# The PHP Accelerator 1.2

*By Nick Lindridge*


## *Abstract*

The PHP Accelerator (PHPA) is an extension to PHP that employs compiled code caching and code optimisation techniques to deliver a significant acceleration of PHP scripts. This paper presents an overview of the PHPA design, and how PHPA integrates with PHP.


## *Introduction*

Before executing a PHP script, the PHP engine[1] first reads, parses, and finally translates the contents into compiled code[2] ready for execution. This pre-processing, the result of which only ever changes if the script does, can amount to a substantial proportion of the total script processing time, and is the overhead that PHPA targets for elimination. High performance is gained by using a shared memory cache from which compiled code can be executed directly, as well as a secondary file cache. In addition, PHPA applies some basic code optimisation techniques to improve the efficiency and reduce the size of compiled code.

## *PHP Engine Integration*

The PHP engine has two features that allow PHPA to integrate seamlessly – extensions and mutable behaviour. The extensions mechanism allows libraries to be loaded into the engine at run time, and via an api, for those libraries to be called when certain internal events occur; such events including notification of new requests, completion of code compilation, and request completion. PHPA exploits this to perform compiled code optimisation and caching after a script has been compiled. The engine's mutable behaviour allows alternative routines to be provided for certain operations, such as script reading/compilation, and PHPA exploits this by replacing the standard read/compile operation with a routine to perform cache lookup. Thus PHPA is able to integrate with PHP to handle both code caching and cache lookup.

## *Cache Design*

### Request Handling

Having replaced the default PHP compiler, when called to read and compile a script, PHPA first determines the complete source file path and then obtains information about the file. Rather than use path names for identifying scripts, PHPA internally refers to scripts by the two numbers that uniquely identify every file on Unix systems[3], and for detecting stale cache entries, PHPA notes the scripts last modified time. If the shared memory cache has an up to date version of the script then the compiled code from the shared memory cache is used. If not, then the file cache is examined, and if a cache file exists then the compiled code is deserialised from the cache file. If neither a shared memory cache entry nor a cache file was found, then PHPA calls the default PHP compiler, and PHPA remembers that the result of compilation should then be optimised and cached.

### The File Cache and Code Serialisation

The PHPA file based cache provides an unlimited size cache by serialising compiled code to files, named according to the unique file numbers of their corresponding script file. Code serialisation

[1] Also known as the Zend engine.

[2] The compiled code uses instructions from a virtual instruction set that's platform independent and specialised for PHP rather than for the native CPU. Once compiled, the code is executed by a virtual machine that interprets the virtual machine instructions.

[3] These are the device and inode numbers.

ensures that all relevant memory structures representing the compiled code are written to the cache file, but is complicated by the need to represent associations between different structures. In memory, the links between data structures are simply memory pointers, but memory pointers have no useful meaning once written to files because subsequently restoring data to specific memory addresses is not usually possible. PHPA therefore serialises items with pointers, such as references to strings, by turning pointers into offsets into part of the cache file, and that can be turned back into pointers once the file has been read.

Cache files are read and deserialised by accessing them as memory mapped files, permitting the file to be seen as an area of memory. Items from the file are then copied to allocated memory, and pointers that were turned into offsets during serialisation are turned back into memory pointers once the referenced items have been copied to allocated memory and their addresses are known.

### *The Direct Access Shared Memory Cache*

Whilst the file cache provides effective acceleration by eliminating the repeated parsing and translation of scripts, the necessity to read and deserialise cache files in itself introduces an undesirable overhead. The direct access shared memory cache has no such overhead, and achieves much higher performance.

As its name suggests, the principle of this cache is to store compiled code in shared memory[4], allowing code execution directly from the cache, and requiring no copying or deserialisation into local server memory. To allow this, PHPA ensures that the shared memory is made available at the same memory address in each apache process, and is necessary in order for memory pointers within the shared memory space to be valid for all processes[5]. The cache memory is divided into a process table containing information about the scripts that each server process is executing, a hashtable of cache entries that's keyed by the unique source file number, and a memory pool for allocating memory as necessary for cache entries.

## Cache Locks and Concurrent Reading/Writing

The first version of the shared memory cache used reader/writer locks on the cache to allow either multiple concurrent requests to be executing code or a single writer to be caching a script. This had the requirement that all readers must complete execution before code could be cached, and that if a script was waiting to be cached, any new requests would remain pending until all active requests had completed and until the pending script caching had completed. Whilst this performed well for lightly loaded sites, if a script was modified on a more heavily loaded site, the effect of caching the modified script could be a delay on caching the modified script and a backlog building up of pending requests.

The current cache design is more advanced and addresses the issues of the first design to achieve higher throughput. Rather than use reader/writer locks, a simpler lock is used to only lock the cache while entries are being found, cached, or removed, and not during execution. The design permits cached scripts to be executed while at the same time a script is being cached, and additionally, each cache entry may have multiple versions, allowing a script to be executed whilst simultaneously a new version of the same script is being cached. In this latter case, PHPA marks the current cache entry as being obsolete, and the newly cached version then becomes the current version. The cache maintains a reference count of active requests for each script, and once the reference count reaches zero for an obsolete script, the cache entry is then removed.

---

[4] Shared memory is an area of memory that can be accessed by more than one process.

[5] The Apache Server design has a parent process that spawns children as necessary, and because child processes automatically inherit any shared memory of their parent at the same address, a simple solution to achieve this is for the parent process to acquire the shared memory. This approach, however, isn't adopted by PHPA because it unfortunately complicates or prevents the implementation of some other features.

## Crash Detection

An unfortunate characteristic of the locks that PHPA uses for controlling access to the shared memory cache is that the operating system doesn't automatically release the locks acquired by a process if it crashes. The effect of this being that the server will be unable to server more page requests if a crashed process had locked the cache. To handle this situation, PHPA keeps track of the process ID of any process holding a lock on the cache as well as the cache entries that each process is using, and incorporates crash detection and recovery routines that attempt to ensure a consistent state in the event that a process did crash. To do this, PHPA periodically verifies that server processes known to PHPA still exist, and if PHPA is unable to acquire a lock on the cache, it verifies that the process thought to be holding the lock still exists. If a process that had acquired a lock is found not to exist, then the lock is released, allowing processes to once again acquire a lock. PHPA can still be affected by corruption of the shared memory cache, but the crash detection is effective in being tolerant of the occasional crashes that PHP installations can experience.

## *Code Optimisation*

PHPA performs a number of peephole optimisations on the compiled code to reduce its size and improve performance. Due to the way that compiled code is often generated, such code can be inefficient unless optimisations are made prior to code generation or if the compiler is particularly dumb. For example, the compiled code for 'if' statements without an 'else' clause contains an unnecessary jump to the very next instruction at the end of the 'if' block, and code may contain jump instructions that jump to other jump instructions. The handling of double quoted strings is also inefficient, with the language parser tokenising each word and separator within the string, and code generation producing individual string concatenation instructions. For example, the PHP source below generates the following compiled code of 16 compiled instructions, and shows that string add instructions are generated for each word and word separators of the string to be echoed.

```
<?
        $x = 'inefficient';
        echo "This is rather $x code\n";
?>
```

```
op 0 2 ZEND_FETCH_W str='x' fetch local
op 1 2 ZEND_ASSIGN var* 0 str='inefficient' R1 (unused)
op 2 3 ZEND_INIT_STRING R2
op 3 3 ZEND_ADD_STRING tvar T2 Vconst 'This' R2
op 4 3 ZEND_ADD_STRING tvar T2 Vconst ' ' R2
op 5 3 ZEND_ADD_STRING tvar T2 Vconst 'is' R2
op 6 3 ZEND_ADD_STRING tvar T2 Vconst ' ' R2
op 7 3 ZEND_ADD_STRING tvar T2 Vconst 'rather' R2
op 8 3 ZEND_ADD_STRING tvar T2 Vconst ' ' R2
op 9 3 ZEND_FETCH_R str='x' fetch local R3
op 10 3 ZEND_ADD_VAR T2 var* 3 R2
op 11 3 ZEND_ADD_STRING tvar T2 Vconst ' ' R2
op 12 3 ZEND_ADD_STRING tvar T2 Vconst 'code' R2
op 13 3 ZEND_ADD_CHAR '\n' R2
op 14 3 ZEND_ECHO T2
op 15 5 ZEND_RETURN val=1
```

In contrast, once optimised by PHPA the code size is halved to just 8 instructions, and also eliminates reading the value of variable $X because the value is available as a side effect from the assignment statement of op 1.

```
op 0 2 ZEND_FETCH_W str='x' fetch local
op 1 2 ZEND_ASSIGN var* 0 str='inefficient' R3
op 2 3 ZEND_INIT_STRING R2
op 3 3 ZEND_ADD_STRING tvar T2 Vconst 'This is rather ' R2
op 4 3 ZEND_ADD_VAR T2 var* 3 R2
op 5 3 ZEND_ADD_STRING tvar T2 Vconst ' code\n' R2
op 6 3 ZEND_ECHO T2
op 7 5 ZEND_RETURN val=1
```

Optimisations that are in development optimise this still further by recognising that the value of $X is a constant, and that the echo can be replaced with an echo of a constant string.

Other optimisations include turning post-increment/post-decrement into pre-increment/pre-decrement where possible, elimination of unconditional jumps to other unconditional jumps or return statements, reordering of some code constructs to eliminate jumps, and removing unreachable code.

Further optimisations that are not yet in production versions of PHPA include eliminating reading of variable values wherever possible, and elimination of some variables entirely where more efficient temporary values can be used.

## *Performance Gains*

PHPA is typically able to deliver at least a 2 to 4 times performance gain, although sites that include substantial amounts of code can benefit much further. With scripts completing quicker, and no reading of source files required, PHPA is also effective in achieving significant reductions in load average for heavily loaded sites. Example performance figures and benchmarks are available from http://www.php-accelerator.co.uk/performance.php and via the links from that page. Each page on the http://www.php-accelerator.co.uk site also shows the approximate time to generate the page at the bottom, and contains a link to the same non-accelerated page for comparison.

## *Summary*

The use of a tool such as PHPA can be very effective in reducing the total processing time of PHP scripts, and without sacrificing in any way the dynamic content generation of PHP based sites. Site developers can avoid spending time implementing and debugging coding tricks that minimise the amount of code to be loaded by PHP, but that tend to make code more unmanageable. Sites can make more efficient use of server hardware, and as site load increases, potentially save money from being able to defer the time at which when a hardware upgrade becomes necessary in order to maintain an acceptable level of site performance.