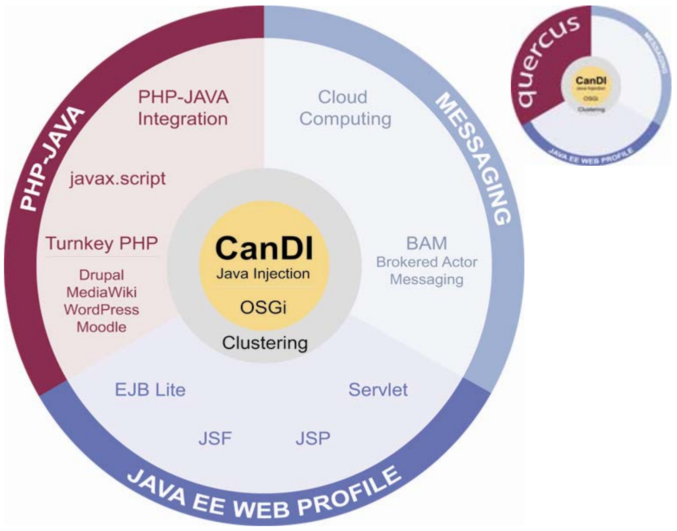
By Nam Nguyen I Software Engineer I nam@caucho.com

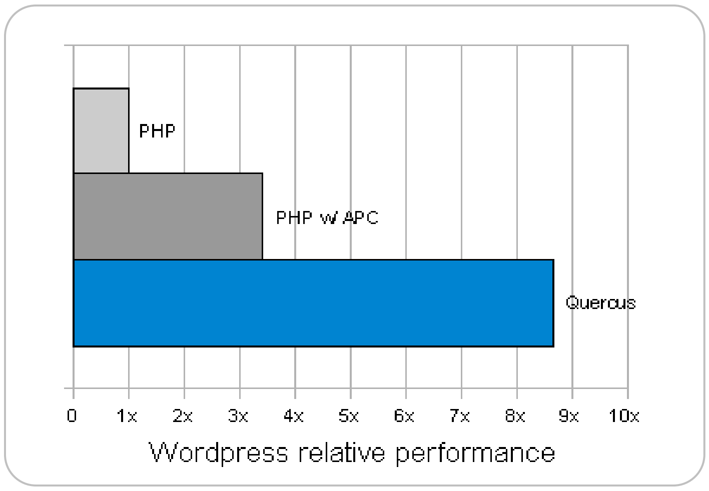
caucho reliable open source

Faster PHP Through Java



ABSTRACT

Using benchmarks and code samples throughout to anchor the discussion, this paper explores how Quercus -- Caucho's Java implementation of PHP -- can perform faster than the standard Apache-based PHP both in synthetic benchmarks and real-world applications. The results are quite impressive.



1. INTRODUCTION

PHP Performance Challenges

The prototypical Apache/MySQL/PHP architecture stack is the root of PHP performance limitations (see Figure 1). Each request that travels to Apache gets its own process that is more or less independent from each other. The processes do not automatically talk with each other and caching is very limited. Because of Apache's process module, PHP can not take advantage of the numerous opportunities to cache frequently used data (which is unfortunate because caching is a universal method to easily increase performance by several magnitudes).

Furthermore, as one tries to scale an application to several commodity PHP/Apache servers, there is one component that doesn't scale very well: the database. The typical PHP application stores everything in the database including the address of the site, the name of the site, user permissions, and even cached results of queried database data. More servers mean ever greater loads on the database and eventually, the database will become the bottleneck.

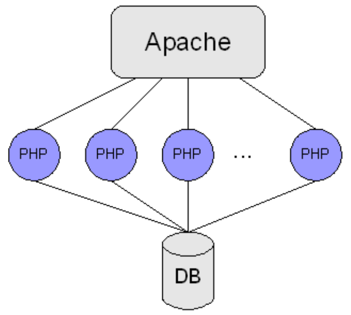


Figure 1: Apache/MySQL/PHP Stack

The PHP/Apache process model further adds load to the database. Each process is wholly responsible for connecting and authenticating with the database on each and every request. This handshake can consume a significant portion of the request time. For many connections to the database, this constant stream of connects and reconnects can severely strain the database and negatively impact the whole performance of the application.

PHP Needs to be Taken Seriously

PHP powers a staggering 32.84% of the world wide web[1], which makes it an important player in the web server space. According to Alexa.com as of December 2008, three out of the top ten sites in the entire world (Yahoo!, Facebook, Wikipedia) are using PHP for their main infrastructure.

In addition, virtually every hosting solution provider has a PHP hosting plan. Compared to Java's rate of adoption in this area, it is a no-contest in PHP's favor. A major factor for this popularity is PHP's simplicity and ease-of-use. PHP's C-like syntax and its extensive libraries makes it very easy for novices and professionals alike to create a dynamic website that communicates with a database in no time. Unfortunately, we cannot say the same about Java.

Quick Introduction to Quercus

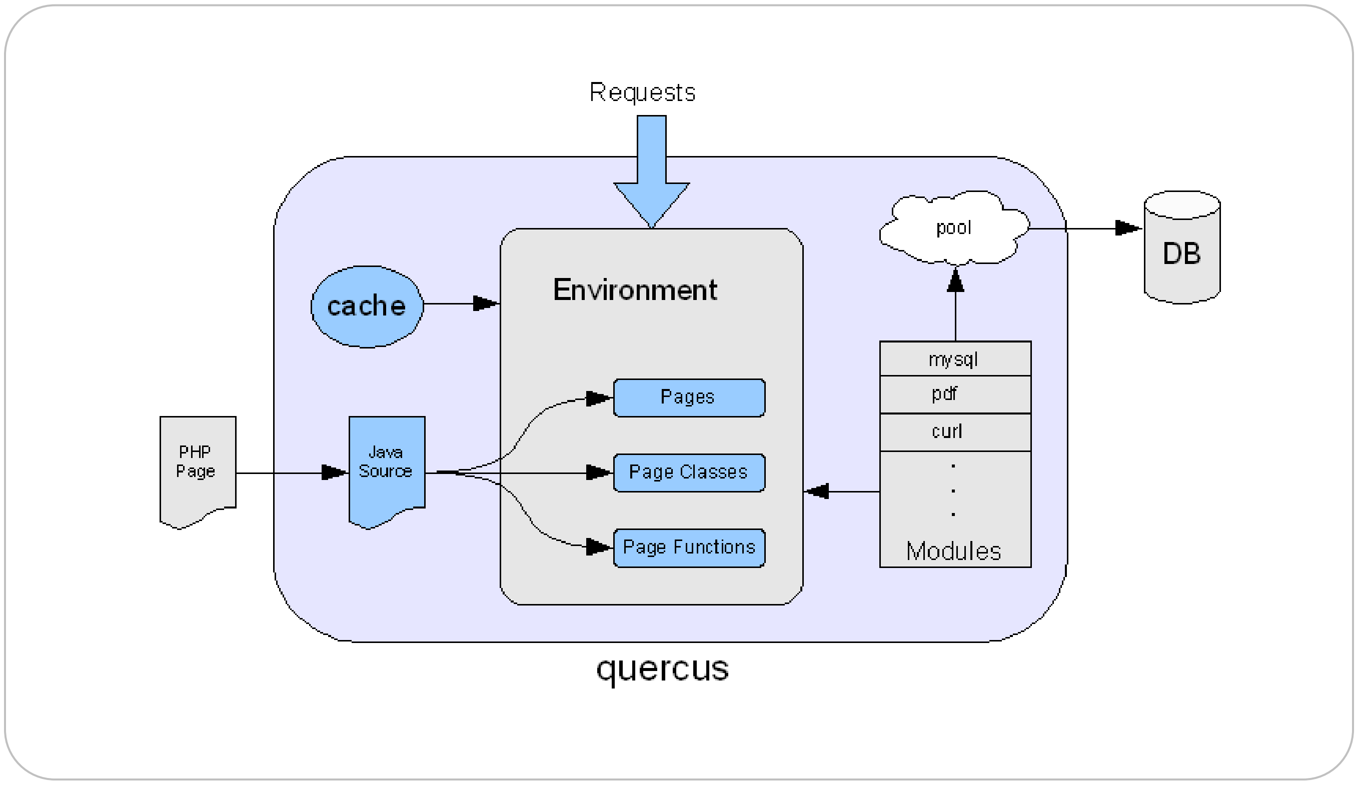
Quercus aims to solve the performance problems of PHP by completely re-implementing the standard PHP interpreter in Java. Quercus is coded entirely in Java and it can take advantage of being run in a long-lived environment of the Java Virtual Machine (JVM). There are countless opportunities for Quercus to cache reoccurring data and state like a pool for database connections to virtually eliminate the overhead of the database connection handshake.

1. http://www.nexen.net/chiffres_cles/phpversion/18824-php_statistics_for_october_2008.php

2. IT'S ALL ABOUT PERFORMANCE

Quercus Architecture

Quercus runs in Resin or any other Java Application Server as a servlet. Quercus compiles PHP sources into equivalent Java sources. Those Java sources are then compiled down into Java bytecodes.



One of the reasons for Quercus' blazing performance is that it is built from the ground up in 100% Java. Everything from start to finish is in Java and Quercus benefits from the long-lived environment of the JVM. Unike PHP on Apache, Quercus threads come from the global thread pool of the application server. The threads are not isolated and Quercus can cache items like strings, serialized data, function definitions, regular expressions, etc. across requests. These things are not normally cached on a standard PHP/Apache server and the caching of these items gives a considerable speed boost. Furthermore, Quercus uses the database connection pool of the Java application server. This saves the application from the overhead of connection initialization on every request and prevents the database from reaching its maximum open connection limit.

Performance Optimizations

We have invested a considerable amount of time in making Quercus fast. The following is a partial list of optimizations that are in Quercus:

- compilation to Java source code
- static code analysis
- direct function calls
- lazy array and string copying
- lazy loading of user-defined functions for faster start-up
- precomputed hashes and keys for strings (for array performance)
- custom regular expression library built from the ground up (independent of *java.util.regexp*)
- libraries/modules in pure Java
- page cache
- file lookup cache
- regular expression program cache
- serialize/unserialize cache
- database connection pooling

Sample Generated Code

Quercus does static code analysis before compiling PHP sources to the equivalent Java sources. The analysis ensures that the generated Java code are efficient. Table A shows a highly optimized code sample from Quercus.

Table A: Generated Code Sample

PHP source	Generated Java source

`<?php`
`    function foo($count)`
`    {`
`        for ($i = 0; $i < $count; $i++) {`
`        }`
`     }`
`?>`
	`public final Value call(Env env, Value p_count)`
	` {`
	`     env.checkTimeout();`
	`     Value v_count = p_count.toArgValue();`
	`     long v_i = 0;`
	
	`     for (v_i = 0; v_i < v_count.toDouble(); v_i = v_i + 1) {`
	`         env.checkTimeout();`
	`     }`
	`     return com.caucho.quercus.env.NullValue.NULL;`
	`}`

For all variables, Quercus determines the type, scope, whether the variable has been assigned, and to what type it was assigned to. With this information in hand, Quercus can compile variables into Java primitive types. In the *for* loop above, variable *$i* becomes a Java long and assignments to it are marshaled to its primitive type.

Synthetic Benchmarks

Because of just-in-time compilation and hand-tuned optimizations, Quercus is very speedy in benchmarks. Table A shows the results from running a standard PHP benchmark available on php.net[2].

Table B: Time in seconds, lower is better

	PHP 5.2.6 w/ APC	Quercus	Faster by
Simple	0.253	0.047	**440%**
simplecall	0.345	0.016	**2100%**
simpleucall	0.567	0.047	**1100%**
simpleudcall	0.732	0.031	**2300%**
Mandel	0.665	0.047	**1300%**
mandel2	0.834	0.031	**2600%**
ackermann	0.872	0.016	**5400%**
Ary	0.035	0.015	**100%**
ary2	0.028	0.016	**75%**
ary3	0.347	0.156	**122%**
fibo	2.231	0.078	**2800%**
hash1	0.067	0.063	**6.4%**
hash2	0.060	0.016	**280%**
heapsort	0.210	0.062	**240%**
matrix	0.171	0.063	**170%**
nestedloop	0.367	0.093	**290%**
sieve	0.172	0.078	**120%**
strcat	0.025	0.016	**56%**
Completion time	**7.981**	**0.891**	**796%**

2. http://cvs.php.net/viewvc.cgi/ZendEngine2/bench.php?view=co&revision=1.5&content-type=text%2Fplain
Windows XP SP2, Dell Precision T5400, Intel Xeon X5450 3.0GHz Quad Core, 4GB DDR2

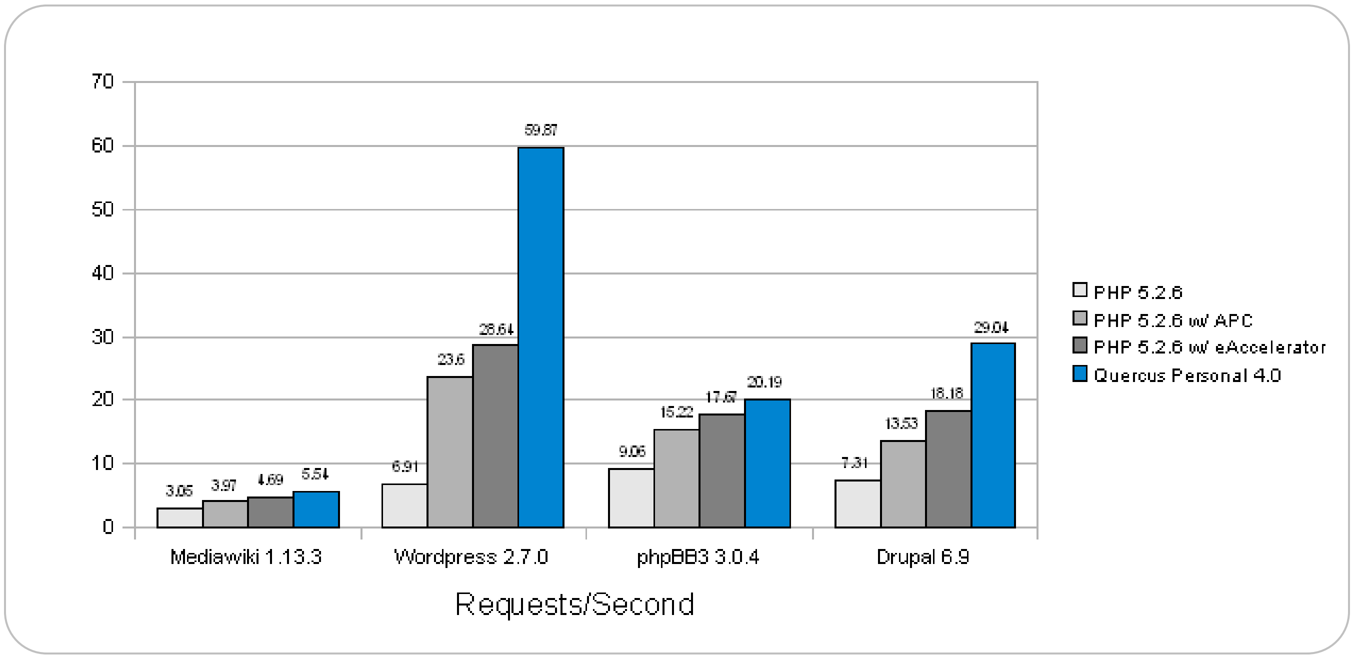
Quercus is faster than PHP in all eighteen tests of the PHP benchmark. Quercus is very fast for math (fibo). Because of heavy loop optimizations, Quercus is about five to fifty times faster for function calls (simplecall, simpleucall, simpleucall, ackerman). Quercus completed the entire benchmark in an impressive 0.891 seconds compared to 7.981 seconds for PHP.

Application Benchmarks

But what about real-world performance? Out of the box, Quercus is several factors faster than plain PHP for several popular PHP applications. This is without any optimizations for PHP, so that wouldn't be a fair comparison.

Generally, users can improve PHP's performance on their machine by enabling either APC or eAccelerator op-code caching. Both store compiled PHP bytecodes into global shared memory and execute them instead of the original PHP source. Even with this optimization enabled, Quercus still retains a large performance edge. Quercus is able to service two times more WordPress requests than PHP with either APC or eAccelerator enabled (see Figure 3)[3]. For WordPress, just using Quercus has the same impact as adding another commodity

PHP/Apache machine!



3. > ab c=1 n=1000 main page
Windows XP SP2, Dell Precision T5400, Intel Xeon X5450 3.0GHz Quad Core, 4GB DDR2
drupal: devel module dummy data (default populate parameters)

Disclaimer

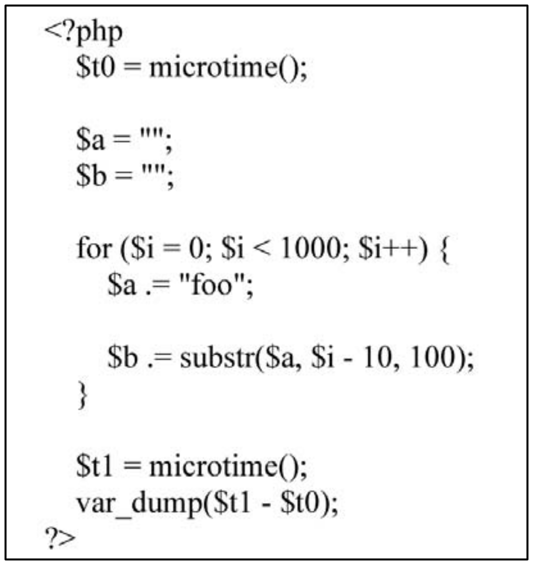
The WordPress result is probably the best-case scenario for Quercus. The benchmark is of a fresh install of WordPress where there are very few database entries. In that situation, Quercus is two times faster than PHP due to various optimizations like the caching of functions, static code optimizations, and a custom regular expression implementation independent of *java.util.regexp*. Quercus' performance advantage will be less as more content are added to WordPress and in turn making the database queries a larger common factor for each request. Therefore, applications that vigorously cache persistent data will reap the greatest benefit from switching to Quercus.

3. CASE STUDY: WORDPRESS AND PROFILING

WordPress on Quercus is already very fast, but application developers can further improve performance by identifying hotspots and remedying them. However, it is not trivial task to profile a running PHP process on Apache. Quercus on the Resin Application Server makes profiling quite a bit easier because it comes with its own profiler that is accessible from the administration panel.

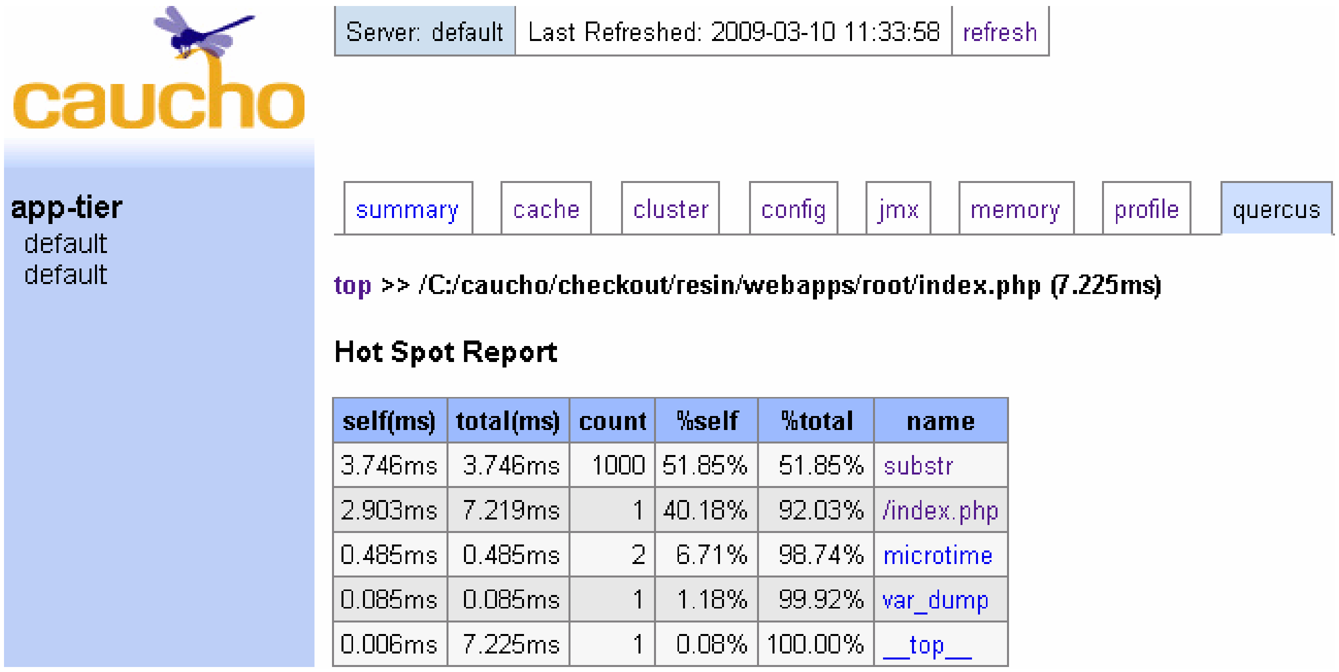
Hot Spot Report

The hot spot feature is unique to Resin's built-in PHP profiler and is incredibly powerful. Take for example the following PHP snippet and resulting profile (see Figure 4).



```
<?php
   $t0 = microtime();

   $a = "";
   $b = "";

   for ($i = 0; $i < 1000; $i++) {
      $a .= "foo";

      $b .= substr($a, $i - 10, 100);
   }

   $t1 = microtime();
   var_dump($t1 - $t0);
?>
```

Sample Code (1000 Interations)



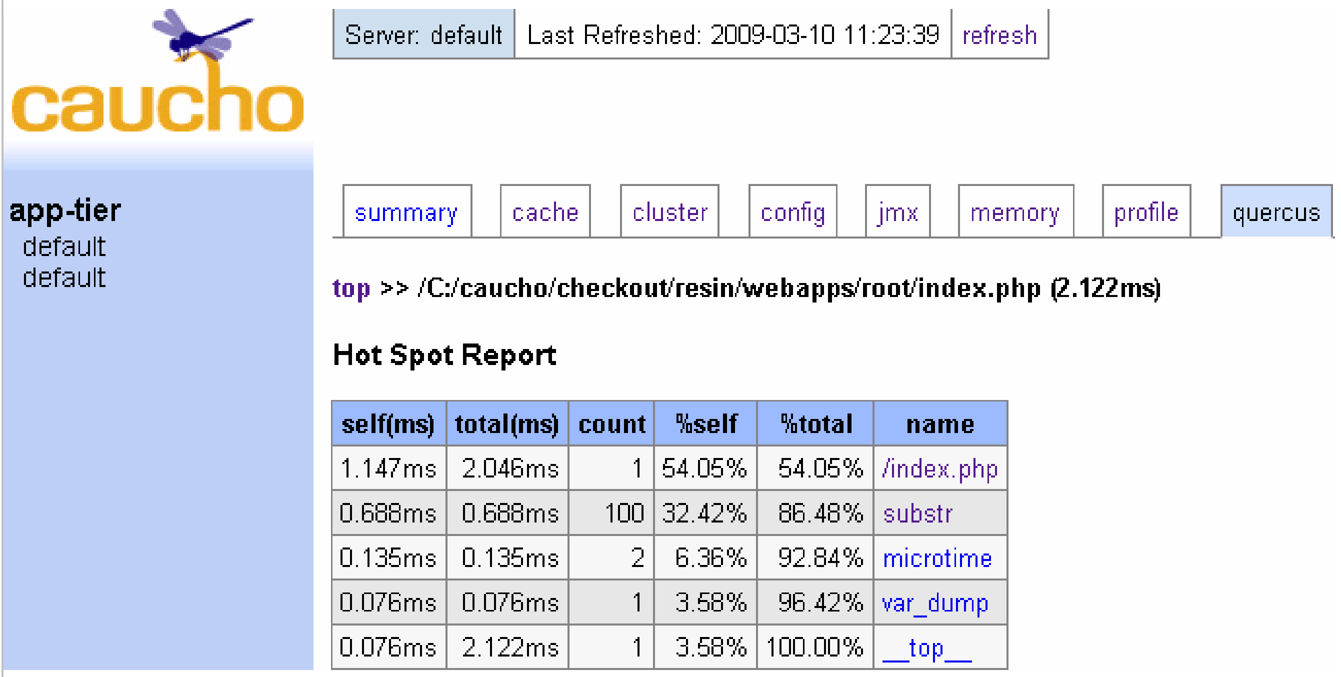
top >> /C:/caucho/checkout/resin/webapps/root/index.php (7.225ms)

Hot Spot Report

self(ms)	total(ms)	count	%self	%total	name
3.746ms	3.746ms	1000	51.85%	51.85%	substr
2.903ms	7.219ms	1	40.18%	92.03%	/index.php
0.485ms	0.485ms	2	6.71%	98.74%	microtime
0.085ms	0.085ms	1	1.18%	99.92%	var_dump
0.006ms	7.225ms	1	0.08%	100.00%	__top__

Figure 4: Hot Spot Report of 1000 Iterations

The report shows the durations inside the page and function calls and the number of calls to the page/function. *substr()* took the longest, followed by the page itself with its *for* loop, string concatenation, and output to the browser. If we decrease the loop iterations to one hundred, then the page execution time dominates *subtr()* in turn (see Figure 5).



Server: default | Last Refreshed: 2009-03-10 11:23:39 | refresh

top >> /C:/caucho/checkout/resin/webapps/root/index.php (2.122ms)

Hot Spot Report

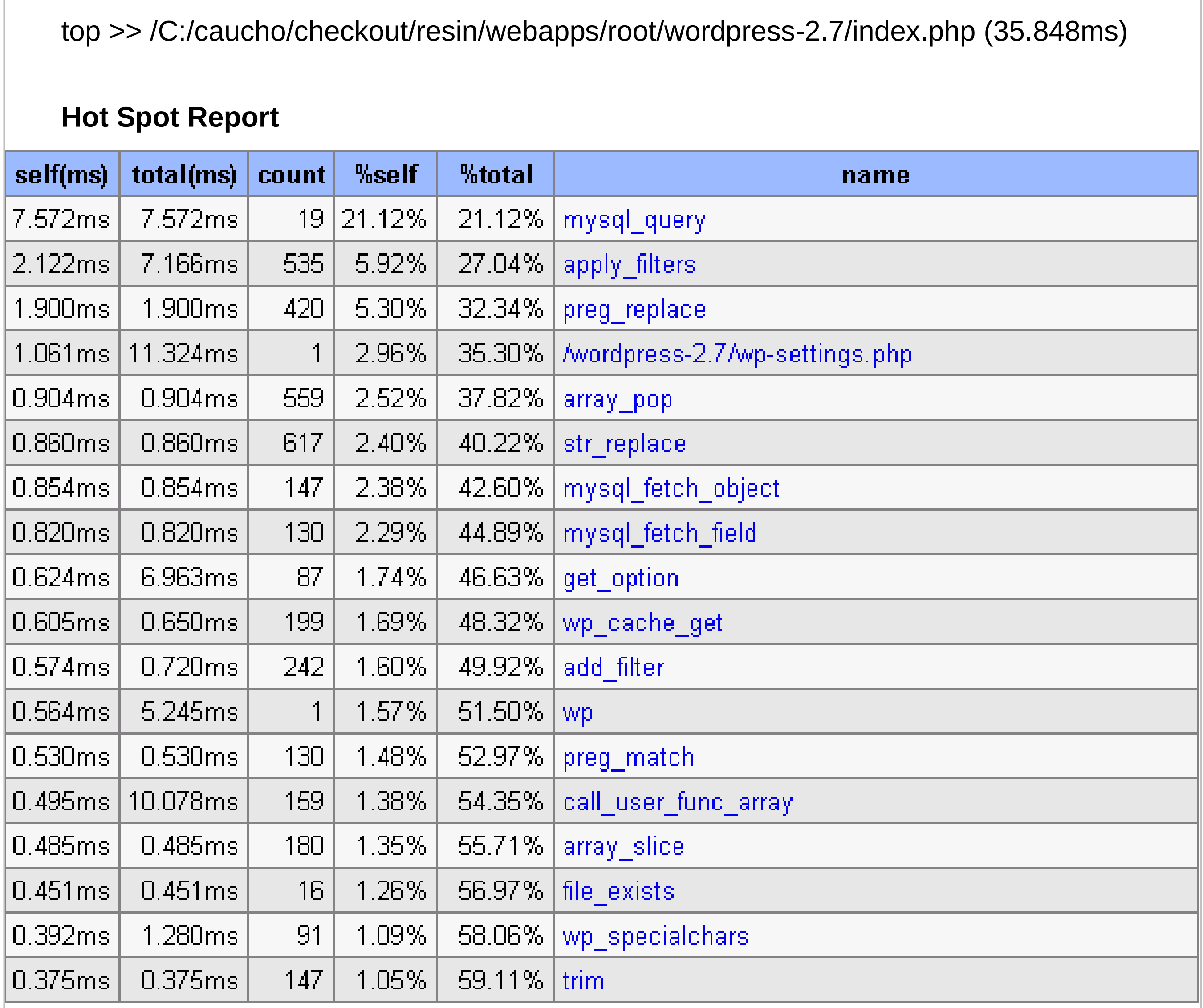
self(ms)	total(ms)	count	%self	%total	name
1.147ms	2.046ms	1	54.05%	54.05%	/index.php
0.688ms	0.688ms	100	32.42%	86.48%	substr
0.135ms	0.135ms	2	6.36%	92.84%	microtime
0.076ms	0.076ms	1	3.58%	96.42%	var_dump
0.076ms	2.122ms	1	3.58%	100.00%	__top__

Figure 5: Hot Spot Report of 100 Iterations

With this powerful tool in hand, we can readily profile any PHP application on live production servers without ever having to modify the application.

WordPress Profile

Running the profiler on WordPress, we find that it is spending 26% of its time in the mysql/jdbc driver with *mysql_query()* and *mysql_fetch_field()* calls (see Figure 6). This large chunk for just the database is typical for PHP applications (but it is lower than average because of WordPress' caching mechanism).



top >> /C:/caucho/checkout/resin/webapps/root/wordpress-2.7/index.php (35.848ms)

Hot Spot Report

self(ms)	total(ms)	count	%self	%total	name
7.572ms	7.572ms	19	21.12%	21.12%	mysql_query
2.122ms	7.166ms	535	5.92%	27.04%	apply_filters
1.900ms	1.900ms	420	5.30%	32.34%	preg_replace
1.061ms	11.324ms	1	2.96%	35.30%	/wordpress-2.7/wp-settings.php
0.904ms	0.904ms	559	2.52%	37.82%	array_pop
0.860ms	0.860ms	617	2.40%	40.22%	str_replace
0.854ms	0.854ms	147	2.38%	42.60%	mysql_fetch_object
0.820ms	0.820ms	130	2.29%	44.89%	mysql_fetch_field
0.624ms	6.963ms	87	1.74%	46.63%	get_option
0.605ms	0.650ms	199	1.69%	48.32%	wp_cache_get
0.574ms	0.720ms	242	1.60%	49.92%	add_filter
0.564ms	5.245ms	1	1.57%	51.50%	wp
0.530ms	0.530ms	130	1.48%	52.97%	preg_match
0.495ms	10.078ms	159	1.38%	54.35%	call_user_func_array
0.485ms	0.485ms	180	1.35%	55.71%	array_slice
0.451ms	0.451ms	16	1.26%	56.97%	file_exists
0.392ms	1.280ms	91	1.09%	58.06%	wp_specialchars
0.375ms	0.375ms	147	1.05%	59.11%	trim

Figure 6: WordPress profile on Quercus

Filters, regular expressions, WordPress initialization, and array and string manipulations are the other top hot spots. With this information in hand, a developer can then optimize away at the application by reducing calls to expensive operations.

4. PROXY CACHE

What Is a Proxy Cache?

A proxy cache is any hardware that serves saved pages to the browser from a cache. Web applications control which pages are saved by the proxy cache and how long they are saved by setting the “Cache-Control” HTTP headers. With a proxy cache, a request rarely would need to go to the server for a dynamic page.

As a result, a cached dynamic page effectively becomes just as fast as a static page. For data that is not updated often but is read many times, proxy-caching will do wonders for performance. MediaWiki fits this profile very well and it is the perfect candidate for a proxy cache. Best of all, MediaWiki natively supports proxy caches.

With a “squid” proxy cache in front of MediaWiki, Wikipedia.org is quick and very responsive under heavy load. Quercus can achieve similar performance because the Resin Application Server has a proxy cache built-in and Quercus will automatically benefit from the “Cache-Control” headers that MediaWiki can set. With a proxy cache, MediaWiki processes requests about 500 times faster on Quercus (see Figure 7). MediaWiki's performance now becomes only limited by just how fast the proxy cache can send pages from its cache to the client.

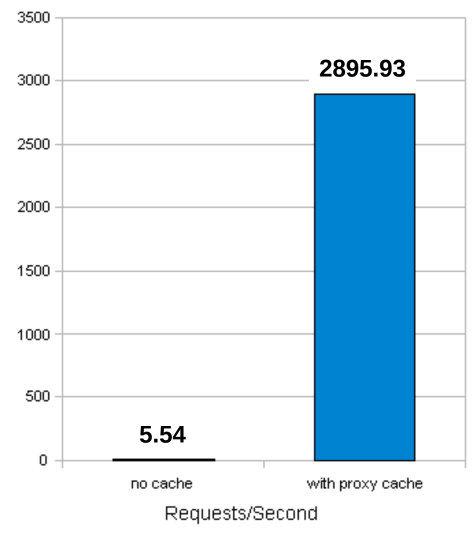


Figure 7: MediaWiki on Resin/Quercus, no cache vs. with proxy cache

4. > ab c=1 n=1000 main page
Windows XP SP2, Dell Precision T5400, Intel Xeon X5450 3.0GHz Quad Core, 4GB DDR2

5. PHP-JAVA INTEGRATION

Increasing specialization is a general method to improve performance. PHP is unbeatable when it comes with running the front end of a site. Meanwhile, Java excels in mission-critical applications and existing PHP business logic can be offloaded to a Java application server like Resin. There the PHP application can benefit from proven enterprise features like transactions, clustering, distributed sessions, distributed caching, etc.

There are several solutions that can help users exploit this PHP-Java integration. PHP-Java Bridge is one of the faster ones. It communicates between PHP and Java via a binary protocol. Because Quercus is 100% Java running in the same JVM, it is no surprise that Quercus is around twenty to forty times faster than the PHP-Java Bridge (see Table B)[5].

Table B: PHP-Java integration benchmark

	PHP-Java Bridge	Quercus	Result

`$a = new java("java.lang.StringBuilder");`
 	
`for ($i = 0; $i < 1000; $i++) {`
`    $a->append("foo");`
`}`
	16.08 requests/sec	357.54 requests/sec	**22 times faster**

`$a = new java("java.lang.StringBuilder");`
 	
`for ($i = 0; $i < 1000; $i++) {`
`    $a->append("foo");`
`    $b = substr($a, $i – 10, 10);`
`}`
	2.94 requests/sec	128.00 requests/sec	**44 times faster**

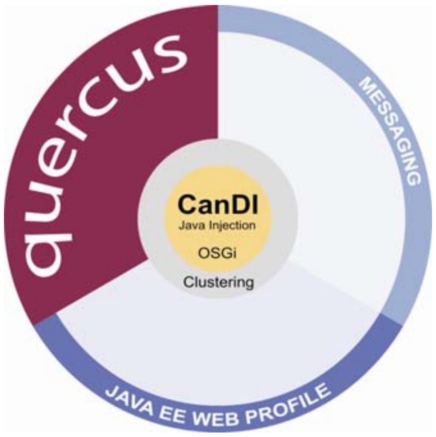
5. > ab c=1 n=1000
Windows XP SP2, Dell Precision T5400, Intel Xeon X5450 3.0GHz Quad Core, 4GB DDR2

With this kind of performance, Quercus makes it feasible for developers to re-architect a pure PHP application into a hybrid PHP-Java application, or even the other way around from a pure Java application to a hybrid Java-PHP application. PHP can then take advantage of Java technologies like distributed caches (i.e. ehcache) to propel applications to performance which are unheard of for a pure PHP solution.

6. SUMMARY

By using the Java architecture to its fullest potential, Quercus solves some of PHP's performance bottlenecks. With its custom regular expression module, global caching, and static code optimizations for PHP, Quercus is super fast in both synthetic benchmarks and in real applications. As a result, Quercus is able to achieve twice the performance of PHP for WordPress.

Quercus introduces connection pooling and PHP-Java integration to the PHP fold. This permits PHP applications to benefit from proven Java technology. Java developers can incorporate the strengths of PHP into their applications without a prohibitive performance penalty as was the case previously. Whereas the relationship was a tumultuous one at best before, Quercus now allows PHP and Java to benefit generously from each other in a new symbiotic coexistence.



About Caucho Technology

Caucho Technology is an engineering company devoted to reliable open source and high performance Java-PHP solutions. Caucho is a Sun Microsystems licensee whose products include Resin application server, Hessian web services and Quercus Java-PHP solutions. Caucho Technology was founded in 1998 and is based in La Jolla, California. For more information on Caucho Technology, please visit www.caucho.com.