P2PC - The PHP-to-PHP Compiler

Jeff Disher (Open Autonomy Inc.)

March 5, 2014

ABSTRACT

P2PC is a compiler for processing PHP source code for easier deployment and faster loading.

Although this task has been attempted by other PHP compilers, they typically impose restrictions on the target deployment environment or restrict access to PHP extensions. P2PC, while only supporting a subset of PHP language features, does not add additional restrictions to the deployment environment or what extension libraries can be accessed by the user code.

The use of P2PC, at OpenAutonomy, has reduced deployed package and code size, and increased performance of our offering without compromising the primary goal of the product to easily be deployed on virtually any PHP-enabled web server.

This paper outlines the goals of the compiler, the approaches taken in implementation, and results observed in terms of APC opcode cache consumption, physical script size, and script execution behaviour.

1. INTRODUCTION

P2PC is a compiler whose input and output languages are both PHP and it is also implemented in PHP. This is a divergence from common PHP compilers which typically produce machine code, C++, or some kind of bytecode, from the PHP input source.

The design goals of the compiler permitted this decision as they were more directly targeting high-level performance concerns related to how scripts are found, at run time, as well as simplification of deployment packaging. The goal was not to replace or defeat the interpreter.

This means that the compiled result of a given script, assuming the input can be compiled by P2PC (as only a subset of the PHP language is supported), can run in the same environment as the input script. No additional constraints or limitations are added.

2. COMPILER GOALS

The OpenAutonomy source code is structured much like a set of programs (the PHP entry-point scripts) and a backing collection of libraries (the common code used by all the entry-points).

This means that a typical entry-point will have several supporting dependencies (which, in turn, may have their own dependencies). These dependencies are included via `require_once` calls. Only the entry-point script contains code to be executed on load, while the included code typically only includes the definitions of code (although there are a few exceptions to this).

Additionally, only the entry-point scripts contain any content other than the contents of the PHP start and end tags. This can be considered a rule with no exceptions.

The design goals were directed with this structure in mind.

2.1 ELIMINATE REQUIRE_ONCE COST

Early profiling of the OpenAutonomy distribution demonstrated that a large portion of the script execution time was spent in executing the `require_once` calls at the top of the script.

Upon closer examination, this cost appeared to be proportional to the product of the number of `require_once` calls and the size of the include path. PHP's APC opcode cache only makes this cost more obvious as other script execution costs are reduced while this stays constant.

Given that the list of files included via this mechanism doesn't change from one invocation to the next, this was identified as an obvious area of interest.

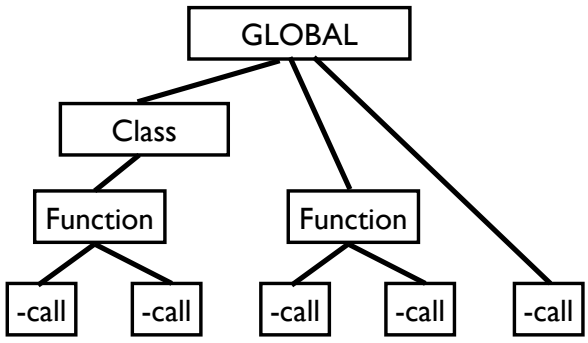
2.2 SIMPLIFY DEPLOYMENT OF SUPPORTING LIBRARY CODE

As the number of files and directories

storing the common support code grew, manipulations to the include path became a requirement. This added extraneous complexity and rigidity to the deployment.

3 DESIGN OVERVIEW

P2PC is designed much like a traditional compiler except that it doesn't create a full *abstract syntax tree* (AST) for analysis but only creates a *coarse symbol tree*. This exception is for performance of the compiler and expedience of implementation.



Coarse symbol table structure

If future extensions to the logical model are required, a more complete AST implementation will be required.

The high-level design follows a pipeline model, with each component *pulling* data from the previous component and many components being optional. Generally speaking, the components are in this order: pre-processor, lexer, parser, analyser, and output.

3.1 PRE-PROCESSOR

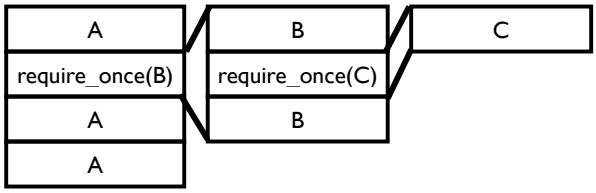
The pre-processor makes assumptions regarding the shape of a PHP source file. First of all, it assumes that only the entry-point script can have content outside of the PHP start and end tags and that there is only one set of these tags. Any content before or after this set of tags is passed through to output, untouched.

When parsing the tag contents, the pre-processor breaks the it into lines and makes a single decision regarding each line: inline a `require_once` or pass the line through to the next phase of the compiler.

If the line contains a `require_once` statement, the include path provided to the compiler is consulted to find the referenced file. If it isn't found, the line is passed through to output. If it is found, however, then the pre-processor pushes the remaining array of file lines onto a stack of pending work and recursively invokes itself to process that file (discarding any lines before or after the PHP start and end tags in said file). When it finishes processing all lines in a file, it pops the next array of file lines from the stack to continue where it left off.

It also maintains a list of files it has already included so that the same file referenced via multiple paths is only included once. Additionally, a list of ignored file names can be provided. Any file name in the ignore list or already included is treated as though it were empty. That is, the `require_once` line is removed but nothing is produced.

This creates a stack of file lines currently under evaluation which allows the contents to be "pulled" by the next phase without changing the order of lines in the script.



Inlining `require_once`

3.2 LEXER

The lexer operates as a tokenizing stream operator: the next phase in the compiler requests tokens from it and it maintains an internal buffer of one line which it, in turn, replenishes from the pre-processor.

Multi-line comments are handled as a special case, wherein the lexer will eagerly request new lines until it finds the end of the comment. The entire comment is then returned as a single token.

Multiple comment token types are provided since the next phase typically ignores single-line and multi-line comments but some

for user-defined purposes. The additional comment tokens are called *special* and *export*.

Special comments and normal single-line comments are tokenized the same way: the token consumes the rest of the line. The only difference is the token type, thus allowing them to be filtered out of the token stream, independently.

Export comments are treated differently as they contain information which is required by the parser so only the start of the comment "*//EXPORT*" is returned as the export token.

Language keywords are tokenized by exact string match while variables (anything starting with "$"), white-space, string constants (both kinds), numerical constants, and other identifiers (alpha-numerical sequences not starting with "$" or a number) are tokenized by reading the line buffer until a character which isn't part of the set available for that token type is found.

As the lexer adds considerable time to the execution of the compiler, this strategy might change in the future. Specifically, indexing into a string to read a character at a given offset appears to be exceptionally expensive, in PHP.

```
<?php
// Comments...
class MyClass
{
 public function myFunction($arg)
 {
 }
}
?>
```

PHP Source

```
PHP_START COMMENT CLASS
IDENTIFIER OPEN_BRACE PUBLIC
FUNCTION IDENTIFIER OPEN_PAREN
VARIABLE CLOSE_PAREN OPEN_BRACE
CLOSE_BRACE CLOSE_BRACE PHP_END
```

Lexer Token Stream

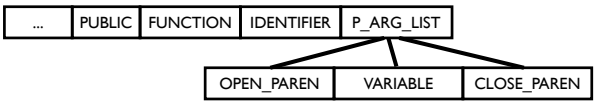
3.3 PARSER

The parser is implemented as a look-up table state machine emulator. The actual grammar is specified as a Bison grammar file. Bison is run on the grammar to produce an XML description of the state machine to parse the grammar. This XML is read into P2PC and emulated by the parser.

Note that this XML is treated as a resource to be distributed with the compiler so Bison is only run when developing the compiler and making changes to the grammar. The end-user does not need to run Bison or have it available on their machine.

This means that there is relatively little complexity to this component as all the heavy-lifting is done by Bison.

The parser itself operates by requesting tokens from the lexer and passing them into this state machine. Reduction rules create *parse tree objects* from the *elements* on the parser's stack. Both lexer token objects and parse tree objects implement a common element interface which allows them to coexist in the resultant parse tree.



Parser stack after argument list reduction

Upon completion of the parser, it returns a single completed parse tree to its caller which represents the entire structure of the code reachable from the lines provided via the pre-processor.

3.4 ANALYSER

As mentioned earlier, a complete AST is avoided in favour of a coarse collection of high-level symbols. Concretely, this means that the information found is limited to class, abstract class, and interface declarations along with the functions defined in either a class, abstract class, or global scope. Additionally, function calls and constructor calls are identified along with their calling context (that is, the function containing the call, which could be the global scope).

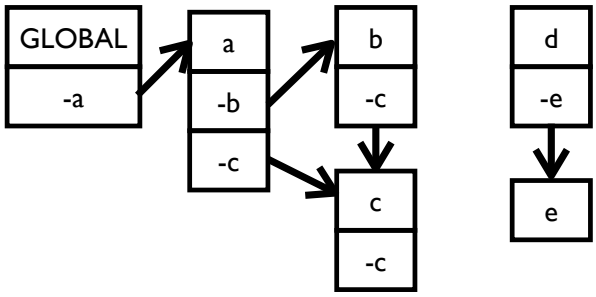
This information is sufficient for a basic implementation of the dead code elimination optimization as well as information which is required

by future ideas regarding type checking and function inlining.

3.5 DEAD CODE ELIMINATOR

The dead code eliminator is effectively a mark-and-sweep garbage collector which treats function definitions as the *objects*, function calls as *object references*, the global execution context (that is, code which is not inside a function) as a *root*, and currently treats interface functions or abstract function declarations as *roots*.

Since the eliminator is single-threaded and call-stacks aren't expected to become exceptionally deep, the marking operation is implemented as a recursive traversal. All functions start out as *dead* and marking begins with the root set, marking any referenced dead functions as *alive* and traversing their function references. Note that functions which were already alive are not traversed.



Function call marking (d and e are dead)

Global and static functions can be unambiguously identified but virtual functions are not as concrete so all virtual functions sharing the same name are treated as a single unit: any call to any of them marks all of them as alive. Tighter constraints would be possible with further type analysis in the analyser. Because of this, constructors are treated as static functions (their receiver implementation is unambiguous and can be concretely bound).

As many PHP functions are called via runtime evaluation and cannot be bound at compile time, a list of *exports* is required in order for these functions to be kept alive. The `//EXPORT` comment token is used for this purpose. This comment token names the function (either a global or static function) which should be kept alive when its containing function is marked alive. This means that the export list is not global but is bound on a per-function basis. This has the advantage of meaning exported functions called only by dead functions can be safely marked as dead, as well.

4 MODES OF OPERATION

The depth of effort expended by the compiler can be controlled via command-line switches. The main reason for this is performance (the pre-processor and parser are relatively fast while the lexer and the analyser are very slow) but this is also done to make the compiler more accessible to other projects.

As stated earlier, P2PC only implements a subset of PHP and does so in a very strict way. This may cause incompatibilities with other projects which they do not wish to resolve. The earlier parts of the P2PC pipeline should still be of some use to them, in these cases.

4.1 PRE-PROCESS REQUIRE_ONCE

Running only the pre-processor is very fast and has the effect of only inlining require_once calls and leaving the code otherwise untouched. This is a useful option as the full compiler takes much longer to process the same input and provides the largest performance benefit to the output script.

This is enabled with the `--preprocess` option.

4.2 STRIP COMMENTS AND WHITESPACE

This mode is substantially slower as it enables the lexer. Comments (except special or export comments) and white-space are normally stripped from the token stream, before being consumed by the parser. When running in this mode, the operation is the same except that the parser is not enabled and this "filtered" token stream is written directly to the output.

While this has no effect on runtime performance (assuming APC is enabled), when compared to just running the pre-processor, it does make the resultant file substantially smaller, which allows for smaller packages in deployment and

can give a small performance benefit to APC-disabled environments.

This is enabled with the `--strip` option.

4.3 DEAD CODE ELIMINATION

This option requires all components of the compiler to run and is therefore the slowest mode of operation.

The output is similar to what is seen with `--strip` except that functions which are not called are removed from the output, entirely.

While this has little effect on runtime performance (beyond the benefits of `--strip`), it does make the resultant file substantially smaller and reduces the total APC capacity it consumes.

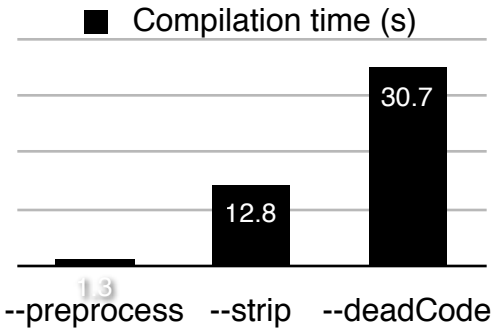
This option requires `--parser` to specify the grammar XML file and is then enabled with the `--deadCode` option.

5 PERFORMANCE RESULTS

Performance runs and size measurements were conducted using revision 267 of OpenAutonomy (which includes the compiler) on a Raspberry Pi Model B (details in Appendix A).

5.1 TIME TO COMPILE COMPILER

The typical time to compile P2PC using a non-compiled version:



Compilation time grows quickly (more than 23x from the 1.3 second pre-process run) as new compiler phases are added. The scale of these numbers mean that it is currently not feasible to run higher optimization levels as part of active development although it may be acceptable for distribution or deployment builds which happen less frequently.

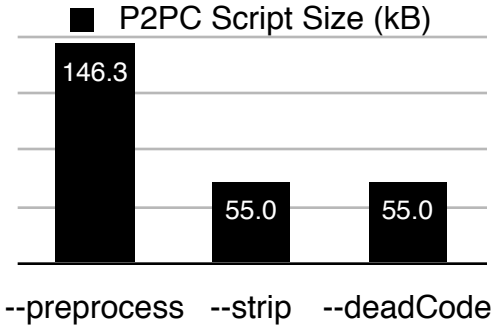
5.2 TIME TO COMPILE IDENTITY

The typical time to compile the OpenAutonomy identity entry-point (`identity.oa/index.php`):

This chart shows the same kind of growth as seen when compiling P2PC but the size of this larger script shows how the later optimization phases do not scale as well as the pre-processor (39x from the 5.3 pre-process run).

5.3 COMPILED COMPILER SIZE

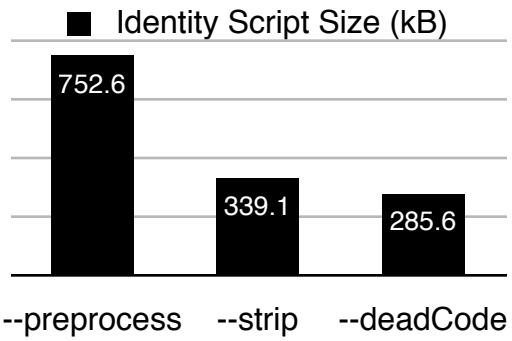
The total size of the output file generated when compiling P2PC:



There are many small classes in P2PC so license headers and other comments quickly dominate the total deployed script size. This is why there is such a sharp reduction in size simply by running in strip mode. There is no change (the files are identical) when enabling dead code elimination since there is no dead code in P2PC (as it only has a single entry-point).

5.4 COMPILED IDENTITY SIZE

The total size of the output file generated when compiling the OpenAutonomy identity:

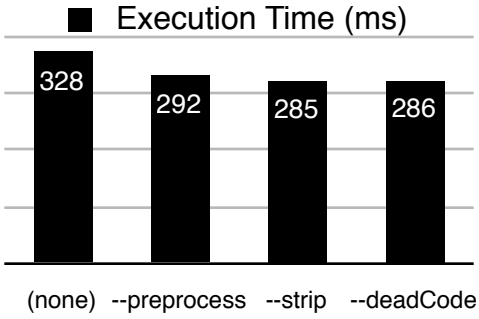


Similarly to P2PC, OpenAutonomy source files contain license headers but most classes are still reasonably large so the proportional win by enabling strip is less (although still substantial).

Also, because OpenAutonomy has many entry-points and identity is only one of them, there is a substantial amount of code not reachable from this one entry-point. This is why dead code elimination can still have such a substantial effect (15.7% reduction in size).

5.5 P2PC START-UP TIME

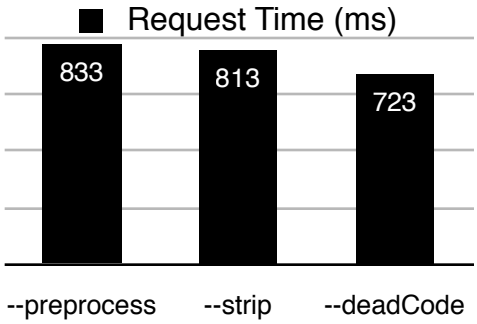
The average amount of time taken for P2PC to start up, print its usage string, and terminate (average of 1000 runs):



The non-compiled sources are provided here to show some of the benefit of running with the pre-processor (as OpenAutonomy no longer supports being run without it). The added cost seen in the non-compiled run is purely due to the execution of `require_once` statements (11% of the time) and this overhead grows as more files are included or more components are added to the include path.

5.6 TIME TO FETCH IDENTITY PAGE (APC DISABLED)

The average amount of time taken to fetch the identity page (`wget`, no cookies) while APC is disabled on the server (average of 400 runs):



Without APC, the script needs to be compiled for every request. Reducing the size of the script reduces this cost. The cost per request is still very high, demonstrating the need for APC in order to see more reasonable performance.

The reason why this page load, in particular, is being used is that it has a very long path length, thus demonstrating the effects of changes to the PHP. In practice, this page is only loaded once and RPCs are used to dynamically update the page on the client.

5.7 TIME TO FETCH IDENTITY PAGE (APC ENABLED)

The average amount of time taken to fetch the identity page (`wget`, no cookies) while APC is enabled on the server (average of 4000 runs):

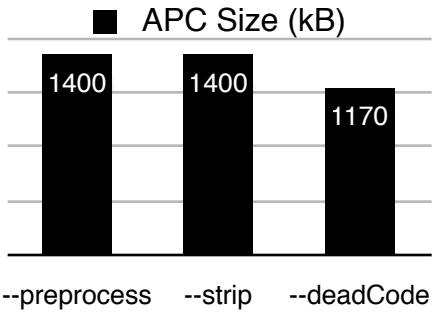
The time is the same across all variants since APC will only actually read the code once

and other calls will hit the cache. The benefits of dead code elimination aren't seen here as the same code is being executed.

This also shows the substantial performance win attributed to APC (7x) and why it should be enabled in all environments where it remains stable.

5.8 APC CACHE SIZE FOR IDENTITY

The size, in bytes, of the APC opcode cache consumed by the OpenAutonomy identity entry-point:

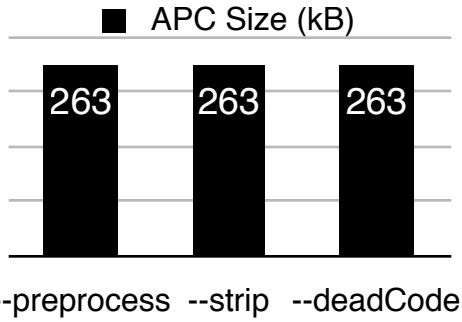


Pre-process and strip are the same, since they only change comments and white-space which are discarded by the interpreter, anyway.

Running with dead code elimination, however, reduces the consumed APC size by 16.4% since code has actually been removed.

5.8 APC CACHE SIZE FOR P2PC

A contrived test, measured by trying to access P2PC through the web browser, just to show the code size as seen by the PHP interpreter.



The cache size is the same for all variants since there is no dead code in P2PC as it is a single entry-point into its code.

6 FUTURE POSSIBILITIES

While there are no immediate plans to continue with substantial work on this project, below is a discussion of some possible directions which have been partially sketched-out.

6.1 OPTIMIZATIONS TO P2PC IMPLEMENTATION

Much of the overhead of the later phases of the compiler could probably be avoided via further profiling and tuning. Specifically, the lexer may benefit from being re-written to use regular expressions or `strtok` as these would both avoid the enormous cost of accessing a string by index, directly in PHP.

Additionally, the analyser could benefit from changing its current set of small tree walkers (some of which walk the tree, redundantly, since they only look for one piece of information) into a single, stateful walker. The symbol implementations also may be faster to access if they held the direct strings underlying the identifiers they represent, instead of their token objects, as their are many calls which need the actual string.

6.2 CODE VERIFICATION

PHP's lack of types make it difficult to develop large applications as there is no static limitation on what kinds of data can be passed to or returned from a given function. Some static analysis could be done to ensure that parameters and return types are viewed consistently between caller and receiver.

PHP's local variables do not require explicit declaration and it is possible that some paths will never declare a variable which is referenced after a join point. This promotes basic typos into confusing runtime bugs and means that uncommon paths might fail unexpectedly due to a simple typo or missing variable declaration. Adding variable scope detection to the analyser would catch these problems at compile time.

6.3 OUTPUT OPTIMIZATION

Many functions are called only once or are called with very specific parameters, effec-

tively creating a logical idiom. Deeper variable knowledge could allow for function inlining to eliminate additional function calls or function versioning to remove parameters or statically-decidable conditions (potentially leading to further inlining opportunities).

Assigning to new local variables grows the hash table underlying the stack frame which both adds runtime cost to the execution of the function but also increases the number of objects the function keeps alive. Function performance could improve and memory footprint could be reduced by adding variable life-span tracking and treating the hash table as a register file, allowing entries to be aggressively re-used or cleared.

7 CONCLUSIONS

P2PC, despite being a simple compiler, has provided the means to resolve runtime performance problems related to a large multi-file code base, reduce total package size, and reduce extraneous APC cost.

Additionally, it provides the basis for future code verification and performance-enhancing features.

All of this is possible without imposing deployment environment restrictions, or limitations on what kind of PHP interpreter or native extension libraries can be used.

APPENDIX A: BENCHMARKING ENVIRONMENT

Raspberry Pi (Model B) details:

- **Distribution:** Raspbian GNU/Linux 7
- **Kernel:** Linux raspberrypi 3.6.11+ #557 PREEMPT Wed Oct 2 18:49:09 BST 2013 armv6l GNU/Linux
- **Total available memory:** 497504 kB
- **Processor:** ARMv6-compatible processor rev 7 (v6l) (BogoMIPS : 697.95)
- **Storage:** Kingston SDHC 8GB class 10 (formatted: btrfs - noatime,compress=lzo)
- **Apache:** Apache/2.2.22 (Debian) PHP/5.4.4-14+deb7u4
- **PHP:** PHP 5.4.4-14+deb7u4
- **ACP config:** apc.max_file_size=5m, apc.shm_size=32M

APPENDIX B: Further Reading

OpenAutonomy reference implementation on SourceForge (including P2PC source):

- http://sourceforge.net/projects/openautonomy/

Open Autonomy Inc. website:

- http://openautonomy.com/

Contact the author (Jeff Disher):

- jeff.disher@openautonomy.com