XHProf Documentation (Draft)

Contents

1. Introduction

XHProf is a hierarchical profiler for PHP. It reports function-level call counts and inclusive and exclusive metrics such as wall (elapsed) time, CPU time and memory usage. A function's profile can be broken down by callers or callees. The raw data collection component is implemented in C as a PHP Zend extension called **`xhprof`**. XHProf has a simple HTML based user interface (written in PHP). The browser based UI for viewing profiler results makes it easy to view results or to share results with peers. A callgraph image view is also supported.

XHProf reports can often be helpful in understanding the structure of the code being executed. The hierarchical nature of the reports can be used to determine, for example, what chain of calls led to a particular function getting called.

XHProf supports ability to compare two runs (a.k.a. "diff" reports) or aggregate data from multiple runs. Diff and aggregate reports, much like single run reports, offer "flat" as well as "hierarchical" views of the profile.

XHProf is a light-weight instrumentation based profiler. During the data collection phase, it keeps track of call counts and inclusive metrics for arcs in the dynamic callgraph of a program. It computes exclusive metrics in the reporting/post processing phase. XHProf handles recursive functions by detecting cycles in the callgraph at data collection time itself and avoiding the cycles by giving unique depth qualified names for the recursive invocations.

XHProf's light-weight nature and aggregation capabilities make it well suited for collecting "function-level" performance statistics from production environments. [See additional notes for use in production.]

XHProfLive (not part of the open source kit), for example, is a system-wide performance monitoring system in use at Facebook that is built on top of XHProf. XHProfLive continually gathers function-level profiler data from production tier by running a sample of page requests under XHProf. XHProfLive then aggregates the profile data corresponding to individual requests by various dimensions such a time, page type, and can help answer a variety of questions such as: What is the function-level profile for a specific page? How expensive is function "foo" across all pages, or on a specific page? What functions regressed most in the last hour/day/week? What is the historical trend for execution time of a page/function? and so on.

Originally developed at Facebook, XHProf was open sourced in Mar, 2009.

2. XHProf Overview

XHProf provides:

- **Flat profile** (screenshot)

  Provides function-level summary information such number of calls, inclusive/exclusive wall time, memory usage, and CPU time.

- **Hierarchical profile (Parent/Child View)** (screenshot)

For each function, it provides a breakdown of calls and times per parent (caller) & child (callee), such as:

- what functions call a particular function and how many times?
- what functions does a particular function call?
- The total time spent under a function when called from a particular parent.

- **Diff Reports**

  You may want to compare data from two XHProf runs for various reasons-- to figure out what's causing a regression between one version of the code base to another, to evaluate the performance improvement of a code change you are making, and so on.

  A diff report takes two runs as input and provides both flat function-level diff information, and hierarchical information (breakdown of diff by parent/children functions) for each function.

  The "flat" view (sample screenshot) in the diff report points out the top regressions & improvements.

  Clicking on functions in the "flat" view of the diff report, leads to the "hierarchical" (or parent/child) diff view of a function (sample screenshot). We can get a breakdown of the diff by parent/children functions.

- **Callgraph View** (sample screenshot)

  The profile data can also be viewed as a callgraph. The callgraph view highlights the critical path of the program.

- **Memory Profile**

  XHProf's memory profile mode helps track functions that allocate lots of memory.

  It is worth clarifying that that XHProf doesn't strictly track each allocation/free operation. Rather it uses a more simplistic scheme. It tracks the increase/decrease in the amount of memory allocated to PHP between each function's entry and exit. It also tracks increase/decrease in the amount of **peak** memory allocated to PHP for each function.

- XHProf tracks `include, include_once, require and require_once` operations as if they were functions. The name of the file being included is used to generate the name for these "fake" functions.

Terminology

1. **Inclusive Time (or Subtree Time)**: Includes time spent in the function as well as in descendant functions called from a given function.
2. **Exclusive Time/Self Time**: Measures time spent in the function itself. Does not include time in descendant functions.
3. **Wall Time**: a.k.a. Elapsed time or wall clock time.
4. **CPU Time**: CPU time in user space + CPU time in kernel space

Naming convention for special functions

1. **`main()`**: a fictitious function that is at the root of the call graph.

2. **`load::<filename>`** and **`run_init::<filename>`**:

   XHProf tracks PHP `include/require` operations as function calls.

   For example, an **include "lib/common.php";** operation will result in two XHProf function entries:

   - **`load::lib/common.php`** - This represents the work done by the interpreter to compile/load the file. [Note: If you are using a PHP opcode cache like APC, then the compile only happens on a cache miss in APC.]
   - **`run_init::lib/common.php`** - This represents initialization code executed at the file scope as a result of the include operation.

3. **`foo@<n>`**: Implies that this is a recursive invocation of `foo()`, where `<n>` represents the recursion depth. The recursion may be direct (such as due to `foo()` --> `foo()`), or indirect (such as due to foo() --> `goo()` --> foo()).

Limitations

True hierarchical profilers keep track of a full call stack at every data gathering point, and are later able to answer questions like: what was the cost of the 3rd invokation of foo()? or what was the cost of bar() when the call stack looked like a()->b()->bar()?

XHProf keeps track of only 1-level of calling context and is therefore only able to answer questions about a function looking either 1-level up or 1-level down. It turns out that in practice this is sufficient for most use cases.

To make this more concrete, take for instance the following example.

```
Say you have:
 1 call from a() --> c()
 1 call from b() --> c()
 50 calls from c() --> d()
```

While XHProf can tell you that d() was called from c() 50 times, it cannot tell you how many of those calls were triggered due to a() vs. b(). [We could speculate that perhaps 25 were due to a() and 25 due to b(), but that's not necessarily true.]

In practice however, this isn't a very big limitation.

3. Installing the XHProf Extension

The extension lives in the "extension/" sub-directory.

Note: A windows port hasn't been implemented yet. We have tested `xhprof` on **Linux/FreeBSD** so far.

Version 0.9.2 and above of XHProf is also expected to work on **Mac OS**. [We have tested on Mac OS 10.5.]

Note: XHProf uses the RDTSC instruction (time stamp counter) to implement a really low overhead timer for elapsed time. So at the moment `xhprof` only works on **x86** architecture. Also, since RDTSC values may not be synchronized across CPUs, `xhprof` binds the program to a single CPU during the profiling period.

XHProf's RDTSC based timer functionality doesn't work correctly if **SpeedStep** technology is turned on. This technology is available on some Intel processors. [Note: Mac desktops and laptops typically have SpeedStep turned on by default. To use XHProf, you'll need to disable SpeedStep.]

The steps below should work for Linux/Unix environments.

```
% cd <xhprof_source_directory>/extension/
% phpize
% ./configure --with-php-config=<path to php-config>
% make
% make install
% make test
```

php.ini file: You can update your php.ini file to automatically load your extension. Add the following to your php.ini file.

```
[xhprof]
extension=xhprof.so
;
; directory used by default implementation of the iXHProfRuns
; interface (namely, the XHProfRuns_Default class) for storing
; XHProf runs.
;
xhprof.output_dir=<directory_for_storing_xhprof_runs>
```

4. Profiling using XHProf

Test generating raw profiler data using a sample test program like:

foo.php

```
<?php

function bar($x) {
  if ($x > 0) {
```

```
      bar($x - 1);
    }
  }

  function foo() {
    for ($idx = 0; $idx < 2; $idx++) {
      bar($idx);
      $x = strlen("abc");
    }
  }

  // start profiling
  xhprof_enable();

  // run program
  foo();

  // stop profiler
  $xhprof_data = xhprof_disable();

  // display raw xhprof data for the profiler run
  print_r($xhprof_data);
```

Run the above test program:

```
% php -dextension=xhprof.so foo.php
```

You should get an output like:

```
Array
(
    [foo==>bar] => Array
        (
            [ct] => 2         # 2 calls to bar() from foo()
            [wt] => 27        # inclusive time in bar() when called from foo()
        )

    [foo==>strlen] => Array
        (
            [ct] => 2
            [wt] => 2
        )

    [bar==>bar@1] => Array    # a recursive call to bar()
        (
            [ct] => 1
            [wt] => 2
        )

    [main()==>foo] => Array
        (
            [ct] => 1
            [wt] => 74
        )

    [main()==>xhprof_disable] => Array
        (
            [ct] => 1
            [wt] => 0
        )

    [main()] => Array         # fake symbol representing root
        (
            [ct] => 1
            [wt] => 83
        )

)
```

Note: The raw data only contains "inclusive" metrics. For example, the wall time metric in the raw data represents inclusive time in microsecs. Exclusive times for any function are computed during the analysis/reporting phase.

Note: By default only call counts & elapsed time is profiled. You can optionally also profile CPU time and/or memory usage. Replace,

```
  xhprof_enable();
```

in the above program with, for example:

```
xhprof_enable(XHPROF_FLAGS_CPU + XHPROF_FLAGS_MEMORY);
```

You should now get an output like:

```
Array
(
    [foo==>bar] => Array
        (
            [ct] => 2        # number of calls to bar() from foo()
            [wt] => 37       # time in bar() when called from foo()
            [cpu] => 0       # cpu time in bar() when called from foo()
            [mu] => 2208     # change in PHP memory usage in bar() when called from foo()
            [pmu] => 0       # change in PHP peak memory usage in bar() when called from foo()
        )

    [foo==>strlen] => Array
        (
            [ct] => 2
            [wt] => 3
            [cpu] => 0
            [mu] => 624
            [pmu] => 0
        )

    [bar==>bar@1] => Array
        (
            [ct] => 1
            [wt] => 2
            [cpu] => 0
            [mu] => 856
            [pmu] => 0
        )

    [main()==>foo] => Array
        (
            [ct] => 1
            [wt] => 104
            [cpu] => 0
            [mu] => 4168
            [pmu] => 0
        )

    [main()==>xhprof_disable] => Array
        (
            [ct] => 1
            [wt] => 1
            [cpu] => 0
            [mu] => 344
            [pmu] => 0
        )

    [main()] => Array
        (
            [ct] => 1
            [wt] => 139
            [cpu] => 0
            [mu] => 5936
            [pmu] => 0
        )

)
```

Skipping builtin functions during profiling

By default PHP builtin functions (such as `strlen`) are profiled. If you do not want to profile builtin functions (to either reduce the overhead of profiling further or size of generated raw data), you can use the **`XHPROF_FLAGS_NO_BUILTINS`** flag as in for example:

```
// do not profile builtin functions
xhprof_enable(XHPROF_FLAGS_NO_BUILTINS);
```

Ignoring specific functions during profiling (0.9.2 or higher)

Starting with release 0.9.2 of xhprof, you can tell XHProf to ignore a specified list of functions during profiling. This

allows you to ignore, for example, functions used for indirect function calls such as `call_user_func` and `call_user_func_array`. These intermediate functions unnecessarily complicate the call hierarchy and make the XHProf reports harder to interpret since they muddle the parent-child relationship for functions called indirectly.

To specify the list of functions to be ignored during profiling use the 2nd (optional) argument to `xhprof_enable`. For example,

```
// elapsed time profiling; ignore call_user_func* during profiling
xhprof_enable(0,
              array('ignored_functions' =>  array('call_user_func',
                                                  'call_user_func_array')));

or,

// elapsed time + memory profiling; ignore call_user_func* during profiling
xhprof_enable(XHPROF_FLAGS_MEMORY,
               array('ignored_functions' =>  array('call_user_func',
                                                   'call_user_func_array')));
```

5. Setting up XHProf UI

1. **PHP source structure**

   The XHProf UI is implemented in PHP. The code resides in two subdirectories, `xhprof_html/` and `xhprof_lib/`.

   The `xhprof_html` directory contains the 3 top-level PHP pages.

   - `index.php`: For viewing a single run or diff report.
   - `callgraph.php`: For viewing a callgraph of a XHProf run as an image.
   - `typeahead.php`: Used implicitly for the function typeahead form on a XHProf report.

   The `xhprof_lib` directory contains supporting code for display as well as analysis (computing flat profile info, computing diffs, aggregating data from multiple runs, etc.).

2. **Web server config:** You'll need to make sure that the `xhprof_html/` directory is accessible from your web server, and that your web server is setup to serve PHP scripts.

3. **Managing XHProf Runs**

   Clients have flexibility in how they save the XHProf raw data obtained from an XHProf run. The XHProf UI layer exposes an interface iXHProfRuns (see xhprof_lib/utils/xhprof_runs.php) that clients can implement. This allows the clients to tell the UI layer how to fetch the data corresponding to a XHProf run.

   The XHProf UI libaries come with a default file based implementation of the iXHProfRuns interface, namely "XHProfRuns_Default" (also in xhprof_lib/utils/xhprof_runs.php). This default implementation stores runs in the directory specified by **xhprof.output_dir** INI parameter.

   A XHProf run must be uniquely identified by a namespace and a run id.

   a) Saving XHProf data persistently:

   Assuming you are using the default implementation **`XHProfRuns_Default`** of the **`iXHProfRuns`** interface, a typical XHProf run followed by the save step might look something like:

   ```
   // start profiling
   xhprof_enable();

   // run program
   ....

   // stop profiler
   $xhprof_data = xhprof_disable();
   ```

```
//
// Saving the XHProf run
// using the default implementation of iXHProfRuns.
//
include_once $XHPROF_ROOT . "/xhprof_lib/utils/xhprof_lib.php";
include_once $XHPROF_ROOT . "/xhprof_lib/utils/xhprof_runs.php";

$xhprof_runs = new XHProfRuns_Default();

// Save the run under a namespace "xhprof_foo".
//
// **NOTE**:
// By default save_run() will automatically generate a unique
// run id for you. [You can override that behavior by passing
// a run id (optional arg) to the save_run() method instead.]
//
$run_id = $xhprof_runs->save_run($xhprof_data, "xhprof_foo");

echo "---------------\n".
     "Assuming you have set up the http based UI for \n".
     "XHProf at some address, you can view run at \n".
     "http://<xhprof-ui-address>/index.php?run=$run_id&source=xhprof_foo\n".
     "---------------\n";
```

The above should save the run as a file in the directory specified by the **`xhprof.output_dir`** INI parameter. The file's name might be something like **`49bafaa3a3f66.xhprof_foo`**; the two parts being the run id ("49bafaa3a3f66") and the namespace ("xhprof_foo"). [If you want to create/assign run ids yourself (such as a database sequence number, or a timestamp), you can explicitly pass in the run id to the `save_run` method.

b) Using your own implementation of iXHProfRuns

If you decide you want your XHProf runs to be stored differently (either in a compressed format, in an alternate place such as DB, etc.) database, you'll need to implement a class that implements the iXHProfRuns() interface.

You'll also need to modify the 3 main PHP entry pages (index.php, callgraph.php, typeahead.php) in the "xhprof_html/" directory to use the new class instead of the default class `XHProfRuns_Default`. Change this line in the 3 files.

```
$xhprof_runs_impl = new XHProfRuns_Default();
```

You'll also need to "include" the file that implements your class in the above files.

4. **Accessing runs from UI**

a) Viewing a Single Run Report

To view the report for run id say <run_id> and namespace <namespace> use a URL of the form:

```
http://<xhprof-ui-address>/index.php?run=<run_id>&source=<namespace>
```

For example,

```
http://<xhprof-ui-address>/index.php?run=49bafaa3a3f66&source=xhprof_foo
```

b) Viewing a Diff Report

To view the report for run ids say <run_id1> and <run_id2> in namespace <namespace> use a URL of the form:

```
http://<xhprof-ui-address>/index.php?run1=<run_id1>&run2=<run_id2>&source=<namespace>
```

c) Aggregate Report

You can also specify a set of run ids for which you want an aggregated view/report.

Say you have three XHProf runs with ids 1, 2 & 3 in namespace "benchmark". To view an aggregate report of these runs:

```
http://<xhprof-ui-address>/index.php?run=1,2,3&source=benchmark
```

Weighted aggregations: Further suppose that the above three runs correspond to three types of programs

p1.php, p2.php and p3.php that typically occur in a mix of 20%, 30%, 50% respectively. To view an aggregate report that corresponds to a weighted average of these runs using:

```
http://<xhprof-ui-address>/index.php?run=1,2,3&wts=20,30,50&source=benchmark
```

6. Notes on using XHProf in production

Some observations/guidelines. Your mileage may vary:

- CPU timer (getrusage) on Linux has high overheads. It is also coarse grained (millisec accuracy rather than microsec level) to be useful at function level. Therefore, the skew in reported numbers when using XHPROF_FLAGS_CPU mode tends to be higher.

  We recommend using elapsed time + memory profiling mode in production. [Note: The additional overhead of memory profiling mode is really low.]

  ```
  // elapsed time profiling (default) + memory profiling
  xhprof_enable(XHPROF_FLAGS_MEMORY);
  ```

- Profiling a random sample of pages/requests works well in capturing data that is representative of your production workload.

  To profile say 1/10000 of your requests, instrument the beginning of your request processing with something along the lines of:

  ```
  if (mt_rand(1, 10000) == 1) {
    xhprof_enable(XHPROF_FLAGS_MEMORY);
    $xhprof_on = true;
  }
  ```

  At the end of the request (or in a request shutdown function), you might then do something like:

  ```
  if ($xhprof_on) {
    // stop profiler
    $xhprof_data = xhprof_disable();

    // save $xhprof_data somewhere (say a central DB)
    ...
  }
  ```

  You can then rollup/aggregate these individual profiles by time (e.g., 5 minutely/hourly/daily basis), page/request type,or other dimensions using **`xhprof_aggregate_runs()`**.

7. Lightweight Sampling Mode

The xhprof extension also provides a very light weight **sampling mode**. The sampling interval is 0.1 secs. Samples record the full function call stack. The sampling mode can be useful if an extremely low overhead means of doing performance monitoring and diagnostics is desired.

The relevant functions exposed by the extension for using the sampling mode are **`xhprof_sample_enable()`** and **`xhprof_sample_disable()`**.

[**TBD**: more detailed documentation on sampling mode.]

8. Additional Features

The **`xhprof_lib/utils/xhprof_lib.php`** file contains additional library functions that can be used for manipulating/ aggregating XHProf runs.

For example:

- **`xhprof_aggregate_runs()`**: can be used to aggregate multiple XHProf runs into a single run. This can be helpful for building a system-wide "function-level" performance monitoring tool using XHProf. [For example, you might to roll up XHProf runs sampled from production periodically to generate hourly, daily, reports.]

- **`xhprof_prune_run()`**: Aggregating large number of XHProf runs (especially if they correspond to different types of programs) can result in the callgraph size becoming too large. You can use `xhprof_prune_run` function to prune the callgraph data by editing out subtrees that account for a very small portion of the total time.

9. Dependencies

- **JQuery Javascript**: For tooltips and function name typeahead, we make use of JQuery's javascript libraries. JQuery is available under both a MIT and GPL licencse (http://docs.jquery.com/Licensing). The relevant JQuery code, used by XHProf, is in the `xhprof_html/jquery` subdirectory.
- **dot (image generation utility):** The callgraph image visualization ([View Callgraph]) feature relies on the presence of Graphviz "dot" utility in your path. "dot" is a utility to draw/generate an image for a directed graph.

10. Acknowledgements

The HTML-based navigational interface for browsing profiler results is inspired by that of a similar tool that exists for Oracle's stored procedure language, PL/SQL. But that's where the similarity ends; the internals of the profiler itself are quite different.