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
<table>
<thead>
<tr>
<th>Name</th>
<th>Time (%)</th>
<th>Memory (%)</th>
<th>Count</th>
<th>Calls/req</th>
<th>ms/call</th>
<th>kb/call</th>
<th>ms/req</th>
<th>kb/req</th>
</tr>
</thead>
<tbody>
<tr>
<td>-total</td>
<td>100%</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>495.95</td>
<td>15810.94</td>
<td>495.95</td>
<td>15810.94</td>
</tr>
<tr>
<td>MediaWiki::main</td>
<td>44.57%</td>
<td>55.26%</td>
<td>1</td>
<td>1</td>
<td>221.02</td>
<td>8736.57</td>
<td>221.02</td>
<td>8736.57</td>
</tr>
<tr>
<td>SQL Queries [+]</td>
<td>38.71%</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-overhead-total</td>
<td>24.03%</td>
<td>24.82%</td>
<td>1</td>
<td>1</td>
<td>124.13</td>
<td>3923.55</td>
<td>124.13</td>
<td>3923.55</td>
</tr>
<tr>
<td>Output::output</td>
<td>24.64%</td>
<td>22.69%</td>
<td>1</td>
<td>1</td>
<td>122.21</td>
<td>3588</td>
<td>122.21</td>
<td>3588</td>
</tr>
<tr>
<td>Output-skin</td>
<td>24.61%</td>
<td>23.42%</td>
<td>1</td>
<td>1</td>
<td>122.07</td>
<td>3703.44</td>
<td>122.07</td>
<td>3703.44</td>
</tr>
<tr>
<td>SkinTemplate::outputPage [+]</td>
<td>24.25%</td>
<td>10.45%</td>
<td>708</td>
<td>708</td>
<td>0.17</td>
<td>2.33</td>
<td>120.29</td>
<td>1651.74</td>
</tr>
<tr>
<td>-DeferredUpdates::doUpdates</td>
<td>18.39%</td>
<td>0.01%</td>
<td>1</td>
<td>1</td>
<td>91.21</td>
<td>2.27</td>
<td>91.21</td>
<td>2.27</td>
</tr>
<tr>
<td>MediaWiki::performRequest</td>
<td>17.62%</td>
<td>23.03%</td>
<td>1</td>
<td>1</td>
<td>87.4</td>
<td>3641.78</td>
<td>87.4</td>
<td>3641.78</td>
</tr>
<tr>
<td>MediaWiki::performAction</td>
<td>14.68%</td>
<td>18.38%</td>
<td>1</td>
<td>1</td>
<td>72.8</td>
<td>2906.68</td>
<td>72.8</td>
<td>2906.68</td>
</tr>
<tr>
<td>Article::view</td>
<td>14.5%</td>
<td>18.3%</td>
<td>1</td>
<td>1</td>
<td>71.9</td>
<td>2894.16</td>
<td>71.9</td>
<td>2894.16</td>
</tr>
<tr>
<td>JobQueue::isEmpty</td>
<td>9.48%</td>
<td>0.06%</td>
<td>11</td>
<td>11</td>
<td>4.28</td>
<td>0.89</td>
<td>47.03</td>
<td>9.76</td>
</tr>
<tr>
<td>MessageCache::load [+]</td>
<td>9.47%</td>
<td>1.75%</td>
<td>1</td>
<td>1</td>
<td>46.96</td>
<td>276.42</td>
<td>46.96</td>
<td>276.42</td>
</tr>
<tr>
<td>Skin::initPage</td>
<td>7.56%</td>
<td>0.03%</td>
<td>1</td>
<td>1</td>
<td>37.52</td>
<td>5.17</td>
<td>37.52</td>
<td>5.17</td>
</tr>
<tr>
<td>LinkBatch::executeInto</td>
<td>7.44%</td>
<td>0.05%</td>
<td>1</td>
<td>1</td>
<td>36.88</td>
<td>8.54</td>
<td>36.88</td>
<td>8.54</td>
</tr>
<tr>
<td>LinkBatch::doQuery</td>
<td>7.07%</td>
<td>0.04%</td>
<td>1</td>
<td>1</td>
<td>35.06</td>
<td>5.98</td>
<td>35.06</td>
<td>5.98</td>
</tr>
<tr>
<td>MagicWord::load</td>
<td>2.98%</td>
<td>3.28%</td>
<td>148</td>
<td>148</td>
<td>0.1</td>
<td>3.5</td>
<td>14.77</td>
<td>518.56</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Line</th>
<th>Function</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.64%</td>
<td>_zend_mm_alloc_int</td>
<td></td>
</tr>
<tr>
<td>3.76%</td>
<td>_zend_mm_check_ptr</td>
<td></td>
</tr>
<tr>
<td>3.29%</td>
<td>_zend_mm_add_to_free_list</td>
<td></td>
</tr>
<tr>
<td>3.26%</td>
<td>_zend_mm_free_int</td>
<td></td>
</tr>
<tr>
<td>2.80%</td>
<td>_zval_ptr_dtor</td>
<td></td>
</tr>
<tr>
<td>2.40%</td>
<td>[kernel.kallsyms]</td>
<td></td>
</tr>
<tr>
<td>2.35%</td>
<td>[kernel.kallsyms]</td>
<td></td>
</tr>
<tr>
<td>2.28%</td>
<td>execute</td>
<td></td>
</tr>
<tr>
<td>2.21%</td>
<td>_zend_do_fcall_common_helper_SPEC</td>
<td></td>
</tr>
<tr>
<td>2.19%</td>
<td>match</td>
<td></td>
</tr>
<tr>
<td>2.16%</td>
<td>_zend_mm_remove_from_free_list</td>
<td></td>
</tr>
<tr>
<td>1.88%</td>
<td>_zend_hash_quick_find</td>
<td></td>
</tr>
<tr>
<td>1.87%</td>
<td>_zend_inline_hash_func</td>
<td></td>
</tr>
<tr>
<td>1.60%</td>
<td>zval_mark_grey</td>
<td></td>
</tr>
<tr>
<td>1.50%</td>
<td>zval_scan_black</td>
<td></td>
</tr>
<tr>
<td>1.36%</td>
<td>__memset_sse2</td>
<td></td>
</tr>
<tr>
<td>1.20%</td>
<td>__memcmp_sse4_1</td>
<td></td>
</tr>
<tr>
<td>1.09%</td>
<td>_zend_hash_find</td>
<td></td>
</tr>
<tr>
<td>1.00%</td>
<td>_zend_is_inconsistent</td>
<td></td>
</tr>
<tr>
<td>0.99%</td>
<td>_zend_parse_va_args</td>
<td></td>
</tr>
</tbody>
</table>
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

```php
$ /usr/local/php-fast/bin/php eval.php
> $t = microtime(true); for ($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
```

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

```php
// 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
  ```php
  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

```php
// 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:

\[ R_{req} \geq \frac{1}{T_{hold}} \]

- 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.

```php
$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