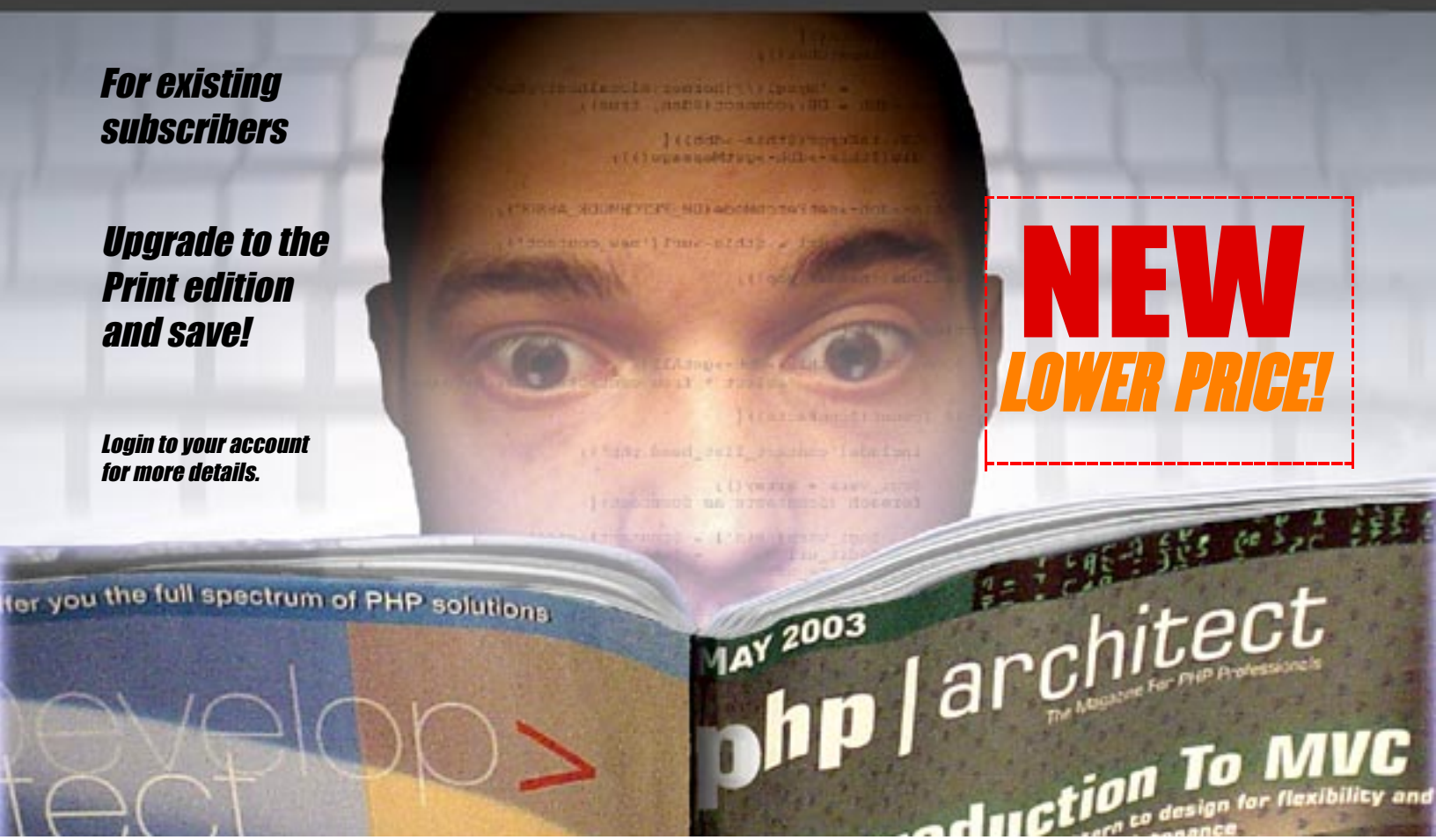
You'll never know what we'll come up with next



php | architect
The Magazine For PHP Professionals

Visit: ***http://www.phparch.com/print*** for more information or to subscribe online.

php|architect Subscription Dept.
P.O. Box 54526
1771 Avenue Road
Toronto, ON M5M 4N5
Canada

Your charge will appear under the name "Marco Tabini & Associates, Inc." Please allow up to 4 to 6 weeks for your subscription to be established and your first issue to be mailed to you.

*US Pricing is approximate and for illustration purposes only.

NEW LOWER PRICE!

Name: _____

Address: _____

City: _____

State/Province: _____

ZIP/Postal Code: _____

Country: _____

Payment type:

VISA	Mastercard	American Express
☐	☐	☐

Credit Card Number: _____

Expiration Date: _____

E-mail address: _____

Phone Number: _____

Choose a Subscription type:

☐	Canada/USA	$ 77.99 CAD	($59.99 US*)
☐	International Air	$105.19 CAD	($80.89 US*)
☐	Combo edition add-on (print + PDF edition)	$ 14.00 CAD	($10.00 US)

Signature: **Date:**

*By signing this order form, you agree that we will charge your account in Canadian dollars for the "CAD" amounts indicated above. Because of fluctuations in the exchange rates, the actual amount charged in your currency on your credit card statement may vary slightly.

To subscribe via snail mail - please detach/copy this form, fill it out and mail to the address above or fax to +1-416-630-5057

FEATURE

Debugging Questions and Xdebug Answers

by Derick Rethans

Despite its relatively young age, Xdebug is a very popular extension for PHP 4 and 5 that adds a great set of functions to the standard interpreter to debug, profile and analyze PHP scripts. In this article, I will provide a Q&A session for PHP developers who want to know how Xdebug can help make solving debugging, profiling and testing problems easier thanks to its advanced features. Here we go...

Why do I Need a Debugger?

Q: I have heard many times that a debugger is a required tool to debug an application, but I don't really agree as there already some functions in PHP to help you with debugging, such as `echo()` and `print_r()`.

A: I beg to differ and will try to convince you during the rest of this Q&A session that those two functions are nowhere close to the tools you need to debug a complex PHP application.

As this article is about debugging, most of the answers below will make use of functionality provided by Xdebug, an Open Source (BSD-style) licensed debugger for PHP.

Most of the examples are written on an Unix-like Operating System, but they should work as well on a Windows platform, except that they might require some changes in path and/or file names. The examples were all run with PHP 5.0.2-dev and Xdebug 2.0.0-beta1, but should work on any PHP version after 4.3.0—although the output may differ slightly. Go ahead with your questions!

How do I Install Xdebug?

Q: Before I ask questions about a few problems that I have while debugging applications, I would like to know how I can install Xdebug, as I assume that for most of the examples in this article to work I will need to have it up and running.

A: That is correct. If you're on a Unix-like system I would recommend to get the source code from the Xdebug website (*xdebug.org*) and follow the instructions on the Xdebug documentation page (*xdebug.org/install.php*). This should not take more than three minutes if you have a correctly set-up PHP installation. Depending on your distribution, you might need to install a **php-dev** package, or something like that, in order to get all the required files you need for a successful compilation of a PHP extension. Debian users can install the "php-dev" package through apt-get, for example. If you still fail installing Xdebug, your distribution might not provide a "correctly" set-up PHP installation and your best shot is to ask on the *php-general@lists.xdebug.org* mailing list.

For installation on a Windows system, please follow the instructions for precompiled modules on the installation documentation page—these are very straightforward, as you don't have to compile the extension at all.

Xdebug is also available through the Windows PECL snapshots for PHP 5.0.x (*snaps.php.net/win32/PECL_5_0/php_xdebug.dll*) and for PHP 5.1.x (*snaps.php.net/win32/PECL_UNSTABLE/php_xdebug.dll*) and the PECL distribution system (just execute **pear install xdebug** from a shell). Make sure you install at least version 2.0.0beta1.

REQUIREMENTS

PHP: 4.3.0+ (4.3.6+ recommended) or 5.0.0+
OS: Any
Other software: Xebug 2.0 beta 1 or higher, KCacheGrind recommended
Code Directory: xdebug

Why Does PHP Crash?

Q: Sometimes I get no output in the browser for a script and, when I look at the Apache error log I see something like "notice child pid 1048 exit signal Segmentation fault (11)". I know this is not a good thing, but how can I find out what happened?

A: When a crash occurs in a PHP script, people are always ready to blame a bug in the PHP interpreter, but, most of the time, that is not really the case. There is one occasion in which PHP is "allowed" to crash, and that is when a script has an infinite recursion loop in its code and an overflow occurs. PHP does not protect against those errors itself because it is always very hard to determine how many levels can be executed before the stack—a limited sized memory structure belonging to a program—overflows and PHP crashes. Without any clue as to where in a complex script this might be, the simple debugging functions—echo() and print_r()—will not be of much help in determining the cause of the crash.

Xdebug does protect against this infinite recursion problem by aborting the script when you reach 100 nested function calls. A nested function call is defined as a function calling another function—Listing 1 shows an example of this. It is, of course, possible to change this default limit of 100 to something else if your scripts so require; you can do that with the xdebug.max_nesting_level php.ini setting. When the configured maximum nesting level is reached, Xdebug aborts the script with the message "Fatal error: Maximum function nesting level of '3' reached, aborting! in <file> on line <linenr>," followed by a stack trace. This is a lot more useful than just crashing your web server.

Listing 1

```
1  <?php
2      function a() {
3          echo "(a) we're now in level ",
4               xdebug_get_stack_depth(), "\n";
5          b();
6      }
7
8      function b() {
9          echo "(b) we're now in level ",
10              xdebug_get_stack_depth(), "\n";
11     }
12
13     echo "main body is level: ",
14          xdebug_get_stack_depth(), "\n";
15     b();
16     a();
17
18 /* This script outputs:
19     main body is level: 1
20     (b) we're now in level 2
21     (a) we're now in level 2
22     (b) we're now in level 3
23 */
24 ?>
```

Where Did The Error Occur?

Q: When receiving a warning or error like:

```
Warning: chdir(): No such file or directory (errno 2)
in include/dir.php on line 8123
```

I know on which line the code occurred but how do I find out by which function the function which raised the error was called?

A: With PHP you can create your own error handler and use the debug_backtrace() function to show a stack trace that gives you exactly this information. See Listing 2 for an example on how to do this. When you run this script, it will output something similar to Figure 1. As you see, it's quite some work to implement something as basic as showing a stack trace when an error occurs, and the debug_backtrace() function did not even exist before PHP 4.2.x. Around the time of the 4.2.x releases, I started working on Xdebug and the display of the stack when an error occurred was the second feature that I implemented. Shortly after this, the PHP development team added a similar function to PHP 5, and later backported it to PHP 4.3.

When Xdebug is loaded, the stack will be dumped for your automatically as soon as an error (or notice or warning) occurs. Parameters to functions are only shown as part of the stack trace when the xdebug.collect_params php.ini setting is set to one. Variables in

Figure 1

```
Error in /dat/docs/magazine/phpa/code/xdebug/listing2.php:34 saying:
        'chdir() [function.chdir]: No such file or directory (errno 2)'.

Local vars:

array
  'dir' => 'foo123'
  'path' => '/foo123'


#0  my_error_handler(2, chdir() [function.chdir]: No such file or
directory (errno 2), /dat/docs/magazine/phpa/code/xdebug/listing2.php, 34, Array ([dir] =>
foo123,[path] => /foo123)) called at [/dat/docs/magazine/phpa/code/xdebug/listing2.php:34]
#1  chdir(/foo123) called at [/dat/docs/magazine/phpa/code/xdebug/listing2.php:34]
#2  change_dir(foo123) called at [/dat/docs/magazine/phpa/code/xdebug/listing2.php:37]
```

the scope of the "highest" function are only shown when **xdebug.show_local_vars** is enabled. The result of this same script (without the set_error_handler() call) is shown in Figure 2.

How Can I Debug Request Variables?

Q: Whenever I get an error, I would like to see information from the superglobals, such as **$_POST**, together with the other debugging information. Is there a way to do that easily?

A: Xdebug can automatically show information from the superglobals whenever an error occurs, very similar to the way it displays local variables as described in the previous section. To configure this feature, you need to make certain settings in php.ini. For each super global, there is a specific setting—for example, **xdebug.dump.POST** for the **$_POST** array, and all other superglobals—**COOKIE**, **ENV**, **FILES**, **GET**, **REQUEST**, **SERVER** and **SESSION**—are supported as well in a similar fashion. The value of each setting is a comma separated list of the indices of the variables that you want to show—make sure that you do not use any spaces in the setting's value. In case you want to see all the elements of a specific superglobal, you can use the special wildcard value *. For example, if you use the settings illustrated in Listing 3, Xdebug will show information about all **COOKIE** variables, the **POST** variables **login** and **password**, and the **SESSION** variables **id**, **login** and **hash** on each error. An example of how this might look like is shown in Figure 3. (You can find the files for "creating" this error in the **superglobals/** directory in the downloadable files accompanying this issue).

Listing 2

```
<?php
    if (!extension_loaded('Xdebug')) {
        set_error_handler('my_error_handler');
    }

    function my_error_handler($errno, $msg, $file, $line, $context)
    {
        switch ($errno) {
            case E_NOTICE:
            case E_WARNING:
                echo "<pre>";
                echo "Error in $file:$line saying:\n\t'$msg'.\n\n";
                echo "Local vars:\n";
                var_dump($context);
                echo "\n";
                ob_start();
                debug_print_backtrace();
                $contents = ob_get_contents();
                ob_end_clean();
                echo wordwrap($contents, 100);
                echo "</pre>";
                break;
            default:
                break;
        }
    }

    function change_dir($dir)
    {
        $path = $dir;
        if ($path{0} != '/') {
            $path = '/'. $path;
        }
        chdir($path);
    }

    change_dir('foo123');
?>
```

How do I "Pretty Print" Variables For Debugging?

Q: Whenever I use print_r() or var_dump() to display variables while debugging, I have to surround the function call with **<pre>...</pre>** in order to make the output somewhat readable. Even then, it's often hard to figure out what I'm seeing.

A: Xdebug implements its own variable display func-

Listing 4

```
<?php
    class TimeStuff {
        private $timestamp;
        private $user_defined;
        private $self;
        protected $tm;
        public $date;

        function TimeStuff($ts = null)
        {
            $this->self = &$this;
            $this->timestamp = $ts === null ? time() : $ts;
            $this->user_defined = ($ts !== null);
            $this->date = date("Y-m-d H:i:s T", $this->timestamp);
            $this->tm = getdate($this->timestamp);
        }
    }

    $ts1 = new TimeStuff(1092515106);

    var_dump($ts1);
?>
```

Listing 5

```
<?php
    function get_time()
    {
        $t = microtime();
        $a = split(' ', $t);
        return $a[1] + $a[0];
    }

    $start = get_time();

    $j = 0;
    for ($i = 0 ; $i < 250000; $i++) {
        $j += $i;
    }

    $end = get_time();

    echo "The script took ". ($end - $start).
        " seconds to execute.<br />";
    echo "The script took ". xdebug_time_index().
        " seconds to execute.<br />";
?>
```

Listing 3

```
xdebug.dump.COOKIE=*xdebug.dump.POST=login,passwordxdebug.dump.SESSION=id,login,hash
```

tion—xdebug_var_dump()—to display variables. When this new function is used with the Command Line Interface (CLI) of PHP, the format is only a bit different from PHP's var_dump() function, but when you use it through your web browser, the function will display a variable laid out in proper HTML and colour-coded depending on its type. Besides this new function, Xdebug will also override PHP's var_dump() function so that your scripts will require no changes to make use of Xdebug's improved variable display capabilities.

Figure 4 shows how Xdebug's variable display function shows the complex variable from Listing 4.

How Much Time Does my Script Take?

Q: I want to know how long my script, or a part of my script, takes to execute. How can I do that?

A: In PHP, you can do this by using the microtime()

Figure 4

```
object(TimeStuff)[1]
  private 'timestamp' => 1092515106
  private 'user_defined' => true
  private 'self' =>
    &object(TimeStuff)[1]
  protected 'tm' =>
    array
      'seconds' => 6
      'minutes' => 25
      'hours' => 22
      'mday' => 14
      'wday' => 6
      'mon' => 8
      'year' => 2004
      'yday' => 226
      'weekday' => 'Saturday'
      'month' => 'August'
      0 => 1092515106
  public 'date' => '2004-08-14 22:25:06 CEST'
```

Figure 2

Warning: *chdir() [function.chdir]: No such file or directory (errno 2)* in */dat/docs/magazine/phpa/code/xdebug/listing2.php* on line *34*

Call Stack

#	Function	Location
1	{main}()	/dat/docs/magazine/phpa/code/xdebug/listing2.php:0
2	change_dir('foo123')	/dat/docs/magazine/phpa/code/xdebug/listing2.php:37
3	chdir ('/foo123')	/dat/docs/magazine/phpa/code/xdebug/listing2.php:34

Variables in local scope (#2)

Variable	Value
$path =	'/foo123'
$dir =	'foo123'

Figure 3

Notice: *Undefined index: login* in */dat/docs/magazine/phpa/code/xdebug/superglobal/info.php* on line *3*

Call Stack

#	Function	Location
1	{main}()	/dat/docs/magazine/phpa/code/xdebug/superglobal/info.php:0

Dump *$_POST*

Dump *$_COOKIE*

$_COOKIE['PHPSESSID'] =	'8a3bfa6310b8387ca62b3758e0a3e8ae'

Dump *$_SESSION*

$_SESSION['id'] =	'derick'
$_SESSION['hash'] =	'2bbfd28be49ff9edc66bff90c6f5c6ce'

Variables in local scope (#1)

Variable	Value
$_SESSION =	array 'id' => 'derick' 'hash' => '2bbfd28be49ff9edc66bff90c6f5c6ce'

You're logged in as !

function—but this is not very trivial, as you can see in Listing 5. Because microtime() returns the time in a strange format (the number of microseconds, followed by a space and the number of seconds), you have to create a helper function to generate a floating point number from it. Defining this function and executing it takes time too, so your measurements are less accurate than they could have been.

You can see in the listing that the second line that prints the number of seconds to execute the script makes use of the Xdebug function xdebug_time_index(), which always returns the time since the request for the script was made through the browser, which is often before the code in your written script starts to execute. This is why you see the difference in time between the "old" method and the Xdebug method.

If you want to measure how much time a specific section of your script takes, then the differences between both methods would be marginal, but if you use Xdebug's xdebug_time_index() function you still save time because you don't have to write your equivalent of the get_time() function. The measurement will also be a bit more accurate, too.

What is The Memory Usage of my Script?

Q: I am concerned about the memory usage of my script; how can I find out how much it uses overall, and what its peak memory usage is?

A: When I added the functionality to measure the memory usage at any given point during the script's execution and the peak memory usage to Xdebug, PHP itself did not have any functions to retrieve this information. Nowadays, PHP provides the memory_get_usage() function for this purpose. Xdebug's equivalent is the xdebug_memory_usage() function, which retrieves the current amount of script memory that is used.

However, Xdebug has an additional function, xdebug_peak_memory_usage(), that can be used to retrieve information about the script's peak memory usage. Both functions are only available when PHP is compiled with the --enable-memory-limit switch, as the information on memory usage itself is only available when memory limit checking is enabled. Of course, this is also true for PHP's native memory_get_usage() function.

The example in Listing 6 shows how you can use the two memory-related functions. The output of this script can be found in Figure 5, although when you run the script you might get slightly different results due to differences in your PHP installation compared to the one I used to generate the screenshot.

When looking at the figure, you might notice that there is some kind of memory 'leak' as the memory usage at the end of the script is more than 14kb higher than at the start of the script. You do not have to worry about this—the 'leak' is introduced because the Zend Engine intelligently caches some memory structures which are reused later in your script. This is why, after the second set of function calls, the memory usage does not really grow once the array is unset, as the same structures were used in both sets of functions.

One note should be made about the amount of memory that is returned by the functions. They often do not include memory allocated by the libraries that the PHP extensions link against because those libraries do not use PHP's memory allocation functions; this memory then also not included in the memory limit protections that PHP offer. There are a few exceptions to this; the bundled GD library and some parts of PHP 5's XML extensions, for example, do use PHP's memory allocation functions. Other extensions may follow later.

What is The Flow of my Script?

Q: I inherited maintenance duties for a script and I like to know how it works inside out. Is there a way for me to examine its execution function by function?

A: Xdebug provides functionality to create a trace of a script, which includes calls to all functions and, optionally, the values assigned to each parameter to the functions being called and the values returned by the latter.

There are several ways to make a trace of a script. The first one is called "auto tracing" and can be enabled with a php.ini setting. The name of the setting is, unsurprisingly, xdebug.auto_trace, which can be either set to 1 or 0 (the default). When this setting is enabled, Xdebug will create a trace file for each invocation of a PHP script. There are several php.ini settings to control this behavior. With the setting xdebug.trace_output_dir, for instance, you can specify

Listing 6

```
1  <?php
2  echo xdebug_memory_usage(), '-',
3       xdebug_peak_memory_usage(), "<br />\n";
4
5  $test_array = range(0, 2500);
6  unset($test_array);
7
8  echo xdebug_memory_usage(), '-',
9       xdebug_peak_memory_usage(), "<br />\n";
10
11 $test_array = range(0, 3750);
12 unset($test_array);
13
14 echo xdebug_memory_usage(), '-',
15      xdebug_peak_memory_usage(), "<br />\n";
16 ?>
```

Figure 5

```
38608-47360
52904-195056
52904-265056
```

the directory where the trace file should be written to; this is, by default, the **/tmp** directory—if you're running your scripts on Windows, you will have to change this setting, as **/tmp** obviously does not exist (unless you create a directory called "tmp" in your main drive). The format of the filename of the trace file can be manipulated by tweaking the **xdebug.trace_output_name** setting. When this is set to **crc32**, the name of the trace file has the format **trace.**, followed by a crc32 checksum of the current working directory, followed by the **.xt** extension. If the setting has any other value, the middle part (that is, the crc32 checksum of the current working directory) is replaced by the ID of the current process (PID). Lastly, the **xdebug.trace_options** setting that configures various options for creating traces. At the moment, the only available option is 1, which will cause Xdebug to open the trace files in "append" mode rather than "overwrite" mode.

The settings below will start automatic tracing of scripts and cause Xdebug to write the results to the **/tmp/xdebug-traces** directory in append mode where the filename consists of **trace.** plus the PID of the script followed by the **.xt** extension. The **xdebug.collect_params** and **xdebug.collect_return** settings control whether parameters to function calls and their return values should be included in the trace file.

```
xdebug.auto_trace = 1
xdebug.collect_params = 1
xdebug.collect_return = 1
xdebug.trace_output_dir = /tmp/xdebug-traces
xdebug.trace_output_name = pid
xdebug.trace_options = 1
```

The second way of starting a trace is by calling the **xdebug_start_trace()** function in your script. This function accepts two parameters: the first is the base name

Listing 7

```
1  <?php
2      function find_files_recursive($dir, &$results)
3      {
4          $entries = glob($dir. '/*');
5          foreach ($entries as $entry) {
6              if (is_dir($entry)) {
7                  $results[] = $entry;
8                  find_files_recursive($entry, $results);
9              } else {
10                 $results[] = $entry;
11             }
12         }
13     }
14
15     find_files_recursive("/tmp", $results);
16     var_dump($results);
17 ?>
```

Figure 6

```
TRACE START [2004-08-15 22:03:38]
    0.0004      40040    -> {main}() /dat/docs/magazine/phpa/code/xdebug/listing7.php:0
    0.0005      40184      -> find_files_recursive('/tmp', NULL) /dat/docs/magazine/phpa/code/xdebug
    0.0128      40872        -> glob('/tmp/*') /dat/docs/magazine/phpa/code/xdebug/listing7.php:4
                              >=> array (0 => '/tmp/OSL_PIPE_1000_SingleOfficeIPC_79933bae693855935f
    0.0131      41992        -> is_dir('/tmp/OSL_PIPE_1000_SingleOfficeIPC_79933bae693855935f4039889
                              >=> FALSE
    0.0132      42184        -> is_dir('/tmp/gconfd-derick') /dat/docs/magazine/phpa/code/xdebug/lis
                              >=> TRUE
    0.0132      42208        -> find_files_recursive('/tmp/gconfd-derick', array (0 => '/tmp/OSL_PIP
    0.0133      43000          -> glob('/tmp/gconfd-derick/*') /dat/docs/magazine/phpa/code/xdebug/l
                                >=> array ()
                              >=> NULL
    0.0134      42568        -> is_dir('/tmp/keyring-T97mpb') /dat/docs/magazine/phpa/code/xdebug/li
                              >=> TRUE
    0.0135      42592        -> find_files_recursive('/tmp/keyring-T97mpb', array (0 => '/tmp/OSL_PI
    0.0136      43360          -> glob('/tmp/keyring-T97mpb/*') /dat/docs/magazine/phpa/code/xdebug/
                                >=> array ()
                              >=> NULL
    0.0137      42800        -> is_dir('/tmp/mapping-derick') /dat/docs/magazine/phpa/code/xdebug/li
                              >=> FALSE
    0.0138      42864        -> is_dir('/tmp/mc-derick') /dat/docs/magazine/phpa/code/xdebug/listing
                              >=> TRUE
    0.0138      42880        -> find_files_recursive('/tmp/mc-derick', array (0 => '/tmp/OSL_PIPE_10
    0.0139      43680          -> glob('/tmp/mc-derick/*') /dat/docs/magazine/phpa/code/xdebug/listi
                                >=> array ()
                              >=> NULL
    0.0140      43048        -> is_dir('/tmp/orbit-derick') /dat/docs/magazine/phpa/code/xdebug/list
                              >=> TRUE
    0.0141      43000        -> find_files_recursive('/tmp/orbit-derick', array (0 => '/tmp/OSL_PIPE
/tmp/xdebug-traces/trace.4481.xt                                              1,1           Top
"/tmp/xdebug-traces/trace.4481.xt" 80L, 9656C
```

for the trace file's location and the second optional parameter configures the options for the trace output. Because Xdebug automatically appends ".xt" to the base name of the trace file, there is a function, called `xdebug_get_tracefile_name()`, that returns the full filename of the trace file.

`xdebug_start_trace()` supports two options: `XDEBUG_TRACE_APPEND`, which will create the trace file in append mode, and `XDEBUG_TRACE_COMPUTERIZED`, which will change the format of the trace file (more about that later). To stop a manually started trace, you can use the `xdebug_stop_trace()` function, which does not accept any parameters. Please do keep in mind that there can only be one trace (manually or automatic) running at any given time.

The script in Listing 7 is started with the `php.ini` settings described above. When the script is run, the trace file that is generated is shown in Figure 6. The colour-coded formatting comes from a VIM syntax file called `xt.vim`, which instructs VIM to highlight the trace file according to a set of special rules. The `xt.vim` file can be found in the Xdebug source package and can be easily installed. Please refer to the Xdebug documentation at *http://xdebug.org/install.php#vim* for instructions on how to configure VIM to use this syntax-highlighting file.

The trace file shows the following information: the start and end time of the trace (the first and the last line) and, for each function call, the current time index since the beginning of the script, the amount of memory in use at that moment, the function name, its parameters and the location from where the function was called, all in one line.

Nested function calls are indented, so that you can immediately determine the order in which the execution takes place. For each corresponding function call, there is also a line with the return value of the function. In the trace from Figure 6, you can see that the function `find_files_recursive()` (line 3) was called with the parameters `/tmp` and `null`. This function called the `glob()` function with, as a parameter, the string `/tmp/*`. The `glob()` function returned an array. Then, the `find_files_recursive()` function called the `is_dir()` function for each element in the array. And, when that function returned `True`, the `find_files_recursive()` was called again.

While this trace is very easy to read for humans like you and me, it is not very easy to parse for programs. This is why there is a second format for trace files, called the "computerized format," which you can see in

Figure 7

```
Version: 2.0.0dev
TRACE START [2004-08-15 22:06:18]
1    0    0    0.000358    40048    {main}  1       /dat/docs/magazine/phpa/code/xdebug/listing7.php
2    1    0    0.000521    40192    find_files_recursive    1       /dat/docs/magazine/phpa/code/xdebug
3    2    0    0.000596    40904    glob    0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
3    2    1    0.000736    41928
3    3    0    0.000801    42152    is_dir  0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
3    3    1    0.000846    42224
3    4    0    0.000877    42312    is_dir  0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
3    4    1    0.000917    42336
3    5    0    0.000944    42360    find_files_recursive    1       /dat/docs/magazine/phpa/code/xdebug
4    6    0    0.001011    43048    glob    0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
4    6    1    0.001051    43128
3    5    1    0.001081    42552
3    7    0    0.001125    42592    is_dir  0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
3    7    1    0.001166    42592
3    8    0    0.001192    42592    find_files_recursive    1       /dat/docs/magazine/phpa/code/xdebug
4    9    0    0.001255    43216    glob    0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
4    9    1    0.001293    43216
3    8    1    0.001321    42592
3    10   0    0.001364    42632    is_dir  0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
3    10   1    0.001405    42632
3    11   0    0.001433    42672    is_dir  0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
3    11   1    0.001472    42672
3    12   0    0.002268    42672    find_files_recursive    1       /dat/docs/magazine/phpa/code/xdebug
4    13   0    0.002334    43312    glob    0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
4    13   1    0.002372    43312
3    12   1    0.002400    42688
3    14   0    0.002443    42728    is_dir  0       /dat/docs/magazine/phpa/code/xdebug/listing7.php
3    14   1    0.002485    42728
/tmp/xdebug-traces/trace.4482.xt [RO]                                   1,1           Top
"/tmp/xdebug-traces/trace.4482.xt" [readonly] 84L, 4403C
```

Figure 7. The information shown in this format is very much the same, except that the fields are all separated by tabs and that there is no indentation for the function names. Instead, the first column of the file consists of the "nesting" level, which, in combination with the second column (that shows the function number) can be used to re-create the same format as the human-readable output. The third field is a number; zero means the start of the function call and one the end of the function call. Together with the time index in the fourth field, it is easy to calculate how long each function call takes—and it is also not very hard to subtract the time spent in calling other functions from this function.

The fifth column shows the memory usage, except this value is also shown when the function exists. This is followed, in order, by: the name of the function, whether it is a user-defined function (one) or a PHP internal function (zero), the filename from which the function was called and the line number from which the function was called.

Currently, there is only one tool that is able to read the trace files in "computerized format"—Xgui—but this tool is still under development and not yet publicly available. However, once we release it we'll make both a Windows and Linux version available (and, indeed, provide support for any other system on which the QT libraries can be compiled and the necessary build tools are available).

Figure 8 shows the trace file from Figure 8 displayed with this tool. In the screenshot, you can see three columns; the first one displays a tree of functions, which shows immediately which function called which. By default, the whole tree is expanded so that all function calls are shown. The second column shows the dif-

Listing 8

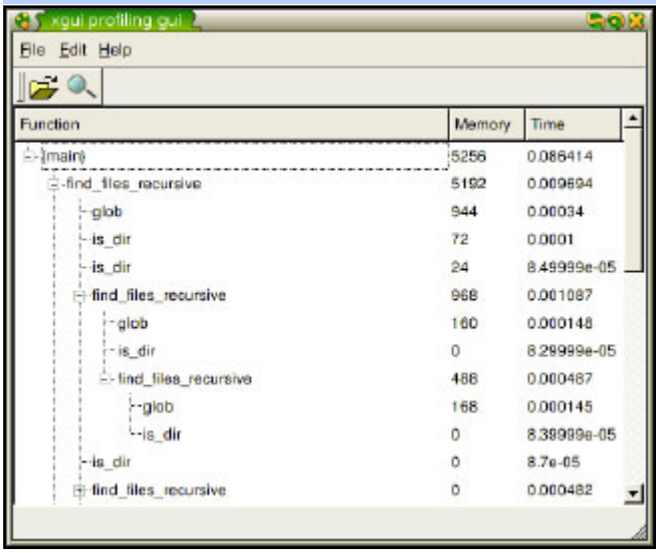
```
<?php
    error_reporting(E_ALL);

    if (!isset($_POST['file'])) {
        /* (A) Show list with tests */
        echo "Select your test:\n";
        echo "<form method='post'>\n";
        $dir = glob('tests/*.t');
        foreach ($dir as $file) {
            echo "<input type='submit' name='file'
                value='$file'/><br/>\n";
        }
        echo "</form>\n";
    } else {
        /* (B) Run test */
        echo "Output:\n\n";
        xdebug_start_code_coverage();
        include $_POST['file'];
        $cc = xdebug_get_code_coverage();
        xdebug_stop_code_coverage();

        /* (C) Strip out all unwanted files from the
         * coverage information */
        $clean = array();
        $clean[$test] = $cc[$test];

        /* (D) Open files and write it highlighted */
        echo "Coverage analysis:\n\n";
        foreach ($clean as $file => $cc) {
            echo "<pre>";
            $fc = file($file);
            $line_nos = array_keys($cc);
            foreach ($fc as $ln => $line) {
                if (in_array($ln + 1, $line_nos)) {
                    $bgc = "aaaaaa";
                } else {
                    $bgc = "dddddd";
                }
                echo "<div style='background-color: #$bgc'>";
                echo @sprintf("%3d [%2d]:\t", $ln + 1,
                        $cc[$ln + 1]);
                echo htmlspecialchars($line);
                echo "</div>";
            }
            echo "</pre>";
        }
    }
?>
```

Figure 8



Listing 9

```
<?php
    $test = realpath("tests/bigassfunction.php");
    include $test;

    $formats = array(
        '2004-08-17 21:20',
        '20040817 2120',
    );
    echo "<pre>";
    foreach ($formats as $format) {
        test_parse_time($format);
    }
    echo "</pre>";
?>
```

Listing 10

```
<?php
    function test_parse_time($f)
    {
        if (preg_match("/[0-9]{12}/", $f)) {
            $type = "YYYYMMDDHHii";
        }
        else if (preg_match("/[0-9]{8}\ [0-9]{4}/", $f)) {
            $type = "YYYYMMDD HHii";
        }
        else if (preg_match("/[0-9]{4}-[0-9]{2}-[0-9]{2}/",
$f)) {
            $type = "YYYY-MM-DD";
        }
        else {
            $type = "*unknown*";
        }

        echo "This is a $type format.\n";
    }
?>
```

ference in memory usage between the start of the function call and the end of the function call, including all allocated memory in the functions each listed function called. The last column shows the time used for each function call and calls to its children.

Besides the fact that this tool is not yet available, it is also not very complete. In the future, we plan to add a lot of functionality to it, including features to highlight the slowest functions. Stay tuned to the Xdebug site (*http://xdebug.org*) for more information.

Do I Use All the Code in My Scripts?

Q: While developing a test suite for my web application's functions I want to know if all the code I have written is covered in my tests. How do I do that?

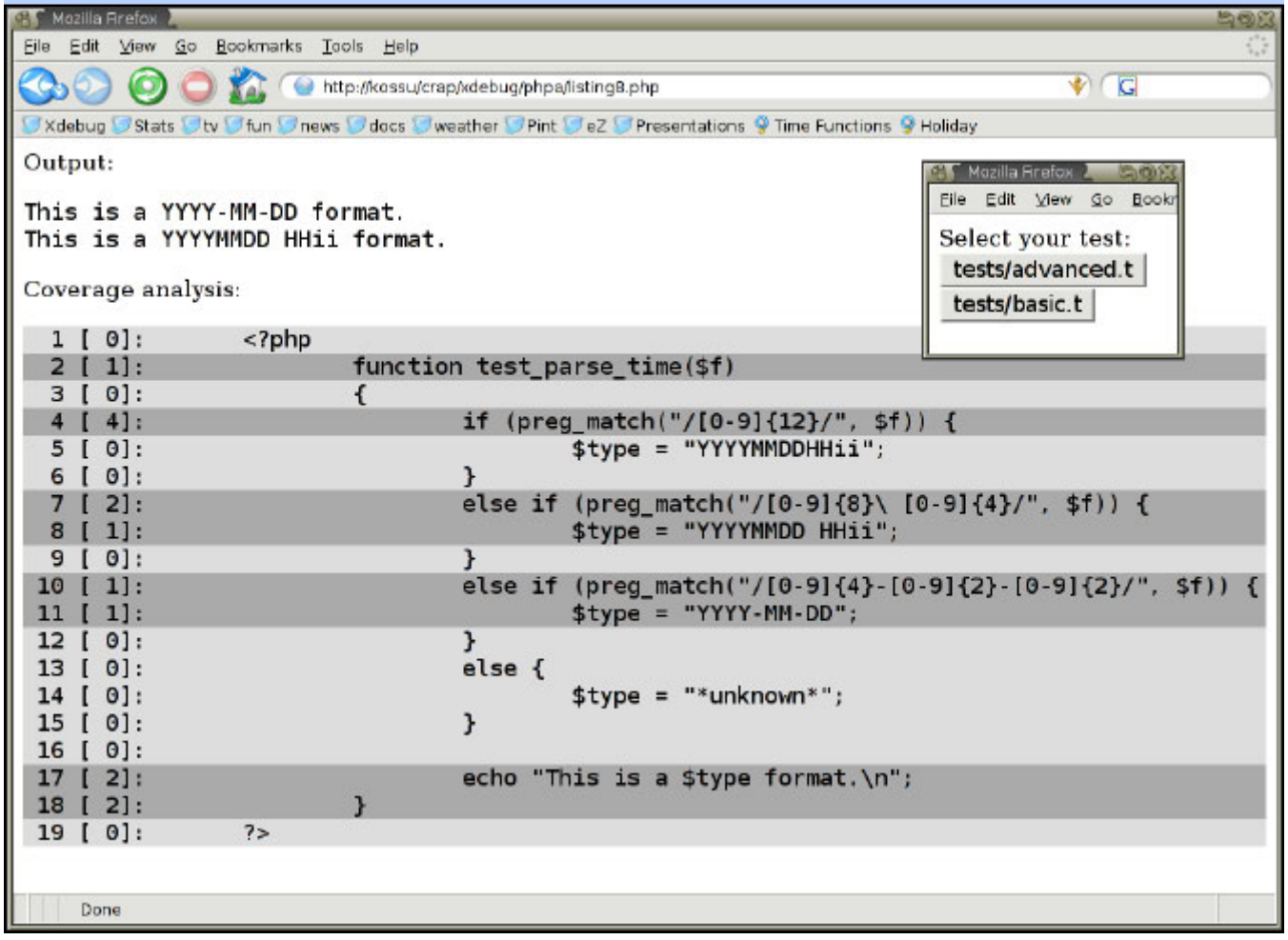
A: A technique to find out which code is actually run in your script is called "Code Coverage." Xdebug has functionality to check on which lines statements in scripts are executed. In order to take advantage of it, you need to make sure that the **xdebug.extended_info** setting in your php.ini file is set to 1—otherwise, Xdebug can not gather the information it requires. Listing 8 shows a very simple and basic way of running unit tests while also checking if all lines in the function-to-be-tested are actually executed. The script expects the following directory structure:

```
./listing8.php
./tests/basic.t
./tests/advanced.t
```

Listing 11

```
1  <?php
2      function find_files_recursive($dir, &$results)
3      {
4          $entries = glob($dir. '/*');
5          foreach ($entries as $entry) {
6              if (is_dir($entry)) {
7                  $results[] = $entry;
8                  find_files_recursive($entry, $results);
9              } else {
10                 $results[] = $entry;
11             }
12         }
13     }
14
15     $start = xdebug_get_function_count();
16     find_files_recursive("/tmp", $results);
17     $count = xdebug_get_function_count() - $start - 1;
18     echo $count, "\n";
19 ?>
```

Figure 9



The *.t files are the test files which are responsible for running the test case. You can find basic.t in Listing 9 and the script that this test case uses (tests/bigassfunction.php) can be found in Listing 10.

When you open up Listing 8 in your browser, it will generate a list (A) of all the test files that are available and display them as a button (inlay image in Figure 9). When you click on a button, the system will run the corresponding test file.

Below "(B)," you can see that the script uses xdebug_start_code_coverage() to start collecting information on code coverage. After the test (Listing 9) is run, it uses xdebug_get_code_coverage() to retrieve a two dimensional array in which the first dimension is the filename and the second dimension the line number. This value contains the number of times a specific line was touched during the execution of the test. After the information is extracted from Xdebug, the xdebug_stop_code_coverage() function is called to stop gathering information. In section (C), we single out the file we're interested in (returned by the test file in the $test variable). Then, at last, in section (D) we read the test file, split it into lines with the file() function, loop through it and display the lines that exist in the code coverage data contained in the $clean variable. We also display the number of "hits" on a specific line. The full result of the display is shown in Figure 9.

Sometimes, the Zend Engine plays a few tricks that make some lines not show up in the code coverage information, thus making Xdebug's output inaccurate. The reasons why this happens bear some further investigation, and we plan to address them in future versions of Xdebug.

How Many Functions do I Call?

Q: I am interested in how many functions a specific part of my script calls because of optimization reasons. How can I find out?

A: Xdebug has a special function for this, called xdebug_get_function_count(). This function always returns the number of functions since the start of the script, but you can, of course do a few simple calculations to figure out how many calls a specific section of your script makes. In Listing 11, we attempt to calculate how many functions are called in our find_files_recursive() function from Listing 7. Before we start the main find_files_recursive() function, we store the number of function calls up to that point in the $start variable and then, when the function ends, we calculate the amount of function calls by subtracting the value of $start + 1 from the current function call count. We subtract the extra "1" because the call to xdebug_get_function_count() is added to the function count as well. You should also note that calls to include(), require(), include_once(), require_once() and eval() add to the function count as well, even though they are not real functions (they are language constructs). This is why the script in Listing 12 outputs "1,4"; function 1 is the first xdebug_get_function_count(), "function" 2 is eval(), function 3 is abs() and function 4 is the second xdebug_get_function_count() function.

Listing 12

```
1 <?php
2     echo xdebug_get_function_count(), ",";
3     $foo = eval("abs(-4);");
4     echo xdebug_get_function_count(), "\n";
5 ?>
```

How do I Analyze the Performance of My Script?

Q: My script is slow and I'd like to know why. Xdebug doesn't have anything to help me here, does it?

A: But of course it does! Xdebug has functionality to analyze the performance of your script—it's called "profiling." I have to admit, however, that in the Xdebug 1.x series profiling was in a pretty bad state, as it introduced too much overhead and thus invalidated the profiling results, particularly when dealing with the larger applications that you usually want to profile.

Xdebug 2 has a whole new profiling concept in which Xdebug only generates a profiler information file, which is similar to the function tracing feature, except that the format of the file is fully focused on generating profiling data in the most efficient way. The file format of the profiler information file is the same format that cachegrind—a memory management and profiling tool for C applications on Linux—uses, since there is a great tool for visualizing profiler information

Listing 13

```
1 <?php
2 $text = "Ce fichier à pour but de rassembler une liste de tra-
duction de ".
3     "termes techniques, pour aider à la consistance du
manuel.";
4
5 function do_split($text)
6 {
7     return split(' ', $text);
8 }
9
10 function convert_word($word)
11 {
12     return iconv('iso-8859-1', 'utf-8', $word);
13 }
14
15 function do_convert($words)
16 {
17     return array_map('convert_word', $words);
18 }
19
20 function do_concat($words)
21 {
22     return join(" ", $words);
23 }
24
25 $splitted = do_split($text);
26 $converted = do_convert($splitted);
27 $joined = do_concat($converted);
28
29 echo wordwrap($joined);
30 ?>
```

files in this format: KCacheGrind. However, this tool requires KDE libraries, which you normally only find on Linux machines—you do not have to run KDE as your desktop environment, just having the libraries installed is more than enough. Debian users can install KCacheGrind with `apt-get install kcachegrind`. In the future, I plan to introduce more profiling output formats through Xdebug.

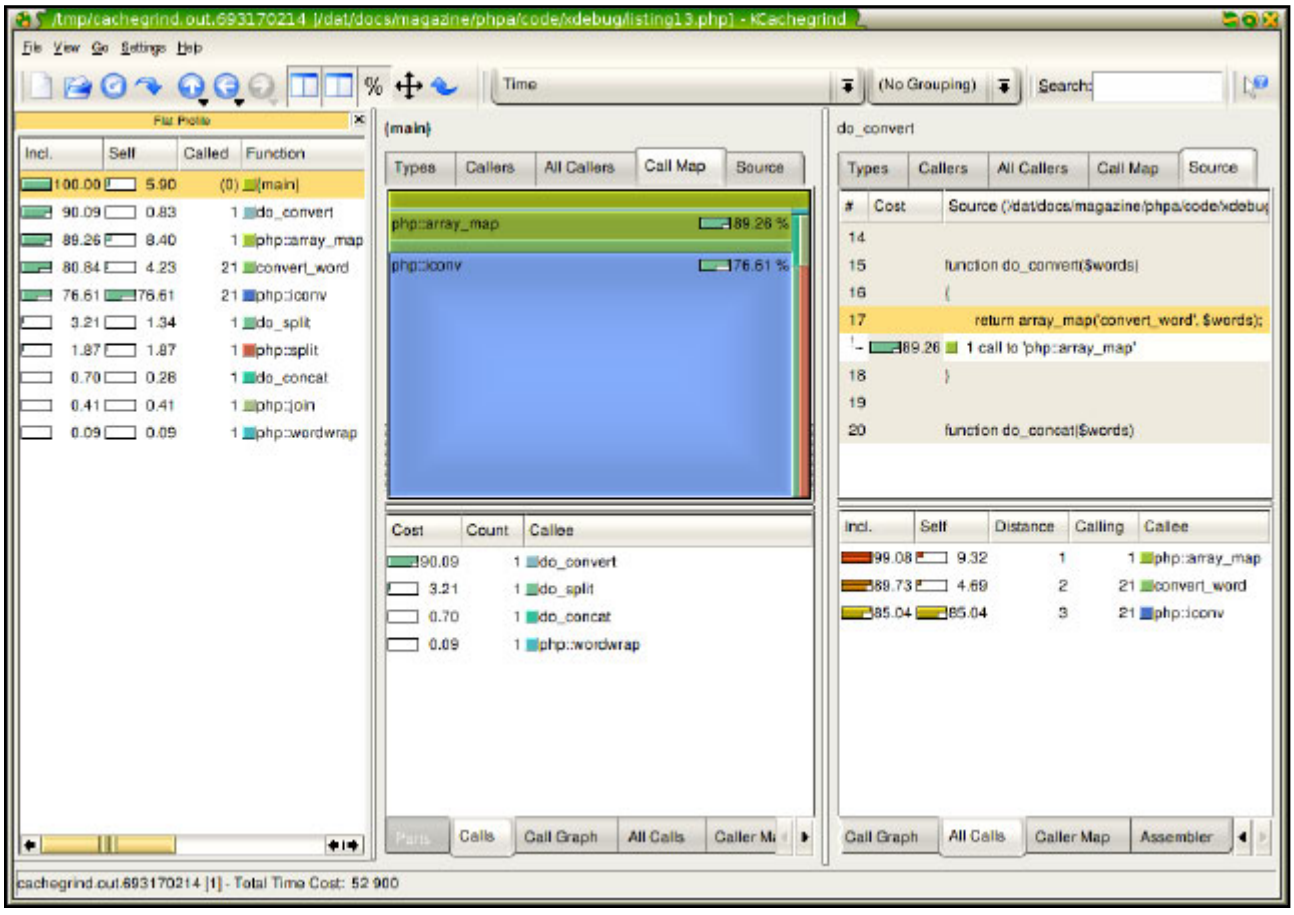
Setting up profiling requires you to make a number of `php.ini` settings, they are:

```
xdebug.profiler_enable=1
xdebug.extended_info=0
xdebug.remote_enable=0
xdebug.auto_trace=0
xdebug.profiler_output_dir=/tmp
xdebug.profiler_output_name=crc32
```

The first one is obviously the one that activates Xdebug's profiling functionality, but the second one is more mysterious. While you need to have this setting enabled for most other features in Xdebug (such as Code Coverage and Remote Debugging), it is usually a good idea to turn it off while profiling code. When this setting is set to one, it will instruct the PHP parser to generate extra instructions for the PHP executor so that it is possible to add a hook after each statement. This is needed in order to be able to set breakpoints on arbitrary lines. However, because extra instructions are generated, the code that PHP internally needs to execute will be about 33% larger—and that, of course, makes your script a bit slower. Because mixing profiling together with the other features in Xdebug doesn't really make much sense—who would want to debug a script while also profiling it?—it is not much of a problem to have to turn off this setting. In the future, Xdebug will most likely do this for you automatically when you enable profiling in your php.ini file.

The next two options turn off the generation of trace files and the remote debugger—the goal of this is, again, to make the script run as "naturally" as possible; both options would dramatically slow down execution if they were still enabled. The last options serve the same purpose as when you are generating trace files: the first one selects the directory in which to place the generated profiler information file and the second one selects its filename. `crc32` will cause Xdebug to include the CRC32 checksum of the current working directory into the filename, while any other value will cause it to include the PID of the running PHP process into the filename. All files generated by the profiler have a name in

Figure 10



the following format: `cachegrind.out.<number>`. This is required in order for KCacheGrind to open the profiler information file.

Let's start with profiling a small example script, which you can find in Listing 13. After I ran the script, a 4kB profiling information file appeared in the `/tmp` directory, which, when opened in KCacheGrind (and with a bit of tweaking of the layout) caused a display like the one in Figure 10 to appear. Although I won't cover all the features that KCacheGrind offers—you can find more information on this on the Xdebug documentation page at *http://xdebug/docs-profiling2.php*—I will describe what you see in the screenshot. On the left side of the screen, you will find a "flat profile" that lists all the functions that were called in the script. The first column shows how much time in percentage was spent in the listed function, including calls to other functions. The second column only shows the time spent in the listed function, excluding calls to other functions. The third column shows how often a specific function was called and the fourth column lists the function's name.

> "Instead of modifying your code, Xdebug offers a non-intrusive way of debugging your application."

Xdebug will prefix the name for internal PHP functions such as `array_map()` with `php::` in order to be able to tell them apart from user-defined functions; included files are listed as a function as well, with names like `include::/path/to/included_file.php`. The last column (not in the screenshot) displays the filename in which the function was defined.

The right side of the screen is split into two columns, which, in turn, is split into two rows. Each column shows information about a specific function, in our case the pseudo-function `{main}` on the left and the `do_convert()` function on the right. In the left column, I selected the **Call Map** tab, which displays a map where a larger area represents more time spent in this function. As you can see here, the execution of the `iconv()` function took 76.61% of the time of the total execution of the script. The maps are stacked and, when you move your mouse pointer over the area, you see a simple stack trace describing the percent of time spend in that function.

The top half of the right column shows the annotated source code of the selected function. For each function call within it, KCacheGrind shows you the number of calls to the function and the total time spent in it while being called from the selected function. In our case, there was just one call to `array_map()`. The numbers don't add up to 100% because the calling of the function itself generates some overhead as well, and non-function calls (such as language constructs like `if` and `foreach`) use up time, too. It's obvious that the source code for internal PHP functions cannot be displayed, as there is simply no source code available for them!

On the lower left column, I selected the **Calls** tab, which shows all the functions that were called from the selected function—`{main}` in our case. The lower right column shows not only the functions that were called directly, but also those that were called through functions that were called from the selected function. As that sentence is probably very hard to understand (how much wood would a wood chuck chuck...), I'll give you an example. The `distance` column describes how far from the selected function the listed function was called. Everything with a distance of 1 was called directly by the selected function. In the screenshot, this only applies to the `array_map()` function. Because there is only one function with a distance of 1, we are sure that the function with a distance of 2 was called by it. In our case, the `convert_word()` function was indeed called as callback through the `array_map()` function.

Similarly, because there is also only one function with a distance of 2 we can conclude that all functions with a distance of 3 were called by it. In our case, this applies to the `iconv()` function. With a more advanced script as your guinea pig, it would have been possible for a specific function to have multiple distance values; for example, `strlen()` could have been called directly by the selected function, and also through a different function first. In that case, KCacheGrind will display something like `1-2(2)` as a distance value, where the `1-2` describes the range of distances and the `(2)` describes the median distance, which is where most of the functions were called from. KCacheGrind has plenty of other features, and the best way to get to know them is just to play around with it.

How do I Analyze my Script While it is Running?

Q: I have a problem somewhere in my script, and I like to know what is going on. Of course, I can add `var_dump()` calls everywhere, but that is not very effective. Any idea for a better method?

A: Instead of modifying your code, Xdebug offers a non-intrusive way of debugging your application. With the built-in "remote" debugger, you can tell Xdebug to connect to a debug client as soon as a request is made

through the web server. Again, there are a few php.ini settings involved. The most optimal combination is listed below:

```
xdebug.remote_enable=1
xdebug.remote_handler=dbgp
xdebug.remote_mode=req
xdebug.remote_host=localhost
xdebug.remote_port=9000
xdebug.extended_info=1
```

All of these are default settings, except for **xdebug.remote_enable**, which defaults to zero. Xdebug understands multiple protocols for its debugging interface. There is the **gdb** protocol, which makes it compatible with the popular GNU debugger that most open-source developers have probably used at some point to debug their C applications, as well as **dbgp**, a brand new language-agnostic debugger protocol that I developed together with Shane Caraveo at ActiveState for adding debugging support in their Komodo IDE. Although the DBGp protocol is a bit harder to use than the GDB protocol for a human being, it is much more powerful and makes it possible to perform more effective debugging—and that is why we will be using it here in our next example. Normally, you would use a GUI client to manage your debugging activities, but there is only a very simple client bundled with Xdebug. The protocol uses plain ASCII commands to the debugger in the form **command -a optiona -b optionb**, very much like the arguments you would pass to programs on the command line. The answer returned by Xdebug is always an XML packet.

> "There is one occasion in which PHP is 'allowed' to crash, and that is when a script has an infinite recursion loop in its code and an overflow occurs."

First of all, we need to compile the **debugclient** application—at least version 0.8—which you can find in the **debugclient/** subdirectory of the Xdebug source package. There is no binary client for Windows at this moment (unless you compile under cygwin). It is a good idea to install **libedit** (**apt-get install libedit2 libedit-dev** for Debian users), as this adds a history buffer to the client. You can issue the following commands to compile **debugclient** and install it in **/usr/local/bin**:

```
./buildconf
./configure –with-libedit
make
make install
```

Now, whenever you run the **debugclient** command, it will wait until Xdebug connects to it. To start debugging, all you have to is point your browser to the script you want to debug and add this query parameter to the URL:

```
?XDEBUG_SESSION_START=application
```

For example:

```
http://ez34/index.php/galleries?XDEBUG_SESSION_START=p
hpa
```

In your running **debugclient** client, you will now see that Xdebug connected to the client and has sent you the **<init/>** packet. The important elements of this packet are the **fileurl** attribute, which contains the file that was requested, and the **idekey** attribute, which contains the value of the **XDEBUG_SESSION_START** request variable. From now on, it is possible to issue commands to the client that are described in the DBGp protocol specification (*http://xdebug.org/docs-dbgp.php*). This protocol is so elaborate that it is unwise to go into much detail here. I included a reformatted trace of a simple debugging session in Listing 14 (the source code for the script being debugged is, once again, the code from Listing 13). After Xdebug sees the special **XDEBUG_SESSION_START** request variable, it will set a cookie, so further requests do not need changes to the URL query string. Xdebug will continue connecting to **debugclient** unless you either remove the cookie yourself, or use another special request variable—**XDEBUG_SESSION_STOP**.

It is, of course, also possible to debug command-line PHP scripts and, since there it is not possible to add an extra request variable to the "URL," you need to tell Xdebug that it needs to contact **debugclient** in a difference way, that is, by setting an environment variable as follows:

```
export XDEBUG_CONFIG="idekey=phpa"
```

Xdebug will stop trying to connect to the debug client once you remove the environment variable.

Since parsing XML by hand to debug an application is not very efficient, it is much better to use a graphical client to help you with your debugging. The XML protocol specification is pretty straightforward and the debug protocol is already implemented in Komodo 3

and in Maguma Workbench 2.1, which will be released soon. Unfortunately, those two products are commercial (though they might have a trial version) and the only free debug client for Xdebug that I know of—Weaverslave (*http://weaverslave.ws*)—only implements the older GDB protocol. In case you want to help out writing a cool interface (especially one running on Linux) for the DBGp protocol, feel free to drop a mail—I will gladly help you if you have any question about the protocol specification.

Where Can I Set All The Different Options?

Q: I read about the **xdebug.*** php.ini settings here, but where exactly can I set those?

A: They will all work from within the php.ini file itself, and all options except **xdebug.default_enable** and **xdebug.extended_info** can be set in an .httaccess file. Also, all options except these two, the **xdebug.profiler*** settings and **xdebug.remote_enable** can be set from within your script itself. However, you should keep in mind that some of these might not have the desired effect when set from within your script with `ini_set()`, as most of Xdebug's features are already activated before your script starts. I usually use an .htaccess file for configuring Xdebug on a per-project bases.

Anything Left?

Q: This all sounded very interesting, are there any famous last words left?

A: No last famous words, but if you still have questions I would like to point you to the Xdebug website (*http://xdebug.org*), which provides extensive documentation about Xdebug's features. If you still have problems, feel free to write to the *xdebug-general@lists.xdebug.org* mailing list.

About the Author ?>

Derick Rethans provides solutions for Internet related problems. He has contributed in a number of ways to the PHP project, including the mcrypt extension, bug fixes, additions and leading the QA team. He now works as a developer for eZ systems A.S.. In his spare time he likes to work on SRM: Script Running Machine and Xdebug, watch movies and travel. You can reach him at derick@derickrethans.nl

To Discuss this article:
http://forums.phparch.com/170