PHP ACCELERATION IN THE CLOUD:
Caching PHP Virtual Machine Opcodes for Reuse in a Secure, Multi-User, Enterprise Programming Environment

by

Leif Wendal Wickland

A project submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

November 2010

APPROVAL

of a thesis submitted by

Leif Wendal Wickland

This project has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citation, bibliographic style, and consistency and is ready for submission to the Division of Graduate Education.

Dr. Rocky Ross

Approved for the Department of Computer Science

Dr. John Paxton

Approved for the Division of Graduate Education

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

In presenting this project in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Leif Wendal Wickland

November 2010

DEDICATION and ACKNOWLEDGEMENTS

To my wife and daughter who love me more than I deserve. -LW

I would like to acknowledge Erich Hannan for giving me the opportunity to wait for the other shoe to drop. I would also like to acknowledge Dr. Ross for making me believe this was possible.

TABLE OF CONTENTS

ABSTRACT

In this paper we discuss a project to accelerate the execution of PHP scripts. The execution of a PHP script consists of two major phases. First, the PHP script is scanned, parsed, and translated into opcodes and into functions and classes that reference opcodes. Second the opcodes are interpreted by the PHP virtual machine. Over the years, a number of attempts have been made to improve the performance of PHP through caching, that is, by storing and reusing the output of the first phase. To the best of the author's knowledge, all of these attempts have crippling limitations for application to the current project: they either only work in a mod_php or fast CGI environment because they depend on shared memory with a daemon process, or they do not support loading cached opcodes from a different file system path than the one at which the cache file was created. This project is targeted at delivering the performance benefits of opcode caching to a CGI environment in which many isolated tenants in a common infrastructure execute copies of the same cached PHP opcodes. Rather than depending on shared memory in primary storage, this project utilizes files in secondary storage to cache opcodes.

Introduction

PHP suffers from the same malady as many other interpreted scripting languages: execution of PHP programs is notoriously inefficient and, hence, slow. The execution of a PHP script consists of two major phases. First, the PHP script is scanned and parsed into opcodes, and into functions and classes that reference opcodes. Second, the opcodes are interpreted by the PHP virtual machine. Both of these phases are performed by default each time a PHP page is accessed by a client, regardless of whether the page has changed or not, resulting in the regeneration of the same opcodes by the first phase.

This project attempts to improve the performance of the PHP runtime environment through elimination of the first phase by caching and reusing the opcodes generated for a PHP page, until the page is changed. The project described here is intended to provide performance improvements by saving to a file on disk (caching) and reusing the PHP opcode file array output by the first phase of the PHP runtime environment. The project has two requirements that preclude the use of existing systems that use caching to improve the performance of PHP: (1) the caching technique cannot depend on shared memory in primary storage, and (2) a single cached opcode file must be viable when used from different file system locations.

Over the years, a number of projects have attempted to improve the performance of PHP through caching, that is, by storing and reusing the opcode file output by the of the first phase. APC (http://www.php.net/manual/en/intro.apc.php), Zend Optimizer (http://www.zend.com/en/products/guard/zend-optimizer), and XCache (http://xcache.lighttpd.net/) are examples of such PHP accelerators. These projects rely

on shared main memory for the cache, in which a daemon process is used to cache compiled opcodes. As a result, they cannot be used in a CGI environment. In the CGI model, each HTTP request is served by a new process; if at some moment no requests were being served, there would be no owner process for the shared memory and the operating system would free it. (In Unix environments the operating system runs in the context of a process.) A possible solution to this shared memory problem for CGI environments is embodied in another PHP accelerator, eAccelerator (http://eaccelerator.net/ticket/3), which supports caching opcodes to disk rather than shared primary memory. However, the eAccelerator is not able to use cached files that have been written for one file system path to be used in a different path, a requirement of the project described here.

Other reasons why earlier attempts to store and reuse the opcode file generated by the PHP processor are not applicable to this project include stricter security and backwards compatibility requirements. In the vast majority of cases, PHP is used as an Apache Web server module or as a fast CGI daemon (http://www.fastcgi.com/). Few users of PHP today run it in a plain CGI environment because of the relatively poor performance afforded by that option. The environment in which this project is used requires that PHP be run as a CGI. The resulting performance hit is taken in trade for a gain in better perceived robustness and security by isolating each PHP execution instance. In this environment, an errant PHP process cannot harm other PHP processes by crashing and taking down a daemon or through corrupting shared memory. Additionally, the CGI model allows new versions of PHP to be introduced to the Web

server without affecting PHP scripts that were written for older versions of PHP. In this project's environment, customers can choose which version of a software package to run. Various versions of the software require different versions of PHP. Each customer's installation of the product contains a link to only the version of PHP that their version of the product requires.

This project was thus conceived as an effort to achieve some of the performance benefits of using a PHP accelerator while maintaining the required isolation of executing PHP scripts offered by CGI. It accomplishes this objective by leveraging parts of the open-source APC PHP accelerator. Specifically, it co-opts the parts of that project that can copy internal PHP structures to shared memory from the PHP runtime and back again. Those pieces are used to write cached opcodes serially to memory and then later to disk. Finally, at runtime, the files are loaded from disk, their contents transformed to make them usable, and the imported pieces of APC are used to copy the opcodes back into the PHP runtime.

An Overview of PHP

As further background for the paper we provide an overview of PHP.

What Is PHP?

PHP is a programming environment most often used in the dynamic creation of Web pages. Major Internet companies, such as Facebook, Wikipedia, and Yahoo use it for the presentation layer of their hugely popular websites.

Like many languages introduced in the last fifteen years, PHP is really composed of three parts. The first part (and the part people most commonly associate with PHP) is a *dynamic programming language*. As a programming language, PHP was heavily influenced by Perl, due to its original role as a replacement for the inventor's collection of Perl scripts, which he used for generating his website. PHP's major difference from Perl is its focus on literal output. By default, PHP programs output the content of HTML source files, enhanced with additional HTML markup resulting from the execution of embedded PHP code. Special tagging is required in these HTML files to demarcate the PHP code. More recent versions of the language have added object oriented features, taking inspiration from C++ and Java.

The second part of PHP is its *standard library of functions and classes*. PHP's libraries have grown quite organically and, thus, are littered with inconsistencies and duplications. Efforts are ongoing to address some of these problems, such as the planned removal of an older family of regular expression functions in PHP 6.

The final part of PHP is the *runtime environment* (or *runtime engine*, or just *runtime*), which consists of a front-end compiler that generates "opcodes" from the PHP source program, and a virtual machine that interprets the opcodes. The standard PHP distribution has seen three major versions of this engine. The two most recent have been driven by an Israeli company, Zend Technologies. Other implementations of the PHP runtime engine exist. For example, the IronPHP project (http://www.ohloh.net/p/ironphp) translates PHP to run inside of the Microsoft .NET environment. Also, Facebook has released the HipHop for PHP project

(http://github.com/facebook/hiphop-php/wiki), which transforms PHP into C++, which is then compiled and executed inside of a reimplementation of the PHP runtime.

Although its syntax is geared toward generating Web pages, PHP is a sufficiently general purpose language that it can be used for other tasks. Indeed, the benefits this project offers could also extend beyond the context of the Web. However, we maintain focus on the Web environment in this paper.

On Compiled versus Interpreted Languages

PHP is an example of a programming language that is intended to be interpreted rather than compiled to native machine code. Other languages, such as C and C++, are intended to be compiled into native machine language that can be run directly on the target processor. A compiler for this second class of languages is generally composed of two parts: a front end that scans, parses, and converts the source language into some intermediate form that reflects the same meaning as the input source program, and a back end that translates the intermediate form program into an equivalent program in the native language of the target computer. Since the computer can then execute the compiled native machine language instructions of the translated program directly, no separate "virtual machine" or "engine" is required.

In languages intended to be interpreted, such as PHP, Java, and JavaScript, a program is (generally) not translated into the native language of the computer on which it is intended to run. Instead, just the front end of the compiling process is applied to programs in these languages. That is, translation of the source code stops after scanning, parsing, and translation has produced a desired intermediate or virtual machine form

rather than native code for some targeted real processor. Thus, this intermediate or virtual machine form must be interpreted (or executed) by a virtual machine, which is a program written in a language like C or C++ that itself has been compiled all the way into native machine language. In other words, the virtual machine executes directly on the target computer. It takes as input the intermediate or virtual machine language form of the PHP, Java, or JavaScript program and analyzes and carries out those instructions. Except for the special case of "just-in-time compiling," the virtual machine does not further translate the intermediate code generated by the front end compiler into machine language (a relatively common misconception).

Within the category of interpreted languages there is a further dichotomy between languages in which programs are intended to be "front end" compiled once to some virtual machine form that is interpreted each time the program is to be run (that is, each time the program is to be run, the virtual machine for that language is invoked on the virtual machine code for that program; the front end compilation to virtual machine code form does not need to be performed again on the program source file unless the source program is altered), and languages in which programs are typically intended to be "front-end" compiled to some virtual machine form each time the program is invoked.

Java falls into the former category; its source programs are usually compiled to an intermediate form (called byte codes) and written to disk to be later read from disk and interpreted by the Java Virtual Machine each time the program is to be run. In this category the acts of compiling and executing are usually carried out by separate processes.

In the latter category, languages such as PHP and JavaScript are usually recompiled into their target virtual machine code with each invocation of the source program (i.e., each time the program is to be run, the source program is invoked--rather than some compiled native machine language or virtual machine language version of the source program--hence the entire process of front-end compiling must be repeated on each invocation before interpretation occurs). The compiler typically runs in the same process as the virtual machine. In both subcategories, compiling consists of scanning, parsing, and generation of virtual machine code, all typical parts of the front end of a compiler. Scanning translates the sequence of characters in the source file into a series of tokens. Parsing consumes those tokens, ensuring that they represent syntactically correct statements in the source programming language. The parser then invokes semantic routines to translate syntactically correct source code into a form that can be interpreted by the attendant virtual machine, or engine. In the case of PHP, the output of the front end compilation process is called *opcodes*.

This project moves PHP from the category that requires full front-end compiling followed by interpretation of the generated opcodes each time a particular PHP program is executed into the category of interpreted languages that are front-end compiled once into some virtual machine form that is then interpreted many times (until the original source code PHP program is altered and hence must be front-end compiled again).

PHP's Organic Growth

Rasmus Lerdorf created PHP in the mid-1990s to scratch his itch for developing a better way to create his personal homepage. He released it as open source in 1995. Over

the next couple of years it became a moderately popular tool for developing sites on the early World Wide Web (http://www.php.net/manual/en/history.php.php).

During this time, Andi Gutmans and Zeev Suraski tried using an early version of PHP to build an eCommerce site, but found its performance insufficient for their needs (http://www.php.net/manual/en/history.php.php). They thus launched into the first major rewrite of the PHP runtime. In 1997, PHP version 3 was released with Andi and Zeev's Zend engine at its heart. Although PHP 3 offered faster execution and added new functionality, it was assiduously backward compatible with the previous version. Subsequent versions have maintained this goal, with the exception of recent efforts to remove redundancy and to eliminate features that pose inherent security risks.

Partially owing to its history as a personal project and as a community-directed open source project, the PHP language and runtime have never been formally defined. No one has penned a description of the PHP language, libraries, and runtime and submitted them to a standards body for codification. For example, no grammar of the PHP language has apparently been published; the most reliable "formal" definition of PHP can be gleaned from the token and syntax definitions supplied as inputs to the Flex scanner (http://svn.php.net/viewvc/php/php-src/trunk/Zend/zend_language_scanner.l?view=markup) and Bison parser (http://svn.php.net/viewvc/php/php-src/trunk/Zend/zend_language_parser.y?view=markup), respectively, which in some implementations are used for the automatic generation of a front-end compiler for PHP. If

one wants to understand the internals of PHP, reading the source code is one of the better options.

An Example PHP Program (or Script)

Suppose that the file thatPage.php contains the following:

```
<html>
  <head></head>
  <body>
    <b>Agent:</b> So who do you think you are, anyhow?
    <br />
    <?php
    $firstName = $_REQUEST['firstName'];
    echo "<b>$firstName:</b> I am $firstName, ";
    echo "but my people call me ";
    echo "{$_REQUEST['nickname']}.";
    ?>
  </body>
</html>
```

The .php extension on the file name implies that this file is a PHP script that will most likely contain embedded PHP scripting code throughout, each instance demarcated with the

```
<?php ... ?>
```

tag pair. (Alternatively, such a file can be viewed as an HTML page with interlaced PHP scripting code.) Thus, when a file with the .php extension is requested by a client, the server will first invoke the PHP runtime on the file, which in turn processes the file, as described earlier. The result is the output of a "pure" HTML page which contains the original HTML content echoed verbatim along with new HTML content injected by processing of the PHP scripting code.

In this particular example, the embedded PHP scripting code is expecting two parameters to be provided in the requesting URL, one for *firstName* and one for *nickname*. Thus, if a request were made for that page with the URL http://an.example/thatPage.php?firstNameame=Neo&nickname=The%20One, then a Web page like the following will be created by PHP runtime engine and delivered to the requesting client by the associated server software:

```
<html>
  <head></head>
  <body>
    <b>Agent:</b> So who do you think you are, anyhow?
    <br />
    <b>Neo:</b> I am Neo, but my people call me The One.
  </body>
</html>
```

When processed by the client side browser, the screen will display something like:

Agent: So who do you think you are, anyhow?
Neo: I am Neo, but my people call me The One.

In summary, this example illustrates that PHP is a dual mode language, defaulting to literal output. PHP code is embedded within the HTML tags of a "regular" Web page, and must be demarcated with special php tags. The code proper bears a similarity to Perl and Perl's ancestor, the Unix shell, with variables denoted by a leading dollar sign. PHP also owes the interpolation of variables in strings (e.g. parsing "Hi, $firstName!" as if it had been written as the concatenation of a literal string to a variable value) to this heritage. PHP's emphasis on the Web is demonstrated by the special super-global

$_REQUEST variable, which contains all parameters passed to PHP in cookies, post data, and the URL.

High Level View of PHP Execution

As a prelude to digging into PHP internals, an overview of the execution of a PHP file provides a framework for discussing what happens behind the scenes. We refer to the following flowchart for this purpose.

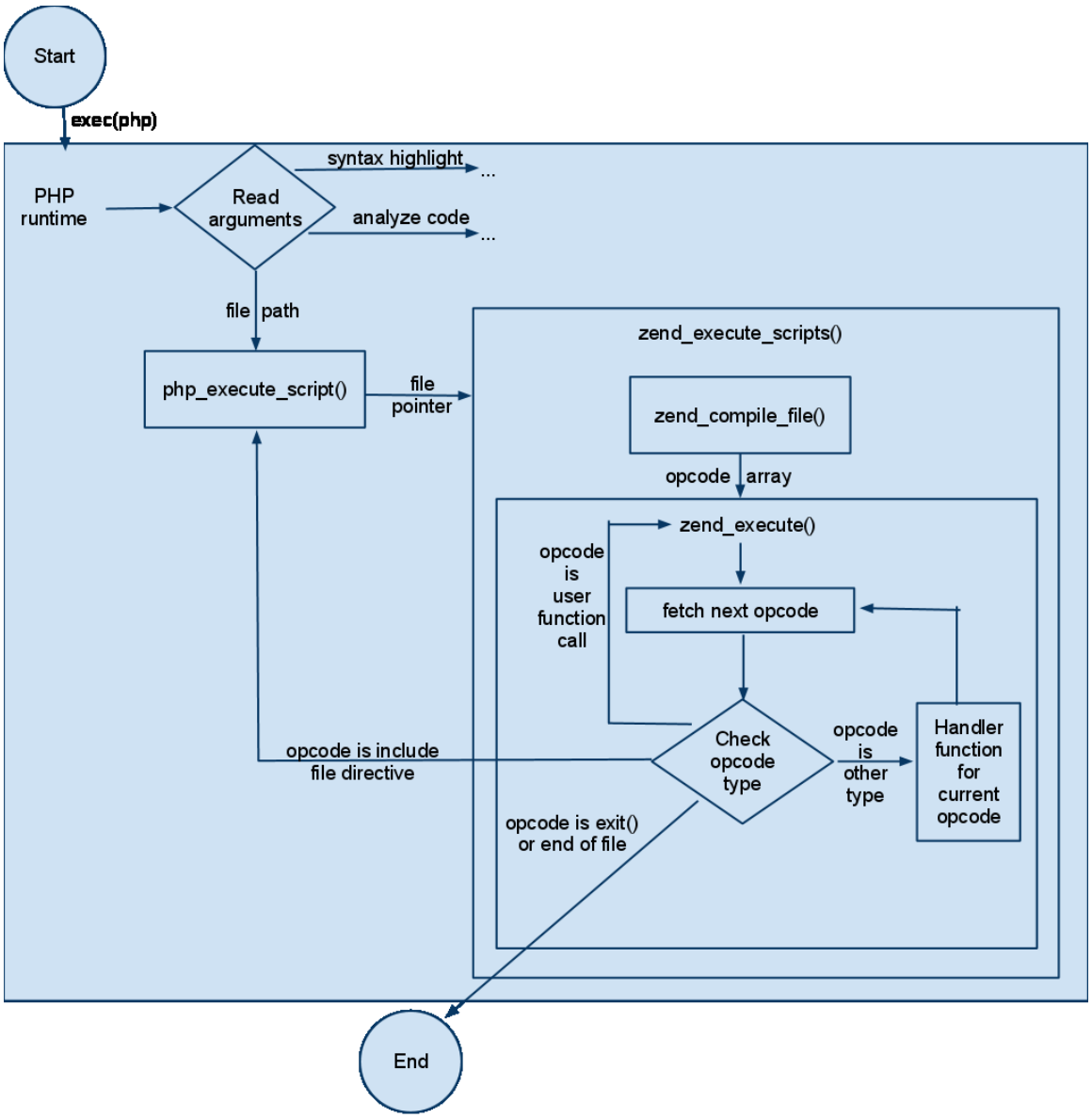

Figure 1 High-Level View of PHP Execution

1. An external process starts the PHP runtime. (It is important to note that the PHP runtime was written in C and has been compiled to the native instruction set for the computer on which it is executing.)

2. The runtime determines what it has been asked to do. The obvious choice is to execute a PHP script. However, there are a surprising number of other things it can do, such as analyze source code or output a syntax-highlighted copy of the code.
3. In the case that a PHP script is to be processed, the PHP runtime starts the Zend Engine with the PHP script file as input.
4. The Zend Engine scans the file, producing a stream of tokens.
5. The Zend Engine then parses the token stream to determine whether the token stream represents syntactically correct source code; if so the parser invokes semantic translation code to generate opcode versions of source code statements in the form of an opcode array, as well as (possibly) functions and classes.
6. The Zend Engine then interprets the opcodes in the opcode array in sequence; interpretation of an opcode can cause a call to another PHP file, which in turn would result in a recursive call to step 3.
7. At the end of execution, PHP tears down the functions and classes that were loaded during the run, preparing to shut down or clean up in preparation for processing of another script.

The heart of the execution phase of the Zend Engine is a loop inside of zend_execute(). At each step in the loop, the Engine evaluates (interprets) the opcode at the current position in the opcode array, carrying out the operation indicated by that opcode. In most cases, the engine then moves to the opcode in the subsequent array

element. However, if the current opcode represents a call to a user-defined function, for example, zend_execute is called (recursively) to handle the processing of that function.

Tracing Processing of a Very Simple PHP Program

Tracing the execution of a PHP page (script) containing only literal output (i.e., no php tags) should provide a vehicle for a deeper, top-to-bottom understanding of PHP processing. For this exercise, the content of the program is not important, so, for the purpose of illustration, the following will be used:

```
Hello, World
```

From the discussion and diagram in the previous section it should be clear that four steps comprise the processing of a PHP program.

1. Scanning the source code into a stream of tokens
2. Parsing the token stream for syntactic correctness.
3. Translating syntactically correct PHP statements into opcodes that are stored in an opcode array
4. Interpreting the opcodes in the opcode array.

Scanning

PHP uses a Flex-generated scanner to scan scripts. The Flex generated scanner is a C function that is incorporated into the Zend engine. The Flex source of that scanner can be found at http://svn.php.net/viewvc/php/php-src/trunk/Zend/zend_language_scanner.l?view=markup.

When the scanner (which is essentially a finite state automaton that is constructed from the regular expressions that make up the input to the Flex scanner generator) scans a script file, it begins in the <INITIAL> state. It then searches the input stream for the next '<', the less-than character. If it finds one, it examines the following few characters to determine if the sequence represents a tag beginning a PHP code block (e.g., "<?php"). If such a tag is found, its location is noted. If no such tag is found, the end of input is recorded. Control then drops to the end of this particular scan, where a copy of the characters from the starting point of this scan to either the "<?php" tag or to the end of input (whichever was encountered first). Then the token T_INLINE_HTML is returned with the accumulated string as an argument.

```
<INITIAL>{ANY_CHAR} {
if (YYCURSOR > YYLIMIT) {
return 0;
}

inline_char_handler:

while (1) {
YYCTYPE *ptr = memchr(YYCURSOR, '<', YYLIMIT - YYCURSOR);

YYCURSOR = ptr ? ptr + 1 : YYLIMIT;

if (YYCURSOR < YYLIMIT) {
<INITIAL>{ANY_CHAR} {
    if (YYCURSOR > YYLIMIT) {
    return 0;
  }

  inline_char_handler:

  while (1) {
    YYCTYPE *ptr = memchr(YYCURSOR, '<',
      YYLIMIT - YYCURSOR);
```

```
    YYCURSOR = ptr ? ptr + 1 : YYLIMIT;

    if (YYCURSOR < YYLIMIT) {
      switch (*YYCURSOR) {
      case '?':
        if (CG(short_tags) ||
            !strncasecmp(YYCURSOR + 1, "php", 3)) {
          /* Assume [ \t\n\r] follows "php" */
          break;
        }
        continue;
      case '%':
        if (CG(asp_tags)) {
          break;
        }
        continue;
      case 's':
      case 'S':
        /* Probably NOT an opening PHP <script> tag,
         * so don't end the HTML chunk yet
         * If it is, the PHP <script> tag rule
         * checks for any HTML scanned before it */
        YYCURSOR--;
        yymore();
      default:
        continue;
      }

      YYCURSOR--;
    }

    break;
  }

  inline_html:
  yyleng = YYCURSOR - SCNG(yy_text);

  Z_STRVAL_P(zendlval) = (char *)estrndup(yytext, yyleng);
  Z_STRLEN_P(zendlval) = yyleng;
  Z_TYPE_P(zendlval) = IS_STRING;
  HANDLE_NEWLINES(yytext, yyleng);
  return T_INLINE_HTML;
}
```

Because the program under consideration only consists of the literal input "Hello, World.", T_INLINE_HTML is the one and only token produced by scanning this input.

Parsing

The parser used in the Zend engine is produced by the open source parser generator program called Bison. Bison expects to be given an input file that is an LALR grammar in EBNF form for the language in question (PHP in our case). The LALR grammar input file must further be marked up with directives about code to be produced for translating syntactically correct statements in the source language into equivalent code in the target output language (opcodes in the case of PHP).

The text below shows the Bison input file used in the Zend engine. The start symbol for the LALR grammar for PHP is "unticked_statement." In PHP the rule with unticked_statement on the left roughly defines one line of code. The full source of the PHP language parser can be found at http://svn.php.net/viewvc/php/php-src/trunk/Zend/zend_language_parser.y?view=markup. We give a portion of it here. Unfortunately, there is no nice way to format this code for the printed page.

```
unticked_statement:
    '{' inner_statement_list '}'
  | T_IF '(' expr ')' { zend_do_if_cond(&$3, &$4 TSRMLS_CC);
} statement { zend_do_if_after_statement(&$4, 1 TSRMLS_CC);
} elseif_list else_single { zend_do_if_end(TSRMLS_C); }
  | T_IF '(' expr ')' ':' { zend_do_if_cond(&$3, &$4
TSRMLS_CC); } inner_statement_list {
zend_do_if_after_statement(&$4, 1 TSRMLS_CC); }
new_elseif_list new_else_single T_ENDIF ';' {
zend_do_if_end(TSRMLS_C); }
  | T_WHILE '(' { $1.u.opline_num =
get_next_op_number(CG(active_op_array));  } expr  ')' {
```

```
zend_do_while_cond(&$4, &$5 TSRMLS_CC); } while_statement {
zend_do_while_end(&$1, &$5 TSRMLS_CC); }
  | T_DO { $1.u.opline_num =
get_next_op_number(CG(active_op_array));
zend_do_do_while_begin(TSRMLS_C); } statement T_WHILE '(' {
$5.u.opline_num = get_next_op_number(CG(active_op_array)); }
expr ')' ';' { zend_do_do_while_end(&$1, &$5, &$7
TSRMLS_CC); }

  | T_FOR
    '('
      for_expr
    ';' { zend_do_free(&$3 TSRMLS_CC); $4.u.opline_num =
get_next_op_number(CG(active_op_array)); }
      for_expr
    ';' { zend_do_extended_info(TSRMLS_C);
zend_do_for_cond(&$6, &$7 TSRMLS_CC); }
      for_expr
    ')' { zend_do_free(&$9 TSRMLS_CC);
zend_do_for_before_statement(&$4, &$7 TSRMLS_CC); }
      for_statement { zend_do_for_end(&$7 TSRMLS_CC); }
  | T_SWITCH '(' expr ')' { zend_do_switch_cond(&$3
TSRMLS_CC); } switch_case_list { zend_do_switch_end(&$6
TSRMLS_CC); }
  | T_BREAK ';' { zend_do_brk_cont(ZEND_BRK, NULL
TSRMLS_CC); }
  | T_BREAK expr ';' { zend_do_brk_cont(ZEND_BRK, &$2
TSRMLS_CC); }
  | T_CONTINUE ';' { zend_do_brk_cont(ZEND_CONT, NULL
TSRMLS_CC); }
  | T_CONTINUE expr ';' { zend_do_brk_cont(ZEND_CONT, &$2
TSRMLS_CC); }
  | T_RETURN ';' { zend_do_return(NULL, 0 TSRMLS_CC); }
  | T_RETURN expr_without_variable ';' { zend_do_return(&$2,
0 TSRMLS_CC); }
  | T_RETURN variable ';' { zend_do_return(&$2, 1
TSRMLS_CC); }
  | T_GLOBAL global_var_list ';'
  | T_STATIC static_var_list ';'
  | T_ECHO echo_expr_list ';'
  | T_INLINE_HTML { zend_do_echo(&$1, 1 TSRMLS_CC); }
  | expr ';' { zend_do_free(&$1 TSRMLS_CC); }
  | T_UNSET '(' unset_variables ')' ';'
  | T_FOREACH '(' variable T_AS
```

```
    { zend_do_foreach_begin(&$1, &$2, &$3, &$4, 1
TSRMLS_CC); }
    foreach_variable foreach_optional_arg ')' {
zend_do_foreach_cont(&$1, &$2, &$4, &$6, &$7 TSRMLS_CC); }
    foreach_statement { zend_do_foreach_end(&$1, &$4
TSRMLS_CC); }
  | T_FOREACH '(' expr_without_variable T_AS
    { zend_do_foreach_begin(&$1, &$2, &$3, &$4, 0
TSRMLS_CC); }
    variable foreach_optional_arg ')' {
zend_check_writable_variable(&$6); zend_do_foreach_cont(&$1,
&$2, &$4, &$6, &$7 TSRMLS_CC); }
    foreach_statement { zend_do_foreach_end(&$1, &$4
TSRMLS_CC); }
  | T_DECLARE { $1.u.opline_num =
get_next_op_number(CG(active_op_array));
zend_do_declare_begin(TSRMLS_C); } '(' declare_list ')'
declare_statement { zend_do_declare_end(&$1 TSRMLS_CC); }
  | ';' /* empty statement */
  | T_TRY { zend_do_try(&$1 TSRMLS_CC); } '{'
inner_statement_list '}'
    T_CATCH '(' { zend_initialize_try_catch_element(&$1
TSRMLS_CC); }
    fully_qualified_class_name { zend_do_first_catch(&$7
TSRMLS_CC); }
    T_VARIABLE ')' { zend_do_begin_catch(&$1, &$9, &$11, &$7
TSRMLS_CC); }
    '{' inner_statement_list '}' { zend_do_end_catch(&$1
TSRMLS_CC); }
    additional_catches { zend_do_mark_last_catch(&$7, &$18
TSRMLS_CC); }
  | T_THROW expr ';' { zend_do_throw(&$2 TSRMLS_CC); }
  | T_GOTO T_STRING ';' { zend_do_goto(&$2 TSRMLS_CC); }
;
```

The T_INLINE_HTML token that was produced by the scanner is buried a little over halfway down in this section of code and has been bolded. Whenever the token T_INLINE_HTML is parsed, a semantic call to zend_do_echo() is made, which in turn accesses the text that the scanner accumulated (in yytext) when it matched the token T_INLINE_HTML, as described in the previous subsection.

The implementation of zend_do_echo(), which can be found in http://svn.php.net/viewvc/php/php-src/trunk/Zend/zend_compile.c?view=markup, constructs an opcode structure that will result in echoing the string accumulated by the scanner when that opcode is later interpreted. zend_do_echo() does this by setting the operation code for this opcode instance to ZEND_ECHO and setting operand one for this operation code to the literal text accumulated by the scanner, as shown below.

```
void zend_do_echo(const znode *arg TSRMLS_DC)
{
  zend_op *opline =
    get_next_op(CG(active_op_array)TSRMLS_CC);

  opline->opcode = ZEND_ECHO;
  opline->op1 = *arg;
  SET_UNUSED(opline->op2);
}
```

For the purpose of this simple example, this is essentially the end of the compile phase of PHP processing. The result of any compile phase is an array of opcodes generated by the Zend engine, each consisting of a single operation code and its operands. For this example, the opcode array has just one element, an opcode structure with the ZEND_ECHO opcode and a single operand that points to a string of memory containing what should be echoed (printed) when this opcode is interpreted. (In reality, the opcode array for this example would also contain a few additional helper opcode entries that are immaterial to this example, including a "return null;" statement that PHP includes at the end of every opcode array, as well as a catch point for thrown exceptions.)

Execution (Interpretation)

Once the PHP source code has been compiled into opcodes, the Zend Virtual Machine can be invoked to process the opcode array by interpreting the opcodes in successive opcode array elements. In terms of the execution diagram above, zend_compile_file() has finished and the virtual machine is ready to run zend_execute(). As it evaluates each opcode, zend_execute calls the handler function associated with current opcode type, which in turn carries out the interpretation of that opcode. In the case of an opcode that has operation code ZEND_ECHO, the execution handler can be found in http://svn.php.net/viewvc/php/php-src/trunk/Zend/zend_vm_def.h?view=markup and shown below:

```
ZEND_VM_HANDLER(40, ZEND_ECHO, CONST|TMP|VAR|CV, ANY)
{
  zend_op *opline = EX(opline);
  zend_free_op free_op1;
  zval z_copy;
  zval *z = GET_OP1_ZVAL_PTR(BP_VAR_R);

  if (OP1_TYPE != IS_CONST &&
      Z_TYPE_P(z) == IS_OBJECT &&
      Z_OBJ_HT_P(z)->get_method != NULL &&
      zend_std_cast_object_tostring(z,
      &z_copy, IS_STRING TSRMLS_CC) == SUCCESS)
  {
    zend_print_variable(&z_copy);
    zval_dtor(&z_copy);
  } else {
    zend_print_variable(z);
  }

  FREE_OP1();
  ZEND_VM_NEXT_OPCODE();
}
```

Really, the only interesting things that the handler does are to call zend_print_variable() and to advance the virtual machine to the next op code. If one follows execution through zend_print_variable(), one would see a number of indirect calls to functions through abstract function pointers which eventually results in a call to php_default_output_func() which does nothing more than call the operating system API function fwrite(). That API function prints the string residing in operand one of this opcode and which has been passed along through the many function calls in between. Once the PHP virtual machine reaches the final opcode, it begins its shutdown procedure.

Caching Opcodes

We now turn our attention to caching the opcode array and functions and classes produced by the PHP front-end compiler for later reuse.

Piggybacking on APC

The primary challenge in caching opcodes is copying the memory structures and all their dependencies, which were created by the compilation phase. Making such a copy requires extensive understanding of the internals of, and relationships among, those structures. This project benefited extensively from piggybacking on the work of the APC project. APC is designed so that the code that copies the PHP opcodes is defined almost entirely in a single file, apc_compile.c. Including that file in this project not only saved a tremendous amount of effort in the initial implementation, but also has allowed this project to be upgradable to new versions of PHP much more easily. Essentially, this project uses the parts of APC available in apc_compile.c as an API that can make a copy

of opcodes, etc. from PHP's execution environment out to other memory or, vice versa, from other memory back into PHP's execution environment.

Differentiating PHP's opcodes from typical byte codes

PHP's opcodes differ substantially in implementation from traditional byte codes found in languages whose programs are compiled once and executed many times in a VM. In that latter type of language, such as Java, the byte codes written to disk are conceptually similar to native machine code. They are compact descriptions of what to do. The operands of operations are typically CPU registers. This type of byte can contains no memory addresses that are specific to program execution in which they were generated. Consequently, they can be written to disk in the form they are generated.

PHP's opcodes are implemented quite differently. Rather than a compact stream of instructions, PHP's opcodes are bloated memory structures. In PHP 5.2, each opcode structure required 96 bytes of memory. The greatest portion of that bulk comes from the inclusion of structures for two operands and a result value. Space is allocated for those three structures regardless of the number of operands accepted by the operation type or whether the operation has a return value. The opcode itself and the structures for the operands and return types can include pointers to other regions of memory. Below we'll describe special steps required to deal with those pointers.

Serialization

Because APC is not designed to write cached opcodes to disk, it has no need to carefully allocate memory for the copied opcodes. It simply allocates memory into a shared region in whatever manner the OS chooses. Because this project must eventually

write the opcodes to disk, it must copy the opcodes and their dependencies serially into a contiguous region of memory. That block of memory can then later be written to disk. Conveniently, APC was written in such a way that the implementation of memory allocation is determined by a function pointer. This project was able to provide APC with a serial memory allocator by simply passing it a function pointer to such an implementation.

The Necessity of Rebasing

Another issue to note is that simply using APC to make a serial copy of the PHP compilation result to disk is insufficient. If such a copy were naively written to disk and later loaded back into memory, all of the pointers would very likely be incorrect. That is, whenever the cached opcodes were read back from disk into memory, they would most likely be loaded at a different starting location in memory than the original. Thus, the original pointer values would be invalid.

To handle this problem we took our inspiration from dynamically linked libraries. Program libraries that are loaded dynamically (i.e., at run time) face a similar problem. These libraries, which are called "shared objects" in Unix systems and "dynamic-link libraries" in Windows, can be linked to by many different programs during execution to allow code reuse. Runtime dynamic linking is often implemented in one of two ways. One way is to require that the library always be loaded at a specific location in memory. This approach allows the library to be loaded quickly, since internal addresses do not need to be modified; however, this approach presents a somewhat inflexible interface to the programs using it. For instance, a program would be unable to use two libraries, each

of which required that they be loaded in overlapping address ranges. A second approach is to allow the library to be located anywhere. In this case, the library's internal addresses need to be corrected based on where in memory the library is relocated. There are various ways to manage this relocation. For example, addresses internal to the library can be computed as offsets from a base register; then only the base register needs to be modified when the library is relocated. A second way is to ensure that all fixed addresses in the library are created relative to some fixed address; in this case when the library is relocated, each such address must be modified with respect to the fixed address and the new real location in memory where the library is relocated.

This project fixes up all of the addresses at the time that the opcode cache file is loaded, rather than recomputing each address as an offset from a base address on access, so that the result may be handed off to the borrowed portions of APC. Given inputs in the expected format, APC is able to copy the cached compilation result into PHP's execution space.

Preparing for Rebasing

To copy the compilation result, APC first allocates memory by calling a function pointer that was passed to it. The opcode caching extension provides APC with a function that serially allocates memory into a large, dynamically allocated region. APC then calls memcpy() to copy source structures to the newly allocated memory. Afterward, it handles the members of the copied structures.

The single change thus far made to the APC source is to replace its calls to memcpy() in these contexts with a call to a function pointer. By default the function

pointer references memcpy(). When precompiling, the extension assigns the pointer to a function that copies the requested memory and then examines the destination memory for any words which contain values which could be mistaken for pointers into the current or any previous serialization regions. These values cannot correctly be pointers to these memory regions because such memory regions were reserved by malloc and from which only the allocator function can legitimately allocate other memory. When PHP's compiler ran, these regions were not allocated. If a value points to one of these regions, it's incorrect.

Upon finding a value that appears to point into the serialization region, the special memcpy() function records the offset of the value from the beginning of the region. If the serialization region needs to grow, a larger memory region is allocated, the original is copied to the new, and the new is examined for any words which appear to point into it. If any are found, their offsets are also noted. The old serialization region is not freed immediately to prevent its address range from becoming a legitimate target for new allocations.

When APC finishes copying the compilation result, the serialization region is examined for words which point to the current or any previous serialization regions. When one is found, the extension ensures that the word's offset does not match any that were previously noted as not being pointers. If the word is in fact a pointer, its offset is recorded and its value is translated to an offset relative to the beginning of the region into which it points.

The serialization region and the list of offsets to pointers are then written to the op code cache file. The op code cache file will be named the same as the original with ".op_codes" suffixed and as a sibling of the original.

Using Cached Opcodes

When opcode caching is enabled, the pointer to the PHP compile file function is replaced with the extension's own function. That function will check for the existence of an opcode cache file, that is, a file on disk with the same name as the requested file suffixed by ".op_codes." If the file is not found, is too small, or is obviously corrupted, the default PHP compile file function will be called. It will also be called if any other recoverable error occurs while restoring the opcodes from cache. To illustrate, here is the same diagrammatic high level view of PHP execution updated for the addition of opcode caching.

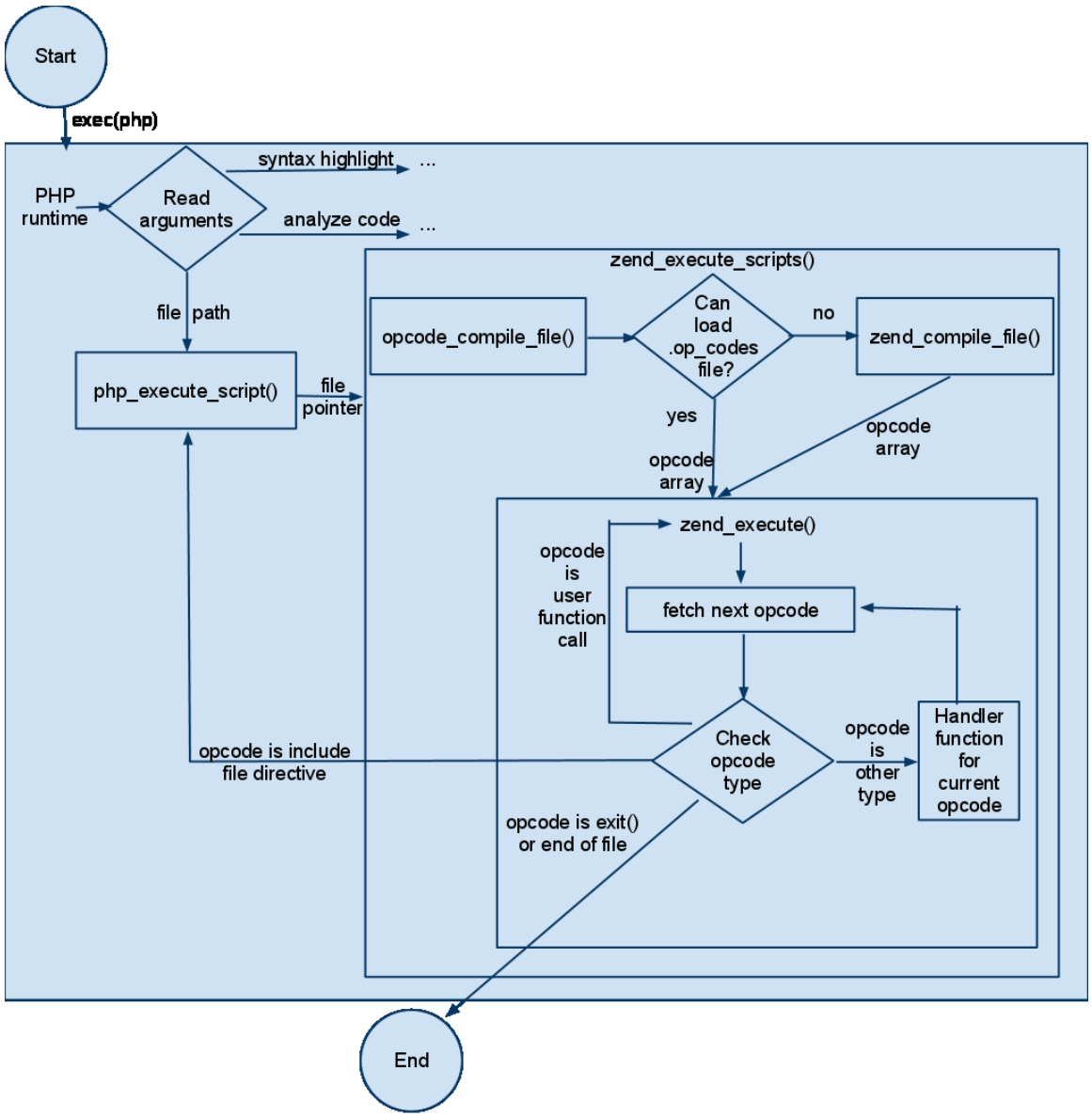

Figure 2 High Level View of PHP Execution with Opcode Caching

If the opcode cache file is found and looks well-formed, it will be read into a single block of memory. Before that block of memory can be handed to APC to restore the cached entities, it needs to be preprocessed as detailed below.

Rebasing Pointers

The opcode cache file contains a list of offsets to locations that represent pointers within the cache. These offsets can be thought of as pointers which are pretending that the opcode cache file was loaded at memory address zero. Of course, it wasn't. The starting memory address (base address) that the file was copied to is added to the value (pointer) at each of these offsets to create a correct pointer.

Correcting Opcode Handler Function Pointers

Each PHP opcode contains a pointer to the PHP VM function that executes the opcode. This pointer may be incorrect when the opcode is restored if different PHP binaries created and loaded the op code cache files. This situation may be encountered when using a stripped versus a debug symbol carrying PHP binary. This scenario is commonly encountered because the version of PHP used by developers differs from the optimized version used in production.

The PHP API has a function that sets an opcode's handler. This function is called for each restored opcode. It would be possible to use a faster rebasing approach to correct these pointers; however the necessary data is not in the public PHP API, so doing so would require modifying the PHP code to expose the necessary function.

Preserving Array Entries' Hash Code

After this project had been in production for over 15 months and had served tens of millions of requests, an error was found when running an unreleased version of software. The bug manifested itself in that some member methods and variables of an object were not available when the PHP runtime tried to call or read them. Analysis

revealed that the members were being compiled correctly, were appropriately present in the opcode cache file, and were duly loaded into the PHP execution space when the opcode cache file was loaded. After the object was initially instantiated, the member was sometimes available, but later would be absent. Uncovering the problem required a deeper look into PHP.

PHP has an "array" data type that serves myriad purposes. Intuitively, the array type allows values to be indexed by an integer value. Unintuitively, it also allows values to be indexed by any other type, which is, in fact, coerced into a string. In this way, the array also serves as a map data type (also known as a dictionary, or hash table). Oddly, the array also retains the order in which entries are added to it independent of the value of the keys. Thus, iterating through the array produces the values in entry order, rather than in the order suggested by the keys' numeric values, as illustrated in the following example.

Running the following PHP code

```
<?php
$array = array();
$array[2] = 'two';
$array[1] = 'one';
$array[0] = 'zero';
$array[3] = 'three';
foreach ($array as $element) {
  echo "$element, ";
}
```

will result in output of

```
two, one, zero, three,
```

A more traditional implementation of an array data type would result in the elements being printed in the natural, numerical order of their indices, that is

```
zero, one, two, three,
```

To provide these capabilities, PHP's array type is implemented as a hash table plus a linked list to record the order in which entries were added. Each hash entry includes the key (which can be either an integer or string), the hash value of the key, and a value (which can be an atomic value or a pointer to a more complex type.) This array type is not only exposed to PHP script, but also used promiscuously throughout the PHP internals. Importantly, the array is used to store the member variables and functions, in separate collections, declared for a PHP class.

Given this knowledge, our problem was uncovered. It arose because the rebasing code could mistake the value of a hash array entry's hash code for a pointer. The mistaken identification would only occur if the hash code was a value that was within the range of memory addresses into which the serialized copy of the PHP internal structures was being copied (although the rebasing logic always examined the array of memory that the hash code was stored in for possible pointers, it would take no action if the value wasn't in the correct range). Mistaking a hash code for a pointer, the rebasing logic would convert the hash code value to an offset that was then written into the opcode cache file. When that value was reloaded, it would be rebased relative to the new location at which the cache file was loaded. That new, erroneous value would then be carried around in the hash entry.

This error would only manifest itself in an array which had been generated and only if a large enough number of items were added to the array to force the array to grow. When looking up an entry in the array, PHP does not compare against the hash code. Instead it uses the modulus of the requested index's hash code to jump to the appropriate bucket. Inside the bucket, PHP does exact matches on keys. However, when an array has become full enough that it needs to grow, PHP reinserts all of the entries based on the modulo operand of the revised capacity. Only at that point does the corrupted hash code effect program flow because it is then that PHP stores the entry in the incorrect bucket.

To correct the problem, an additional modification was made to the APC source code. APC was changed to call a function in this project's serial memory allocator after copying each hash entry. That function would add the hash entry's hash code to a list of addresses that should not later be examined for possibly containing pointers.

Correcting Path Constants

The PHP language includes __FILE__ and __DIR__ constants, which, respectively, evaluate to the path of the executing script and the directory containing the executing script. When encountering these constants during compilation, PHP converts them to strings of the correct value based on the location of the script file being compiled. This, however, presents a problem for opcode cache files because these files are deployed to new locations after they have been created. Consequently, the value that would be hardcoded into the cache file is incorrect. As a result, during the creation of opcode cache files, usages of __FILE__ and __DIR__ are recorded in the cache file.

When the cache file is loaded into memory, those locations are replaced with correct paths.

Handoff to APC

Once all of the aforementioned complications have been resolved, the result is a batch of PHP class and opcode structures that are in the format APC expects to find in a shared memory cache. This project then hands that batch to APC. It asks APC to copy the functions, classes, etc. back into the PHP execution environment. Once APC has finished installing those objects, control is returned to the PHP virtual machine. It then begins to execute the freshly installed opcodes.

Other Complications

Any non-trivial software project uncovers additional requirements while designing, implementing or maintaining. Of course, this project was no exception. Some of those are related below.

Creating Consistent Opcode Cache Files

During the first iteration of this project, the author was asked if the produced op code cache files were consistent for the same input. His initial assumption that they were consistent was found to be incorrect. Although this inconsistency didn't have an impact on the runtime efficacy, it did cause uneasiness about reliance on a nondeterministic process for production software. It also posed a problem for those who maintain the production environment, because when building service packs, they preferred to rebuild the software and include files which differed from the last build. However, because the

op code cache files differed each time they were produced, building the service pack would have required more complicated logic.

The majority of the inconsistencies in the op code cache files were due to PHP not consistently initializing the underlying memory when translating the source program into in-memory op code cache structures. Those structures often have the potential to accommodate a superset of functionality and thus have members that aren't initialized for some usages. Those random, uninitialized values in the underlying memory for unused locations are eventually copied into the op code cache file.

The solution to this problem was to coerce PHP to consistently initialize memory when allocating it. This turned out to be relatively straightforward, as the Zend engine already included a number of memory managers, including one that did exactly what was needed. Invoking this memory manager was as simple as setting an environment variable to the correct value when PHP's command line parser found the switch that told it to create an op code cache file.

This solution worked nearly completely with the exception of one four-byte word which varied with each run. Analysis led to the conclusion that the varying location was always in the first operand in the penultimate op code in a file's opcode array. Further analysis revealed that this operation was the return statement with which all PHP scripts implicitly end. For an unknown reason, APC did not attempt to copy this operand when duplicating PHP's copy of the op codes. The easiest solution, which quite conveniently worked, was to always set that value of that word to zero before writing the op code cache file. This solution has no adverse affect regardless of the type of data being

returned because that penultimate opcode is implicit. Even if the script ends with an explicit return statement, the PHP parser injects an additional opcode equivalent to the statement "return null."

Updating the Project for PHP 5.3

The project was originally written for use with PHP 5.2. Later it needed to be upgraded to work with PHP version 5.3. A naive attempt to run the extension with the new version resulted in process terminating segfaults. The problem was that PHP 5.3 introduced a new garbage collector. The version of APC the project was originally integrated with, 3.0.18, was not compatible with the new runtime. Integrating with a more recent version of APC which supported PHP 5.3 solved the segfaults.

In general upgrading to the newer version of APC was a satisfying low-friction endeavor. The APC programming interface was largely unchanged in version 3.1 allowing this project's code to continue to interoperate. The greatest hurdle was that APC 3.1 introduced the concept of configurable memory allocators, which were fairly similar to the serial memory allocator concept this project introduced to APC originally. The existing memory allocator needed to be adapted to expose the interface used by APC's new allocator type.

In an intriguing bit of parallel work, APC introduced the apc_bin_* functions, which will dump and load data to and from a file. This would seem to be a nice fit for the opcode caching project. Unfortunately, it doesn't seem to work. The functions are labeled as experimental and don't seem to work correctly, yet. Secondly, it's not a good fit because it requires PHP code to load other PHP files into memory. It does not

integrate into the normal PHP mechanism for including PHP scripts. This implies requiring PHP scripts to be written to use a PHP function to perform the equivalent of a require_once(). This change would dramatically reduce the transparency with which the opcode caching extension can currently be turned on and off, which is fantastically useful for determining if an error is due to the caching extension. Finally, it's not a good fit because the functions are designed to load file contents into shared memory, rather than directly into the current process' PHP executor.

Results

This project has seen constant use in a production, commercial environment since November 2008. In that time it has improved the performance of many millions of server requests. It has also allowed a different, larger software initiative to meet the performance criteria for its release. No defects in the project have been reported.

To provide an example of how this project improves the performance of PHP execution, most of the PHP files that form the framework for a commercial software package were combined together into a single file. This file contained roughly 655 KB of code in 20,000 lines. This framework code was composed entirely of function and class declarations, so including it did not cause PHP to execute much code; rather it primarily exercised the scanner and parser. This file was included by another PHP file which read times from the system clock before and after including the file.

Over 1000 trials, including the plain script version of the file took an average of 42.3 ms, while the opcode cached equivalent could be included in 9.5 ms. This saved 32.7 ms, a reduction of 77.4%. Nearly all of the time saved was CPU time, rather than

time waiting for IO. Saving CPU time was the key goal of the project because reduced CPU utilization meant that the larger initiative it supported would scale better on existing hardware and could be deployed without requiring the purchase of additional hardware. This time saving also had the benefit of delivering content to users sooner.

Future Directions

Fundamentally this project exists to serve a larger commercial software package. At the present time, all the stakeholders seem to be satisfied in this project's performance and stability; thus there is no desire to expend further engineering resources to improve it. However, there are interesting developments which will be worth considering advancing this project.

Because this project supports a production software package, it will need to be updated when a new version of PHP is desired. It is hoped that future version upgrades will require minimal effort due to this project's reliance on APC. Historically APC has been ready for new major versions of PHP well before their general release.

The APC project's recent commits have shown signs that they may be preparing to implement functionality somewhat to what this project offers. If indeed that were delivered, this project should be updated to take advantage of that to whatever degree possible. The author expects that reducing the code in this project while further utilizing APC will reduce future maintenance costs.

Facebook has released an interesting alternative to traditional PHP acceleration techniques called HipHop for PHP. The developers at Facebook apparently determined that the PHP virtual machine was irredeemably slow; consequently, they took the novel

approach of translating PHP to C++ and compiling it to a native executable (https://github.com/facebook/hiphop-php/wiki). Currently HipHop has numerous shortcomings which prevent its use in this project's environment. For example, HipHop can only be compiled as a 64-bit executable, but 32-bit would be required. HipHop doesn't allow a mix of translated and natural PHP to be executed in the same process. Also, HipHop has a different extension API than PHP, which would require an enormous rewrite to accommodate. However, were the HipHop project to address these and other deficiencies, it should be considered as a complement to or replacement for this project.

Conclusion

This project was conceived to provide a necessary performance and scalability boost to a larger commercial software package. That package has a unique set of requirements for its operating environment. For improved security and reliability, this environment does not allow shared memory or shared file systems. Furthermore, it does not allow files to be written in the executable area of the file system except during a time window of system maintenance. Consequently, any files containing opcodes need to able to be written in one file system location so that they can be distributed and later read and executed from other paths. These unique requirements precluded using more traditional means for caching opcode files for boosting the performance of PHP.

This project borrows techniques from existing systems, particularly the PHP APC project, to cache the opcodes generated by the PHP parser. These techniques are combined with the injection of serialized allocation and with an adaptation of the dynamic linked library concept of relocation. This amalgamation allows opcodes to be

written to disk in a format that can be loaded on a different computer than the one on which it was created.

This project has demonstrated that it is possible to treat PHP as a "compile once, execute many" language. With the PHP source precompiled to opcodes, subsequent executions can skip the computationally expensive steps of scanning and parsing.

This project has enabled the success of a larger commercial software package. Without this project, that package would not have met the performance metrics required for it to ship. Since that package's release, its execution has been accelerated by this project in tens of millions of production web requests.

This project was implemented in a sufficiently flexible and robust manner that it was able to be modified to work with the substantial changes incorporated into a new version of the PHP virtual machine. Neither large code changes nor a herculean coding effort was required for the upgrade.