Benchmarking PHP modules

dr.sc. Alen Šimec, predavač
Tehničko veleučilište u Zagrebu
Vrbik 8, 10000 Zagreb
098/477 170
alen@tvz.hr

Davor Lozić, struč. spec. ing. techn. inf.
Tehničko veleučilište u Zagrebu
Vrbik 8, 10000 Zagreb
dlozic@tvz.hr

dr.sc. Lidija Tepeš Golubić, viši predavač
Tehničko veleučilište u Zagrebu
Vrbik 8, 10000 Zagreb
098/380956
ltepes2@tvz.hr

Abstract

This paper presents how C programming language could be used for this type of tasks, created as PHP module and then get imported in the PHP language. The purpose of this paper is to show how and when is better to build PHP modules in C, instead of normal PHP functions, and to show negative sides of this type of programming. Profiling, as an important aspect of finding application bottlenecks, is also discussed. Profiling systems like Xdebug, QcacheGrind and WebGrind are also elucidated.

This paper contains source code which calculates Fibonacci sequence and multiplies 800x800 matrices written in C and PHP programming languages. Results show that whenever there is a need for a mathematical computation, C will be many times faster and that it is much more cost-effective to write such code in C and create a PHP module.

Keywords: benchmarking, PHP internals, PHP modules, scripting language, server side

Sažetak

Ovaj rad predstavlja kako se za ovakve probleme može koristiti programski jezik C, kao PHP modul, te pozvati iz PHP jezika. Cilj ovog ujedno je pokazati kako i kada je bolje, umjesto običnih PHP funkcija, raditi PHP module u programskom jeziku C te prikazati i negativne strane takvoga programiranja. Rad objašnjava i pronalazak „uskog-grla” u aplikacijama pomoću sustava kao što su Xdebug, QcacheGrind i WebGrind.

U radu je prikazan izvorni kod koji računa Fibonacci niz te množi matrice veličine 800x800 u programskim jezicima C i PHP. Dobiveni rezultati pokazuju da je u svim slučajevima gdje je bio potreban matematički izračun, programski jezik C bio višestruko brži te da je u takvim slučajevima isplativije napisati izvorni kod u programskom jeziku C i izraditi PHP modul.

Ključne riječi: mjerenje brzine, PHP moduli
1 Introduction

PHP language is mostly designed for developing web applications [1] and, as such, is not suitable for complex mathematical operations, nor is suitable for systems where speed of such operation executions is important. PHP is mostly used for generating dynamic HTML content [9]. Since PHP is built with C, it is possible to create PHP modules in C and import them in the PHP language [2]. Examples given in this paper are executed and tested on Linux operating system, specifically Debian distribution, but it should work on any system with few or not any changes at all. Qcachegrind and Webgrind are tested with OSX.

PHP source code could be downloaded with versioning control system like Git:

git clone https://github.com/php/php-src.git

2 PHP modules

PHP modules must be inside php-src/ext folder and after executing ext_skel script, the skeleton for your model should be created. After the skeleton is created, phpize prepares the build so it needs to be executed:

php-src/ext/mymod# phpize

Inside mymod.c are some templates for functions which a developer can remove or leave for testing purposes. In this example, the module is called mymod so the test function is called confirm_mymod_compiled and this is the function where the testing is performed. This is a simple function which returns a simple number 100 to the user.

// confirm_mymod_compiled() function

PHP_FUNCTION(confirm_mymod_compiled) {
    RETVAL_LONG(100);
}

// inside shell (compiling and installing)
./configure && make && sudo make install

// module needs to be included inside php.ini
extension=mymod.so

// restarting server (in this example, Apache2)
sudo /etc/init.d/apache2 restart

// calling a function from PHP
<?php
    echo confirm_mymod_compiled(); // 100

2.1 Simple PHP benchmarking

Here is a simple test using Fibonacci Series in PHP and in C, both using recursion.

PHP version:

function fibonacci($f) {
    if ($f < 2) {
        return 1;
    } else {
        return fibonacci($f - 1) + fibonacci($f - 2);
    }
}

C version:

long my_fib(long f) {
    if (f < 2) {
        return 1;
    } else {
        return (my_fib(f - 1) + my_fib(f - 2));
    }
}
// declaring PHP function which will call my_fib()
PHP_FUNCTION(confirm_mymod_compiled)
{
    long f;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "l", &f) == FAILURE) {
        return;
    }
    long r = my_fib(f);
    RETVAL_LONG(r);
}

Testing is quite simple with microtime before and after the code. PHP function reference [3] shows microtime returns the current UNIX timestamp in microseconds:

$startTime = microtime(true);

// your code

$endTime = microtime(true);
$result = $endTime - $startTime;

After executing the script few times, test shows (in average) that the C version is about 80 times faster when calculating 34th Fibonacci number:

2.2. Matrix multiplication

Multiplication of two matrices can be very computationally expensive ($O^3$) [7]. There are also ways to reduce some steps and speed up the computation but they are not part of this paper.

In C language, matrix could be implemented as a 2-dimensional array.

PHP_FUNCTION(run_multiplication)
{
    int n = 800;

    // matrices
    int firstM[n][n], secondM[n][n], resultM[n][n];

    // counters and a temporary sum
    int i, j, k;
    int sum = 0;

    srand(time(NULL));
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            firstM[i][j] = rand() % 100 + 1;
        }
    }

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            secondM[i][j] = rand() % 100 + 1;
        }
    }

    float speedC = 0.06488954;
    float speedPHP = 5.378473997;

    Figure 1 - Benchmarking C and PHP with Fibonacci Series calculated with recursion (lower is better)
Two 800x800 matrices are populated with random numbers between 1 and 100. srand function sets the global seed variable which changes the output of the rand function[8]. Multiplication in this example takes $O^3$:

```c
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0;
        for (k = 0; k < n; k++) {
            sum = sum + firstM[i][k] * secondM[k][j];
        }
        resultM[i][j] = sum;
    }
}
```

Defining matrices in PHP is similar:

```php
$n = 800;
$firstM = $secondM = $resultM = [];
for ($i = 0; $i < $n; $i++) {
    for ($j = 0; $j < $n; $j++) {
        $firstM[$i][$j] = rand() % 100 + 1;
    }
}
for ($i = 0; $i < $n; $i++) {
    for ($j = 0; $j < $n; $j++) {
        $secondM[$i][$j] = rand() % 100 + 1;
    }
}
```

Multiplication in PHP stays the same:

```php
for ($i = 0; $i < $n; $i++) {
    for ($j = 0; $j < $n; $j++) {
        $sum = 0;
        for ($k = 0; $k < $n; $k++) {
            $sum = $sum + $firstM[$i][$k] * $secondM[$k][$j];
        }
        $resultM[$i][$j] = $sum;
    }
}
```

### 2.3. Explanation

Relatively speaking, PHP is not a slow language. The problem is that PHP is an interpreted language. While executing C which is already compiled and directly executed on a processor, PHP language needs to go through some phases like parsing, converting to opcodes, etc. and this is the heart of the problem.

The problem also worth mentioning here is the stack. Whenever a function is called, a new stack is created only for that function and that is what is happening inside the Fibonacci version with recursion. The advantage that compiled languages have here, is the possibility of having a great compiler which can inline the function (like copying the code) instead of calling the function and creating the new stack. Without recursion, results (approximately, C was 110 times better) are better:

![Speed (seconds)](image)

<table>
<thead>
<tr>
<th></th>
<th>Speed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1,50204E-05</td>
</tr>
<tr>
<td>PHP</td>
<td>0,001790047</td>
</tr>
</tbody>
</table>

**Figure 2** - Benchmarking C and PHP with Fibonacci Series calculated without recursion - 1000th Fibonacci number (lower is better)

Example with multiplying matrices has few things which needs to be pointed
out. In the example written in C programming language, matrices are inside a stack (no malloc-family function were called) and the PHP example creates matrices on the heap which is slower [4]. Before calling the PHP example, a script must allow PHP to use 1GB of memory:

```php
ini_set('memory_limit', '1024M');
```

**Figure 3** - matrix multiplication

In this example, C is approximately 103 times faster.

### 3 Benchmarking with Xdebug, Qcachegrind and Webgrind

First step when optimizing PHP applications is profiling. With Xdebug, one can determine bottlenecks in the application. To enable Xdebug profiler, several lines must be added to the php.ini file:

```ini
xdebug.profiler_enable = 1
xdebug.profiler_output_name = xdebug.out.%t
xdebug.profiler_output_dir = /tmp
```

The first line enables the profiler. The second line formats the filename where the profiler data will be saved and the third line tells the profiler what is the default directory for saving the result. One can also add a “trigger” which will only enable/save Xdebug results if there is a parameter in the request:

```ini
xdebug.profiler_enable_trigger = 1
# only enabled if XDEBUG_PROFILE is passed
```

As stated on the Xdebug profiler homepage, Xdebug outputs a profiling information in the form of a cachegrind file. To sum it up, there are options for all major operating systems:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows</td>
<td>QCachegrind or WinCacheGrind</td>
</tr>
<tr>
<td>Linux</td>
<td>KCachegrind</td>
</tr>
<tr>
<td>MacOS</td>
<td>QCachegrind</td>
</tr>
</tbody>
</table>

**Figure 4** - "cachegrind" software for different operating systems

It’s recommended to install Graphviz also so a user could see a call graph. For example, to install QCachegrind on OSX, one could do it with homebrew. First, Qt should be installed:

```bash
brew install qt
```

After installing qt, qcachegrind could be installed:

```bash
brew install qcachegrind --with-graphviz
```

Graphviz installs dot, plain text graph description language. Graphviz installs dot in /usr/local/bin/dot but in order to qcachegrind find dot, it should be in /usr/bin/dot. One can create a symlink for dot[6] like this:

```bash
sudo ln -s /usr/local/bin/dot /usr/bin/dot
```
After installing `qcachegrind`, running it is easy:

```bash
$ qcachegrind
```

After opening and importing generated debug file, `qcachegrind` shows a list of functions called in the request. Generated debug file is in the `xdebug.profiler_output_dir` directory specified in `php.ini`.

![Figure 5 - list of function calls in one request](image)

Figure 5 shows a list of function calls in a single PHP request.

There is also a call graph, generated with Graphviz. Figure 6 shows a generated call graph which shows how and where PHP functions are called. There is also a callee map, shown in Figure 7.

![Figure 6 - call graph](image)

![Figure 7 - callee map](image)

There is a `qcachegrind` alternative, called Webgrind. Webgrind is a web frontend which understands Xdebug profiling information. To install Webgrind, it must be downloaded:

```
wget https://github.com/jokkedk/webgrind/archive/master.zip
```

Unzip the application:

```
unzip master.zip
cd webgrind-master
```

Serve the application with PHP:

```
php -S localhost:1234
```

After opening localhost:1234 and opening your debug files, webgrind should show something similar like in Figure 8.

![Figure 8 - webgrind list of function calls](image)
4 Conclusion

If a developer needs to create a cutting-edge code which needs to be fast and efficient, doing module in C and importing it into PHP is the best way to go. Module in C is much faster but the downside is that you need to know something about PHP internals. If there is a need for creating a module which needs to use heap memory, some part of PHP, maybe module needs to be called while PHP is starting or shutting down, developer needs to understand PHP internals and structures like zvalue[5], understand a lot of macros, deal with memory management and with a lot of low programming concepts.

To find bottlenecks, developers can use Xdebug with Qcachegrind or Webgrind. It’s crucial to find a good development time – execution speed ratio. There is no need to write everything as a module in C but writing only parts which are bottlenecks could give some amazing improvements.

4. References


[4] Paul Gribble, Memory: Stack vs Heap (Summer 2012), 2016, URL: http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html


Authors’ addresses

Alen Šimec, PhD, lecturer
University of Applied Sciences
Vrbik 8, 10 000 Zagreb
alen@tvz.hr

Davor Lozić
University of Applied Sciences
Vrbik 8, 10 000 Zagreb
dlozic@tvz.hr

Lidija Tepeš Golubić, PhD, senior lecturer
University of Applied Sciences
Vrbik 8, 10 000 Zagreb
ltepes2@tvz.hr