MediaWiki Performance Techniques

Amsterdam Hackathon 2013

Performance optimisation defined

- Two things we wish to minimise:
  - Latency in user experience
  - Hardware capacity requirements (throughput)
- Each metric suggests a different approach

Performance optimisation defined

- Latency:
  - Identify and eliminate causes of long request times.
  - Request service time <100ms is "good enough", human perception gives diminishing returns.
- Throughput:
  - Collect aggregate data on heaviest users of CPU, RAM, network and disk.
  - Trade-off between hardware cost and software development cost.
  - Stop optimising when the time spent fails to justify the reduced hardware expenditure.

Throughput analysis

- Each limited resource should be treated separately:
  - Apache CPU
  - MySQL CPU
  - Peak memory usage
  - Network volume
  - Disk I/O
  - Lock *X* held, Lock *Y* held, ...

Wall clock time

- Time as measured by the clock on the wall
- A good approximation to latency, but a poor approximation to hardware capacity.
- Example: disk seeks
  - As load increases, average seek distance becomes shorter, and reads from the same track become more common
  - Wall clock time at low load gives a poor indication of maximum capacity at high load

CPU time

- Amount of time a CPU core spent executing the process in question (as opposed to waiting for some other resource)
- Includes system memory latency
- Easily measured with profiling tools

Profiling tools

- MediaWiki's profiler
- XDebug / KCacheGrind
- xhprof
- perf
- microtime()

MediaWiki's profiler

- Advantages:
  - Section labels and lengths can be customised
  - Can include application-level information in section name, like wfGetCaller()
  - Suitable for production
- Disadvantages:
  - High overhead
  - Need to explicitly mark out sections with wfProfileIn()
  - Double-counts recursive functions

MediaWiki's profiler

Filter

Name	Time (%)	Memory (%)	Count	Calls/req	ms/call	kb/call	ms/req	kb/req
-total	100%	100%	1	1	495.95	15810.94	495.95	15810.94
MediaWiki::main	44.57%	55.26%	1	1	221.02	8736.57	221.02	8736.57
SQL Queries [+]	38.71%	0%	0	0	0	0	191.97	0
OutputPage::output	25.03%	24.82%	1	1	124.13	3923.55	124.13	3923.55
Output-skin	24.64%	22.69%	1	1	122.21	3588	122.21	3588
SkinTemplate::outputPage [+]	24.61%	23.42%	1	1	122.07	3703.44	122.07	3703.44
-overhead-total	24.25%	10.45%	708	708	0.17	2.33	120.29	1651.74
DeferredUpdates::doUpdates	18.39%	0.01%	1	1	91.21	2.27	91.21	2.27
MediaWiki::performRequest	17.62%	23.03%	1	1	87.4	3641.78	87.4	3641.78
MediaWiki::performAction	14.68%	18.38%	1	1	72.8	2906.68	72.8	2906.68
Article::view	14.5%	18.3%	1	1	71.9	2894.16	71.9	2894.16
JobQueue::isEmpty	9.48%	0.06%	11	11	4.28	0.89	47.03	9.76
MessageCache::load [+]	9.47%	1.75%	1	1	46.96	276.42	46.96	276.42
Skin::initPage	7.56%	0.03%	1	1	37.52	5.17	37.52	5.17
LinkBatch::executeInto	7.44%	0.05%	1	1	36.88	8.54	36.88	8.54
LinkBatch::doQuery	7.07%	0.04%	1	1	35.06	5.98	35.06	5.98
MagicWord::load	2.98%	3.28%	148	148	0.1	3.5	14.77	518.56

XDebug / KCachegrind

- Advantages:
  - Times every PHP function
  - Awesome visualisation
- Disadvantages:
  - Crashy

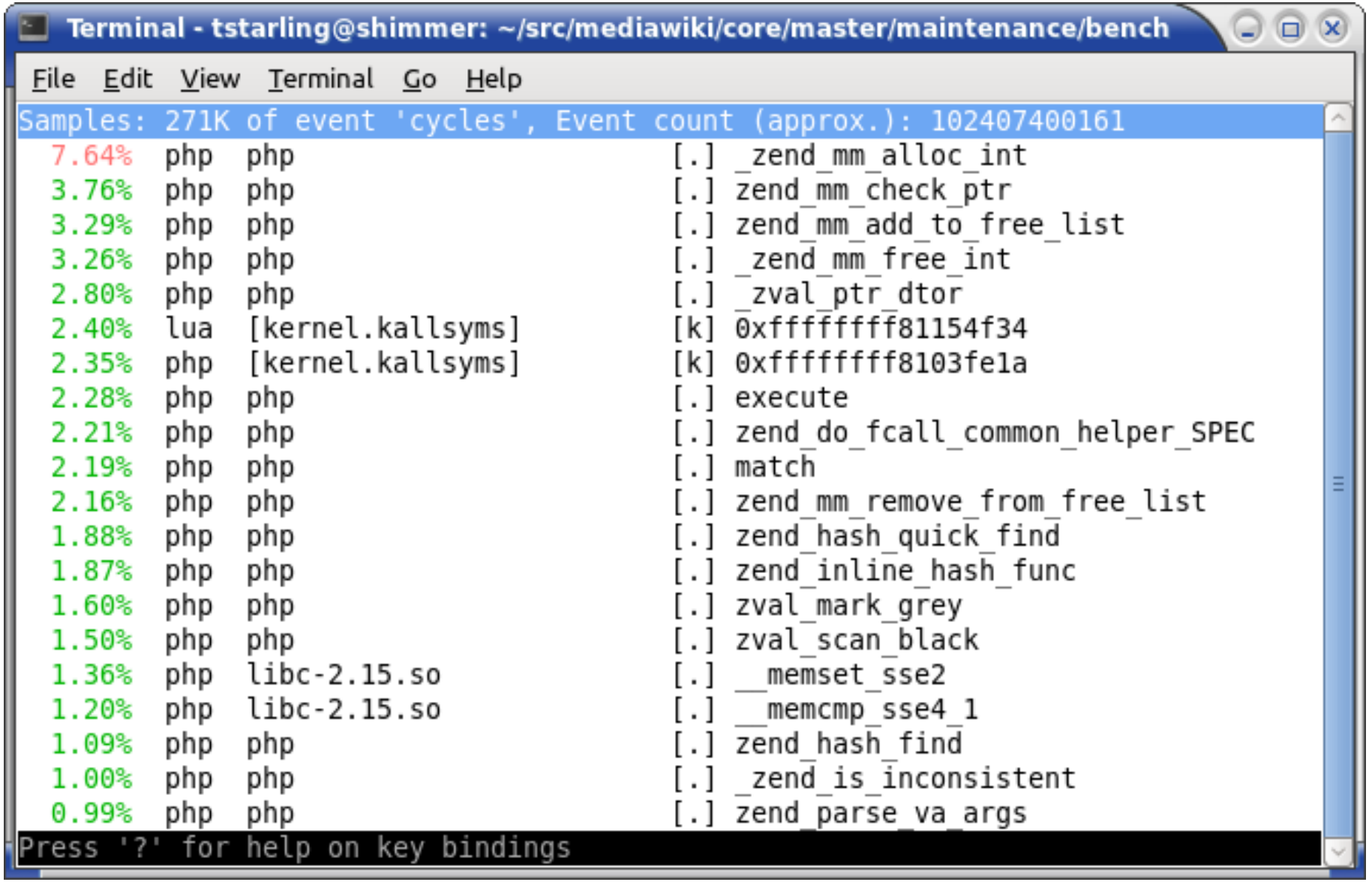
xhprof

- Advantages:
  - Times every PHP function
- Disadvantages:
  - Buggy
  - Web interface full of XSS vulnerabilities

perf

- A lower level (C function) view of process or system performance
- Replaces gprof
- Available in `linux-tools-common`

perf


Terminal - tstarling@shimmer: ~/src/mediawiki/core/master/maintenance/bench
File Edit View Terminal Go Help
Samples: 271K of event 'cycles', Event count (approx.): 102407400161
7.64% php php [.] _zend_mm_alloc_int
3.76% php php [.] zend_mm_check_ptr
3.29% php php [.] zend_mm_add_to_free_list
3.26% php php [.] _zend_mm_free_int
2.80% php php [.] _zval_ptr_dtor
2.40% lua [kernel.kallsyms] [k] 0xffffffff81154f34
2.35% php [kernel.kallsyms] [k] 0xffffffff8103fe1a
2.28% php php [.] execute
2.21% php php [.] zend_do_fcall_common_helper_SPEC
2.19% php php [.] match
2.16% php php [.] zend_mm_remove_from_free_list
1.88% php php [.] zend_hash_quick_find
1.87% php php [.] zend_inline_hash_func
1.60% php php [.] zval_mark_grey
1.50% php php [.] zval_scan_black
1.36% php libc-2.15.so [.] __memset_sse2
1.20% php libc-2.15.so [.] __memcmp_sse4_1
1.09% php php [.] zend_hash_find
1.00% php php [.] _zend_is_inconsistent
0.99% php php [.] zend_parse_va_args
Press '?' for help on key bindings

report.py

- Simple aggregation of production profiling

Graphite

- Flexible time series graphing system for production profiling

microtime()

- Best for micro-optimisation
- Good stability of results

```
$ /usr/local/php-fast/bin/php eval.php
> wfMessage('1movedto2')->plain()

> $t = microtime(true); for ($i=0; $i<100000; $i++)
{wfMessage('1movedto2')->plain();} print microtime(true)-$t;
2.3719320297241

> $t = microtime(true); for ($i=0; $i<100000; $i++)
{wfMessage('1movedto2')->plain();} print microtime(true)-$t;
2.3795449733734
```

warm cache

24µs per call

Micro-optimisation

- Improve performance by optimising fast but frequently-called functions
- Minimise function call count
  - 3µs per call is more expensive than just about anything
  - Reduce abstraction
  - Replace functions with operators, e.g. substr($s, $i,1) with $s[$i]
  - Save invariant function call results in local variables

Micro-optimisation

```
diff --git a/includes/Message.php b/includes/Message.php
index 531551d..2bc72c15 100644
--- a/includes/Message.php
+++ b/includes/Message.php
@@ -481,7 +481,9 @@ class Message {
         }

         # Replace parameters before text parsing
-        $string = $this->replaceParameters( $string, 'before' );
+        if ( $this->parameters ) {
+            $string = $this->replaceParameters( $string,
'before' );
+        }

         # Maybe transform using the full parser
         if ( $this->format === 'parse' ) {
```

- And one other identical change for $type='after'

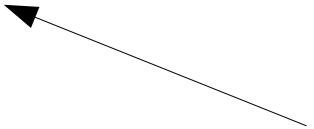
Micro-optimisation

```
$ /usr/local/php-fast/bin/php eval.php
> wfMessage('1movedto2')->plain()

> $t = microtime(true); for ($i=0; $i<100000; $i++)
{wfMessage('1movedto2')->plain();} print microtime(true)-$t;
2.0254280567169

> $t = microtime(true); for ($i=0; $i<100000; $i++)
{wfMessage('1movedto2')->plain();} print microtime(true)-$t;
2.0223109722137

> print (2.3795449733734 - 2.0223109722137) / 2.3795449733734
0.15012702224882
```



15% improvement for 5 minutes of work

Macro-optimisation

- Cache the results of expensive (>100ms) operations
- Avoid or defer unnecessary work
- Use an algorithm with an appropriate time order

```
// Split $s into lines with O(N^2) time order
$lines = array();
while ( strlen( $s ) ) {
    $nlPos = strpos( $s, "\n" );
    $lines[] = substr( $s, 0, $nlPos );
    $s = substr( $s, $nlPos + 1 );        // O(N)
}
```

PHP memory optimisation

- Arrays are expensive

```
$ gdb /usr/local/php-5.4.12-slow/bin/php
(gdb) print sizeof(Bucket)
$1 = 72
(gdb) print sizeof(HashTable)
$2 = 72
```

- Objects are expensive

```
(gdb) print sizeof(zend_object) +
2*sizeof(HashTable)
$3 = 176
```

- Even variables are expensive (compared to C, anyway)

```
(gdb) print sizeof(zval)
$4 = 24
```

PHP memory optimisation

- Use iterators to avoid large array storage

```
foreach ( StringUtils::explode( "\n", $s ) as $line ) {
    ...
}
```

- Use MySQL result objects directly
- Limit user input size where possible

SQL optimisation

- Tends to be more theoretical, since measurement is harder
- Minimise:
  - Number of rows scanned
  - Lock acquisition rate
  - Lock hold time
  - Index size

Number of rows scanned

- Impacts CPU.
- Impacts memory usage due to COW references acquired.
- Impacts disk read rate and cache size requirements.

Number of rows scanned

- 100 rows: usually OK
- 100,000 rows: usually not OK
- Common culprits:
  - SELECT COUNT(*)
  - Partially unindexed queries

Locks

- Locks are on index nodes
- Acquired by write queries
- Released by COMMIT queries

```
// lock indexes referenced in $conds
$dbw->update( 'foo', $conds, $updates );
// hold for a while
sleep( 1 );
// release lock
$dbw->commit();
```

Locks

- Lock contention occurs when:

$$R_{req} \gtrsim \frac{1}{T_{hold}}$$

- Where:
  - R_{req} is the rate at which the lock is requested
  - T_{hold} is the time for which the lock is held
- The problem can be approached either by reducing the rate, or by reducing the hold time

Locks

- In MW 1.20, Aaron introduced Database::onTransactionIdle(), which is an excellent tool for reducing lock hold times.

```
$dbw->onTransactionIdle( function() use ( $dbw, $method ) {
    global $wgRCMaxAge;

    $cutoff = $dbw->timestamp( time() - $wgRCMaxAge );
    $dbw->delete(
        'recentchanges',
        array( 'rc_timestamp < ' . $dbw->addQuotes( $cutoff ) ),
        $method
    );
} );
```

- The callback is invoked with the DB in autocommit mode

Index size

- Performance declines rapidly when indexes cannot fit in RAM
- Index on integers instead of strings where possible
- Remove or reject features which require large index sizes
- Use BINARY/VARBINARY not CHAR/VARCHAR

Getting your code deployed

- Write efficient code
- Choose awesome features that justify extra hardware expenditure
- Think about how your code will behave at scale