# 1 High Performance PHP

George Schlossnagle <george@omniti.com>

## 1.1 Introduction

### 1.1.1 What is Performance?

Performance is in many ways like security, a term so laden with hype and hyperbole that it starts to lack meaning. It is rareto see an application described by its authors as 'low performance', and even the slowest applications tout changelogs with statements like '"improved the speed of foo by 10 percent." The PHP language is a prime example of this. Unlike Perl or Python, which both ship with extensive native-language libraries, PHP still ships with more 'ext' extension libraries (authored in C) than it does PEAR classes (authored in PHP). This phenomenon is not entirely negative -- the extension functions are almost uniformly faster than a native implementation -- but it also means that the implementation details are inaccessible to the majority of PHP users.

This scenaro illustrates a critical trade-off in tuning an application: at a certain point maintainability is sacrificed for performance. Sometimes the trade-off is worth it, and sometimes not , but an analysis of the compromise should kept in mind as we carry on. In Patrick Killelea's *Web Performance Tuning*, he writes regarding tuning:"If you find yourself recoding parts of your application in Assembly, you've probably gone too far.'" Good words to keep in mind.

Tuning can become a morass, bogging down the developer as he seeks to eek out another small gain. For most of us though, performance tuning is not the end goal. We are programming to achieve an end; and a product, no matter how fast, is worthless if it cannot be delivered on time.

All this negativity is not designed to discourage you from tuning your application; on the contrary, adequate performance is a necessary component of any application. This introduction should simply serve as a reminder that performance is just one of many critical components of successful application design and must be balanced with maintainability and delivery scheduling.

So the answer to our initial query is perhaps surprising: Performance is not everything.

### 1.1.2 Approaches to Performance Tuning

People tend to approach performance tuning from one of two angles:those who focus on building applications for performance from the ground-up, and those who focus on meth-

ods to incrementally approach tuning. My personal opinion is that the latter approach is more applicable to people's lives. Most of us have inherited applications that were either too large or too time-consuming to re-engineer from scratch. For those of us saddled with large legacy code bases, finding the code that is most beneficial to optimize is equally important as finding the most efficient optimizations for any given code-block. For those lucky enough to be working from the ground-up on a project, the tools and revelations developed here should provide both a good template for writing new code as well as the skills for retroflecting on their code the first time it hits a bottleneck.

Before tuning application code, the applications run environment should be setup appropriately.. This involves:

- Hardware/Network/OS Setup and Tuning. No matter how fast your code is, an improperly setup server will grind you to your knees. Hardware selection and OS tuning are beyond the scope of this paper, but there are many great resources available to guide you.
- Webserver Configuration. This is also beyond the scope of this paper, but critical nonetheless. Again, there are many great resources for optimally configuring your webserver. See the Appendix for details.
- Database/Data Setup. For database-driven applications, improperly tuned queries or an improperly tuned RDBMS can have severe performance implications. If you have a large or highly transactional database, hire a good DBA or take the time to learn all you can about the idiosyncracies of your particular RDBMS.
- Language Setup. Language setup refers to ways of configuring PHP itself to optimize script performance.
- Application Design. Often the largest improvement to an application comes from improving the overall design of the application itself. Can caching be built into the application? Is their expensive functionality fluff that can be cut out?
- Application Implementation. Once all the lower level details have been dealt with and the application architecture is decided on, we can move onto the main body of this paper which is how to identify and fix bottlenecks in specific sections of code.

## 1.2 Language Setup

Although not the focus of this paper, there are some php configuration options that deserve to be touched on since they can have significant performance impacts.

### 1.2.1 Debug Options

When not actively troubleshooting problems or profiling code, disable debugging options and don’t load profiling extensions. While all attempts have been made to reduce the overhead of these facilities, a degree of slowdown is expected. If you’re not using them, turn them off.

### 1.2.2 Compiler Caches

By far the largest configuration benefit you can achieve is using a compiler cache. The PHP engine is designed so that files are parsed and compiled on every execution. A compiler cache stores the optree generated by the parse/compile stage and caches for use when that file is loaded again. Performance gains of 1000% are not uncommon for otherwise well-tuned scripts.

Some popular compiler caches are

- APC (free and Open Source)
- PHP-Accelerator (free but closed source)
- Zend Accelerator (previously Zend Cache - for-cost and -closed source)

### 1.2.3 Language Optimizers

Language optimizers work by taking the compiled optree for a file and performing optimizations on it. Benchmarks have shown that the benefit of using an optimizer is almost always negligible when compared to the benefit of using a compiler cache.

The only language optimizer currently available is the Zend Optimizer, which is free but closed-source. Further, since the Zend Optimizer prevents the loading of any `zend_extension` other than those produced by Zend Technologies, the advantages of using it are small if you are not also using Zend Accelerator.

Hopefully an Open Source optimzer (or better yet, an optimizer in the stock engine) will materialize….

## 1.3 Application Tuning

Performance tuning is a cyclical process: wash, rinse, repeat until clean. Of course, people's definition of 'clean' varies greatly, as does the definition of ‘high performance.’ Fortunately, the cyclical approach can fit everyone: analyze, tune, repeat until it meets your expectations. So we've broken down our methodology into 4 steps, all critical for successful tuning:

- Set your expectations. If you don't know where you're going, it's hard to know when you've arrived. Too many people approach performance tuning as a universal goal: “I want my application to run faster.” Goals that vague are impossible to successfully achieve. How fast is faster?
- Analyze. Almost all code can be improved upon. The trick is finding the code that needs to be improved. Code that is executed seldom can afford to be even

grossly inefficient. Code that is executed frequently needs to be tuned as highly as possible. First we will identify the scripts run most frequently, then we will use profiling techniques to find the primary causes of the bottleneck so that our tuning is most effective.

- Tune. Now for the exciting part! Fortunately for us, we've already done most of the hard work. Thanks to us doing our homework, we know exactly where we need to look for inefficient code and how fast we need it to be to meet our goals. Here we use knowledge of the Zend Engine and built-in extensions, simple benchmarking techniques, optimization in C, and extra-application tuning to improve the trouble spots.
- Evaluate your results. How did you do? Did you achieve your goals? Were they too conservative or overly ambitious?

### 1.3.1 Setting Goals

The first question we need to answer is “What are our goals? While tuning for tuning’s sake can be educational and fun, we need to know when to call it quits to move on to other projects. For those programming in the corporate or contracting world, setting performance goals is also a way to prove you've achieved your milestones.

Goals primarily need to be achievable. Unachievable goals can be both personally demoralizing and professionally destructive. Goals should also be ambitious. Challenge makes us grow and learn - stretch your abilities by setting goals beyond what you are certain you can achieve but within reason. To keep ourselves honest, goals need to be measurable. The specific nature of the goals really depends on what you are trying to achieve. In a web environment a typical goal is to sustain a certain traffic load, usually measured in requests per second. In a scripting or application environment, we may be focused on overall application runtime or system resource usage.

### 1.3.2 Profiling

Now to get our hands dirty! Profiling is necessary for a multitude of reasons. In a large code base it helps us quickly identify problem code. Also, it is particularly useful in protecting us from our own vanity. No one wants to think that the text macro parser library they wrote is the principal bottleneck in the system. A good profile helps direct your eyes to the right place - even the place you know can't be 'it'.

### 1.3.3 Finding Troublesome Code

The first step is to identify the scripts we want to profile. In stand-alone applications or scripts, this is usually an easy choice. For web applications, there are a few general classes:

- If there are fewer than 10 actively accessed scripts, profile all of them. .
- Sometimes there are Service Level Agreement issues or other political reasons to address certain scripts.
- With a larger set of scripts, using a ranking based off overall resource utilization usage can be effective.

### 1.3.4 Resource Usage Measurement

There are a number of techniques for measuring resource utilization. First, we can take measurements inside all of our scripts by recording their start and exit times; however,this method has a number of drawbacks:

- Maintenance overhead (code needs to be added to every page)
- Only profiles PHP pages (while potentially obvious, this eliminates identification of performance problems outside of PHP)

Another method is to use the execution time logging functionality in Apache's `mod_log_config`. Using this we can estimate the total resource usage for all scripts by summing up their individual execution times over a fixed interval. This method suffers from the seconds-only accuracy of the `mod_log_config` timers.

Fortunately, Apache is easy to extend, so with little additional work we can add high-resolution timers to mod_log_config. See the Appendix for links to more information on this.

### 1.3.5 Choosing a Profiler

There are a number of profilers for PHP. APD and DBG are two open source Zend Extension debuggers that provide profile information. People have also written profile wrapper classes in PHP (Benchmark_Profiler in PEAR, for example).

I would recommend staying away from the latter - the overhead of profilers written in PHP is very high. Extension profilers use one of two facilities in the Zend Engine to provide profile information. DBG (and older versions of APD) use the `begin_func_call` and end_func_call hooks in the Zend Engine. These work by interjecting additional opcodes before and after every function call. The other method is to wrap the calls to zend_execute to track entrance and exit to functions. While the former method works better for interactive debuggers, wrapping the execute call is the least obstructive method for tracking function calls, and is the only way that easily allows for correct representation of nested function calls.

I will use APD in my examples here, but the techniques can be extrapolated to the profiler you prefer.

### 1.3.6 Installing and Using APD

APD is available through PEAR, the PHP Extension and Application Repository. To install APD simply download the software and follow the directions in the INSTALL file. After you have it compiled and installed, you need simply to add the following lines to your php.ini file

```
zend_extension = /absolute/path/to/php_apd.so

apd.dumpdir = /absolute/path/to/trace/directory
```

apd.dumpdir is the location that your profile files will be dumped to.

A simple example of how to use APD is:

```
//simple.php

<?

    apd_set_pprof_trace();  // enable profiling for this script

    function yell($var) {

        print strtoupper($var);

    }

    function hello($name) {

        return "Hello $name\n";

    }

    yell(hello("George"));

?>
```

This will dump a file `pprof.<pid>` our dumpdir. We then run the profiler processor on that file to generate a profile:

```
22:55:10(george@warhammer)[~/> pprofp -T /php/profiles/pprof.14374
main
hello
yell
  strtoupper

Trace for simple.php
Total Elapsed Time =    0.00
Total System Time  =    0.00
Total User Time    =    0.00


         Real         User        System             secs/    cumm
%Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls    call
s/call Name
--------------------------------------------------------------------
------
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.00
strtoupper
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.00
yell
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.00
hello
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.00
main
```

The profile starts out with an indenting call tree (toggleable with the –T flag), then goes into a summary statistics section -- which is very uninteresting in this example! Complete syntax for the profiler processor is:

```
pprofp <flags> <trace file>
    Sort options
    -a          Sort by alphabetic names of subroutines.
    -l          Sort by number of calls to subroutines
    -r          Sort by real time spent in subroutines.
    -R          Sort by real time spent in subroutines (inclusive
of child calls).
    -s          Sort by system time spent in subroutines.
    -S          Sort by 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.
    -f         Display filename/line-numbers for function calls
    -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.
```

### 1.3.7 Profiling Fast Running Code

APD uses the times() system call, which has centisecond accuracy. For particularly fast running code, this can be insufficient to adequately diagnose bottlenecks. While it may seem that code that runs this fast does not need to be optimized, if executed frequently enough, critical bottlenecks can still be incurred.

In these situations, using a testing harness can be very useful. A harness can be as simple as:

```
<?

    include 'foo.php';

    apd_set_pprof_trace();
```

```
    for($i=0;$i<1000;$i++) {

         foo();

}

?>
```

Here `foo()` is a function which calls the entire code section we are interested in profiling. This can be anything from a simple block, a top-level function, or a function call which wraps an entire web page.

Later, when we discuss benchmarking techniques, we will talk about the PEAR class Benchmark_Iterate. It's important to mention here though, that the Benchmark_Iterate framework can skew your profiling results, so while it is good for benchmarking, it is best to use something simpler for a profiling framework.

### 1.3.8 Where to Profile

Applications which use dynamic data sets (such as database-driven applications) can exibit radically different behavior depending on the data which is passed to them. In the performance tuning arena, this often exhibits itself as an application which behaves fine in a development environment, but has issues when running in production.

This raises the question of where profiling should be done: in development or in production?

Profiling code in development has a distinct set of advantages:

- The environment is more in the control of the user. Code changes or profiling techniques which might adversely affect a production environment can more easily be attempted.
- In environments with large clusters of servers or strict change control, developers may have restricted or prohibited access to production machines.

The major advantage of profiling live production code is that some bottlenecks are elusive or only manifest themselves on production data. That advantage aside, the best way to proceed is to profile in development as long as produces positive results.

#### An Example: Filtering Javascript

We are troubleshooting a problem with a personal page application. The application stores user personal page text in a database, fetches it on request and filters out any po-

tentially malicious or obscene content. Since our filter rules may change as new vulnerabilities are discovered (and because we prefer not to delete user content, but just not show it), we filter content outbound, instead of as content is uploaded.

Reports come in from the service representatives that personal page loads are extremely slow, so we pull out our profiling toolkit to try and diagnose the problem.

First we profile the page in question on a test page in our development environment:

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

12 database queries to generate a single personal page seems excessive, so a natural first suspect is that we are experiencing a database bottleneck. An execution plan for the queries looks optimal though (all using indexes, minimal database buffer gets per execution), so we need to move our profiling into production.

A profile in production on our test page produces similar results. Somewhat slower, but still not as bad as has been described. What is going on? We move our profiling code fully into production and access some real user pages through it.

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

```
  0.0  0.04  0.21   0.01  0.16   0.01  0.03    37   0.0005    0.01
include
```

Wow! 9 seconds in preg_match – that is certainly a problem. Next we generate a call graph for the profile, to identify where the matches are coming from

```
[~]>pprofp –f –T /php/profiles/pprof.4231

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

The associated code block is:

```
function detectUnsafeHTML($html)

    {

        global $UNSAFE_HTML;

        foreach ($UNSAFE_HTML as $rule) {

            if(preg_match($rule, $html)) {

                return -1;

            }

        }

        return 0;

    }
```

$UNSAFE_HTML is an array of 'unsafe' tags or profanity. Since the foreach loop processes them in order, we can even count through the array to find the particularly slow calls.

```
…
```

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

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

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

…
```

These regular expressions run in `O(n*m)`, where `n` is the number of `'<'s`, and `m` is the number of `'onerror's`, etc.

With a simple regular expression it is impossible to ascertain whether the 'onload' is inside a set of html tags, so we can eliminate the matching of the leading '<' and radically improve the efficiency of the match:

```
…

$UNSAFE_HTML[] = !onload\s*=!is";

$UNSAFE_HTML[] = !onunload\s*=!is";

$UNSAFE_HTML[] = !onerror\s*=!is";

…
```

Running with this change, we have:

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

Much better! The problem was that our test page was too short and did not adequately represent some of the longer user templates. Since the offending regular expressions were quadratic for pages with javascript on them, the longer pages were much worse than our test case would have indicated.

Bad regular expressions often exhibit their 'true colors' only under certain data, so be careful. These regular expressions would likely have never caused a problem if used only on tiny text strings. Only when applied to a multi-kilobyte html string with javascript do they get ugly.

## 1.4 Benchmarking

Performing benchmarks yourself acquaints you with the nuances of how different functions perform and is easy to do. Build a testing harness to allow yourself to easily employ variations when trying to optimize code. For most purposes the Sebastian Bergmann's Benchmark classes from PEAR provide a great framework for setting up tests. Here's how we might use it to test our my_max() function versus the builtin max():

```
require_once 'Benchmark/Iterate.php'

$benchmark = new Benchmark_Iterate;



function gen_array($size) {

    $array = array();

    for( $j =0; $j < $size; $j++) {

        array_push($array, rand());

    }

    return $array;

}
```

```
    function my_max(&$array) {

        $max = $array[0];

        foreach( $array as $el ) {

            if ( $el > $max ) {

                $max = $el;

            }

        }

        return $max;

    }



    srand();

    foreach (array(10, 100, 1000, 10000) as $size) {

        foreach (array('my_max', 'max') as $func ) {

            $array = gen_array($size);

            $benchmark->run(1000, $func, $array);

            $result = $benchmark->get();

            print "$func run on datasize $size:
".$result['mean']."\n";

        }

    }
```

Be aware that Benchmark_Iterate calls gettimeofday() at the beginning and end of every function, so with functions that have a very short run time (for instance, when comparing various builtin functions like strncasecmp and a pcre regex), the overhead of calling Benchmarks internal timer routines may well bias your results. In situations like this, it is better to write your own quick testing suite which times outside the execution loop and produces an average from that.

Also, be careful when benchmarking functions that modify arguments passed by reference, this may unintentionally skew your data. For instance, when benchmarking our expand_macro() family of functions, we will want to modify them all to copy the text into a private variable (or change them all to pass by value) for the purpose of the test, otherwise every iteration after the first will have nothing to expand.

### 1.4.1 Internal Functions versus Userspace Functions

PHP has a wealth of builtin functions available to the end-user. Variety is a good thing, but given that there are many ways to do things, some ways are better than others. The general rule is, if there is a builtin function to do something, it will be faster than auser-defined function, especially when it avoids looping in userspace. Our previous example comparing the internal 'max' function and a userspace implementation exhibits this nicely.

One place where large performance gains often can be gleaned is in the manipulation of and iteration through arrays, especially when applying builtin functions to their elements. As an exercise, if you have an array of text strings, benchmark the difference between

```
$size = count($array);

for($i=0; $i<$size; $i++){

    $array[$i] = strtolower($array[$i]);

}
```

and

```
    $array = array_map('strtolower', $array);
```

While this example is somewhat contrived, you can use a similar logic in clever ways to avoid costly iterative loops. Lets say that you are selecting a single field from a database and you want to build an array from the result set. You might do something like:

```
$rows = ocifetchstatement($stmt, $rows);

foreach ($rows as $row) {

    $new_rows[] = $row[0];

}
```

That's slow. The following code is much faster, albeit harder to understand at first glance:

```
$rows = ocifetchstatement($stmt, $rows);

$new_rows = array_map('reset', $rows);
```

Just so that we are reminded that tuning can be subtle and tricky, this benefit can really only be realized when using builtin (or C extension) functions. When using userspace functions, the overhead of making the function call on each element obscures any benefit from avoiding the loop.

## 1.5 Implementing Functions in C

At what point does it make sense to start porting your own functions to C, to give them the performance edge of builtin functions? In many ways, this should be a last resort. C extensions require more proficient developers to maintain, are more difficult to port between php versions, and are less flexible.

If you look through the archives of the php-dev mailing list, you will see numerous attempted contributions of extension functions that implement logic that is easy to accomplish in php. Many of these have been rejected from the main distribution on the grounds that they can be implemented rather simply in php using existing functionality.

These functions may not be very general purpose (for example, a function to generate arbitrary length alphanumeric strings) and in 99.9 percent of the cases should be written in php, except for the rare application for which this functionality is a major bottleneck, in which case reimplementing in C may be worthwhile.

### 1.5.1 Implementation Checklist

My personal checklist for reimplementation is:

1. Does it really need to be that fast? - You will have to recompile it everywhere you need it and make sure it stays current between php versions.–Is it worth it? If the function in question is not causing a major bottleneck in an important script, it's better to retain the flexibility and maintainability of leaving it in php.
2. Am I going to need to change it? - Is the function very static, supplying a very standard and well-defined purpose? For example, sample statistic functions have well-defined, unchanging mathematical definitions and will have good re-usability, a function to parse a proprietary cookie format will probably need to be changed often.
3. Is there no way around it? - Could the need for this function be removed by a change in application logic/design somewhere? Is it easier to modify the app not to need it or to maintain the C extension?

### 1.5.2 Inline_C

One alternative to writing full C extensions is to use the Inline_C class from PEAR to define functions in C inline in your php scripts. Inline_C works by using a pre-built build environment to allow for on-the-fly compilation of functions and loading the cached shared objects with dl(). Excepting the overhead of the dl() call, Inline_C provides a convenient way to build simple extensions.

### 1.5.3 Examples

#### Fibonacci Sequences

Recursive user functions are expensive in PHP, and a standard recursive function is a Fibonacci sequence generator.

```
<?

    function fib($n) {

        if ($n == 1 || $n == 0) {

            return 1;

        } else {

            return fib($n - 1) + fib($n - 2);

        }
```

```
    }

    printf("Fib(%d) = %d\n", $argv[1], fib($argv[1]));



?>
```

A replacement for this in C would be:

```
<?

require_once("Inline_C.php");

$function = <<<EOF

long int _fib(long num)

{

        if(num == 0 || num == 1) {

            return 1;

        } else {

            return _fib(num - 1) + _fib(num - 2);

        }

}

PHP_FUNCTION(fib)

{

    long i;
```

```
    if (zend_parse_parameters(ZEND_NUM_ARGS()  TSRMLS_CC, "l", &i)
== FAILURE)

         return;

    RETURN_LONG(_fib(i));

}



EOF;



$inline = new Inline_C;

$inline->add_code($function);

$inline->compile();

printf("Fib(%d) = %d\n", $argv[1], fib($argv[1]));

?>
```

This example actuallydoes not meet our condition 3. With a bit of cleverness, we can radically improve our function performance, while leaving it all in userspace code. We approach it by caching the values of sequence elements we've seen already:

```
<?

    $cache = array();

    function fib($n) {

        global $cache;

        if ($n == 1 || $n == 0) {
```

```
            return 1;

        } else if ($cache[$n]){

            return $cache[$n];

        } else {

            $cache[$n] = fib($n - 1) + fib($n - 2);

            return $cache[$n];

        }

    }

    printf("Fib(%d) = %d\n", $argv[1], fib($argv[1]));



?>
```

This same trick can of course be applied to our C implementation (which would then be faster than our PHP implementation), but the pure PHP implementation may well be fast enough for our needs.

### RC4 Encryption

RC4 is a symmetric stream cipher designed by Ken Rivest for RSA Data Security (now RSA Security). It is a variable key-size stream cipher with byte-oriented operations. While not the most advanced stream cipher around, RC4 is extremely fast (much faster than RC5), which is why it is used in SSL and makes it a candidate for encrypting large amounts of data. PEAR contains a RC4 implementation in pure PHP (Crypto_Rc4), but the implementation is somewhat slow if used extensively. This makes it a great candidate for reimplementation in C[1]:

```
<?
```

[1] Implementation largely adapted from http://www.cypherspace.org/~adam/rsa/rc4.c

```
require_once("Inline_C.php");

$rc4 = <<<EOF

#define swap_byte(x,y) t = *(x); *(x) = *(y); *(y) = t

void _rc4(unsigned char *buffer, int buflen, unsigned char *key,
int k_len)

{

  unsigned char x =0, y = 0, index1 = 0, index2 = 0, i, j;

  unsigned char k[256], s[256];

  unsigned char xorIndex, t;

  char digit[5];

  char seed[256];

  short counter;

  int n;



  if(k_len&1) {

    k_len--;

    key[k_len] = '\0';

  }

  k_len /= 2;

  strcpy(digit, "AA");
```

```
digit[4]='\0';

for(i=0;i<k_len;i++)

{

  digit[2] = key[1*2];

  digit[3] = key[i*2+1];

  sscanf(digit,"%x",&seed[i]);

}

for(counter = 0; counter < 256; counter++)

  s[counter] = counter;

for(counter = 0; counter < 256; counter++)

{

  index2 = (k[index1] + s[counter] + index2) % 256;

  swap_byte(&s[counter], &s[index2]);

  index1 = (index1 + 1) % k_len;

}

for(counter = 0; counter < buflen; counter++)

{

  x = (x + 1) % 256;

  y = (s[x] + y) % 256;

  swap_byte(&s[x], &s[y]);
```

```
    xorIndex = (s[x] + s[y]) % 256;

    buffer[counter] ^= s[xorIndex];

  }

}









PHP_FUNCTION(rc4)

{

    char *data, *key;

    int datalen, keylen;

    if (zend_parse_parameters(ZEND_NUM_ARGS()  TSRMLS_CC, "ss",
&data, &datalen, &key, &keylen) == FAILURE)

        return;

     _rc4(data, datalen, key, keylen);

     RETURN_STRINGL(data, datalen, 1);



}



EOF;
```

```
$inline = new Inline_C;

$inline->add_code($rc4);

$inline->compile();



$encode = rc4("hello world", "d3adb33f");

$plaintext =    rc4($encode, "d3adb33f");

print "Plaintext: $plaintext\n";



?>
```

## 1.6 Summary

## 1.7 Appendix

### 1.7.1 Cache Compilers

APD http://pear.php.net/APD
DBG http://dd.cron.ru/dbg/
Zend Accelerator http://www.zend.com/store/products/zend-accelerator.php