



PHP On Steroids: High Performance PHP

**“It's code, Jim, but not
as we know it”**

Jonathan Oxer

August 5th, 2004
Open Source Developers Conference
Melbourne, Australia

When To Optimise

Jon's 3 Laws Of Robotics Optimisation

1. Don't pre-emptively optimise - only do it when you need to in order to reach defined performance goals. Optimised code is (often) obfuscated code.
2. Don't guess which sections of your code need to be optimised. Measure with profiling tools such as APD or Xdebug, and run server benchmark tools **with real data** to give you hard figures.
3. Optimisation is a game of diminishing returns: go for the low-hanging fruit and keep going until it's “fast enough”.

First Things First: Numbers Count

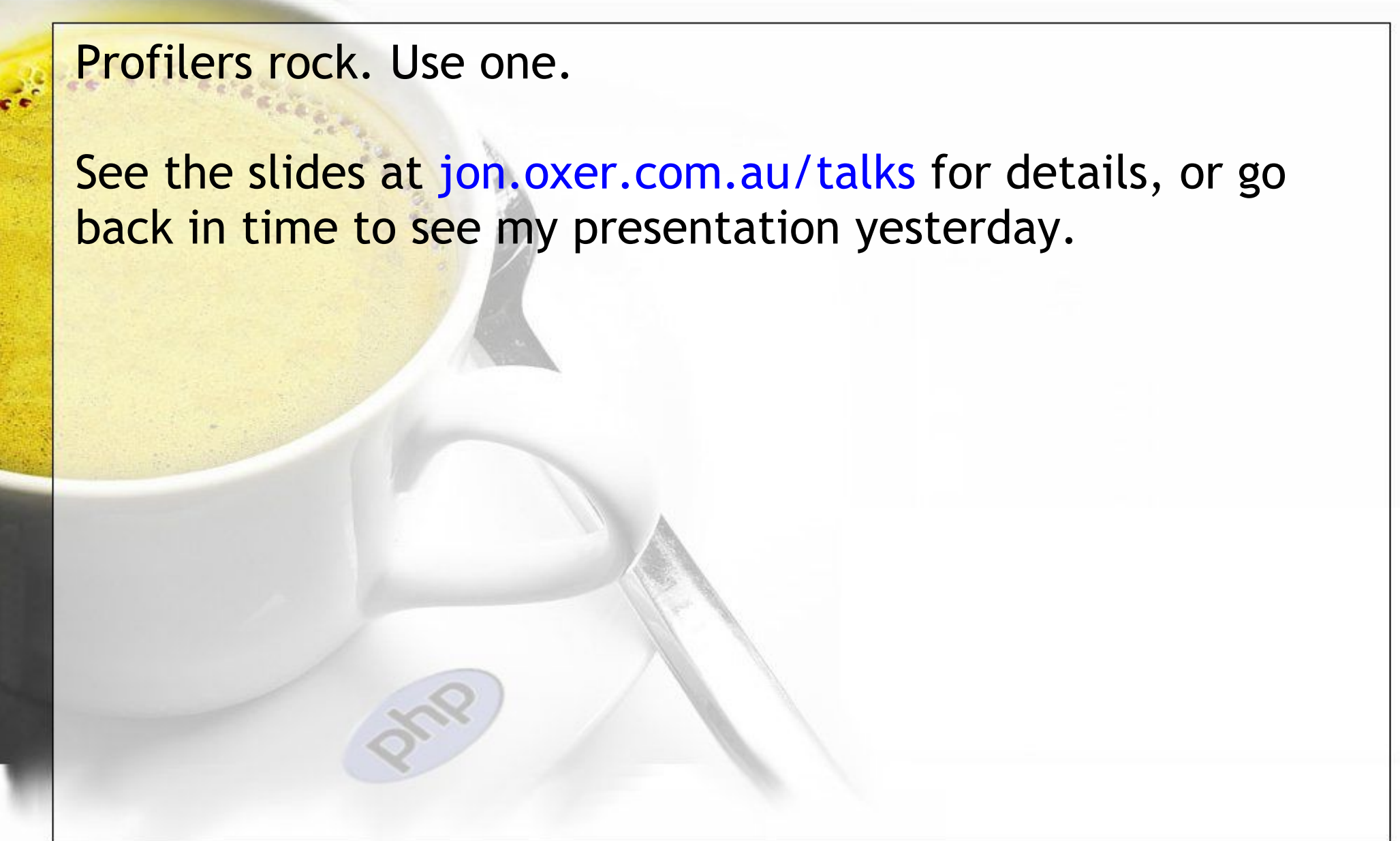
Don't guess, measure. Otherwise you're working blind. Performance bottlenecks are almost never where you *think* they are.

- Install and learn how to use PHP profiling tools such as APD and Xdebug to take a peek inside the code execution process.
- Use server benchmarking tools like Apache-bench (ab) to see how fast it really goes.
- Use system management tools like top and vmstat to see how well your server copes with load and find system bottlenecks.

Performance Profiling

Profilers rock. Use one.

See the slides at jon.oxer.com.au/talks for details, or go back in time to see my presentation yesterday.



Apache Benchmarking

'apt-get install apache-utils' to get ab, Apache-benchmark.

Don't run it on the server though, or it will use so many resources itself that you won't get real results. It also takes the network out of the equation, which is often one of the most important factors in terms of overall performance.

System Analysis

'top' is a common management utility to see how much CPU time and memory is being used by different processes.

'vmstat' is an incredibly useful tool to see where system resources are being used. It will help show you whether your processes are IO-bound, CPU-bound or memory-bound for example.

Where To Start?

Making your PHP run faster involves a lot of factors that aren't directly code related. Don't just focus on code at the expense of the big picture!

There are several general areas to examine:

- Runtime environment
- Associated systems
- Application structure
- Data structure
- Application implementation

Tuning Apache

Apache 1.3 or 2.x? Pre-fork or threaded? Choices, choices!

Apache 2.x is faster in threaded mode, but that's very dangerous and incompatible with many PHP extensions. If you want to see Rasmus start foaming at the mouth, just ask him about thread-safety.

When running Apache2 in pre-fork mode to make it safer with PHP it's about 20% slower than Apache1.3.

So no reason to go to Apache 2, better to use 1.3 - for now.

There are some default settings you can change to help it scale.

Tuning Apache

MaxClients (256)

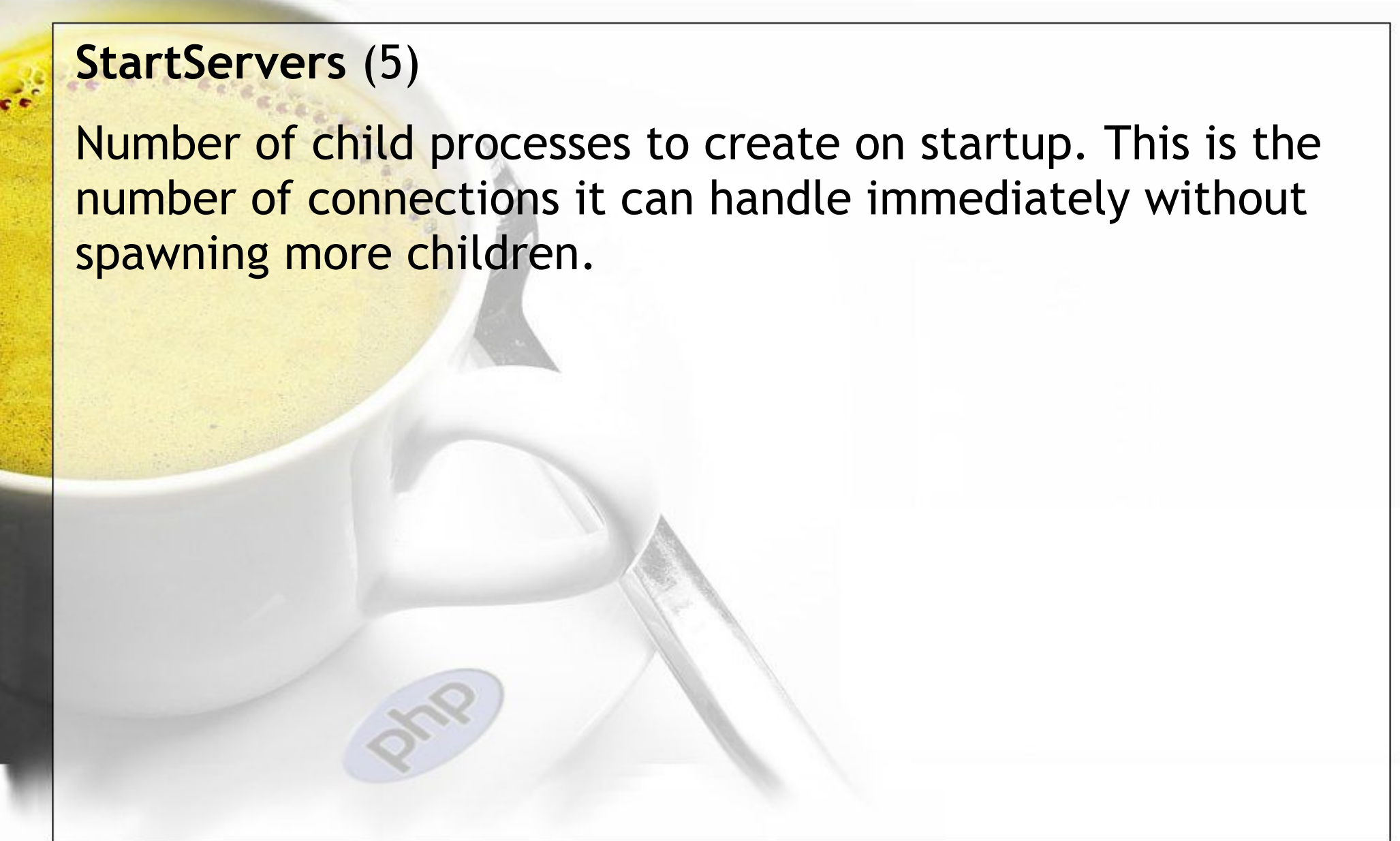
Maximum number of child processes to create. The default means up to 256 requests can be handled simultaneously. Any additional connection requests are queued.

The default should be OK here unless you're handling a large number of long-running connections.

Tuning Apache

StartServers (5)

Number of child processes to create on startup. This is the number of connections it can handle immediately without spawning more children.



Tuning Apache

MinSpareServers (5)

Number of idle child processes to keep hanging around. If the number of idle children (no jokes please!) falls to less than this figure, Apache starts 1 new child. Then after another second it creates 2 more, then 4 after another second, etc until it's creating 32 per second or the MinSpareServers value is reached.

With the default setting there can be some lag as children are created if a large jump in requests happens: best to increase this value to 32. Spawning more children is a slow process (definitely no jokes about that one!).

Tuning Apache

MaxSpareServers (10)

If more than this number of child processes are left hanging around they'll be killed off.

For a high traffic server it's best to increase this value to about 64.

Tuning Apache

MaxRequestsPerChild (0)

Limits the number of requests each child can handle before terminating. The default (0) means never terminate. Setting this to a value like 100 can be useful if you have memory leak problems.

Tuning Apache

KeepAlive (On)

With HTTP 1.0 every request to the server required a whole new connection. HTTP 1.1 adds a 'keep-alive' header to the spec, allowing a browser to reuse the same socket connection to retrieve multiple files.

Having this on (the default) is a good idea if your server handles multiple requests from each client. However, if your server only handles one request per client (such as when running a dedicated banner ad server) turn this off for a dramatic increase in connection rate capacity.

Tuning Apache

KeepAliveTimeout (15)

Number of seconds to leave each socket connection active. If this is set to a long number (preferably longer than the time users spend viewing pages) then each connection will reuse a socket and start faster. But that ties up Apache children which can't serve other users in the meantime, so it's no good on sites with a large number of users.

For sites with lots of users and low network latency set this low, perhaps 5 seconds.

Tuning Apache

HostnameLookups (off)

Determines whether Apache does a reverse DNS lookup on client addresses so it can write hostnames in the logfile. A lot of people turn this on so they can have country reports etc in their traffic reports.

Don't. It'll make Apache slower than a sick 3 legged dog.

If you really need country stats in your traffic reports do the conversion in post-processing.

Tuning Apache

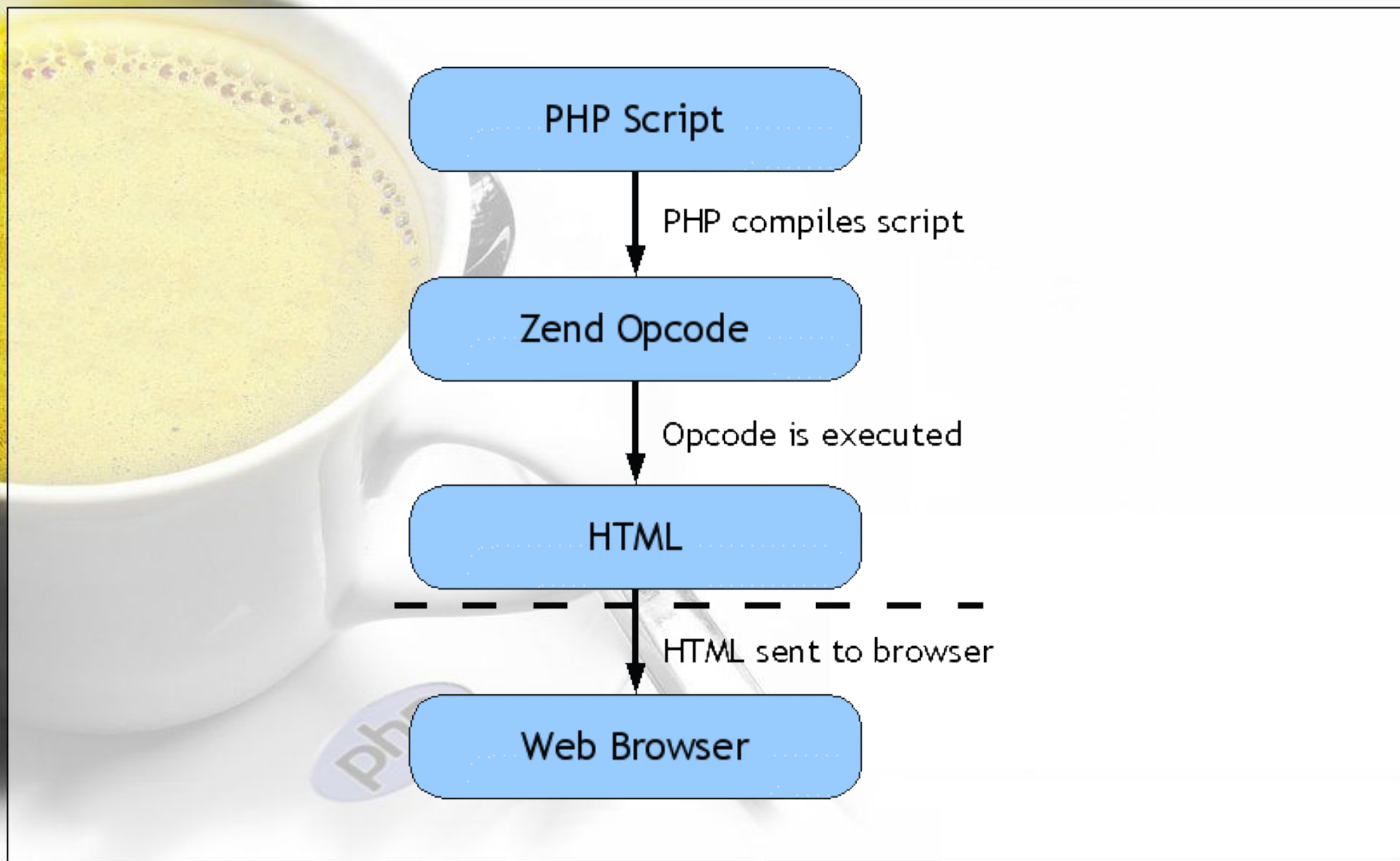
If you don't need to use `.htaccess`, turn it off. It'll save a heap of checks every time a file is requested.

```
<Directory />  
  AllowOverride none  
</Directory>
```

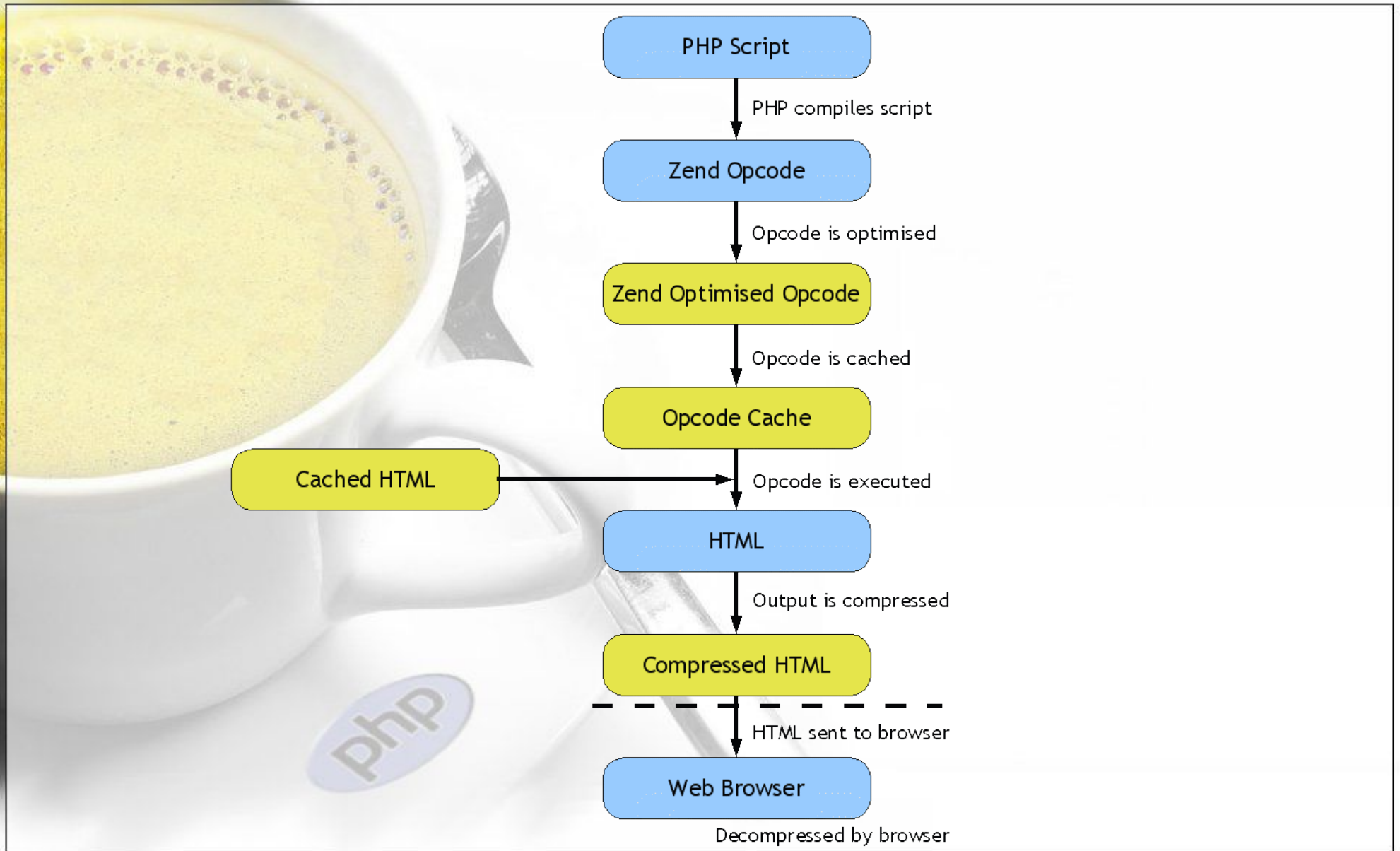
Likewise symlink checks: if you don't care about symlink security, turn on `FollowSymLinks` and turn off `SymLinksIfOwnerMatch` to save a heap of `'lstat()'` system calls:

```
Options FollowSymLinks  
#Options SymLinksIfOwnerMatch
```

Know Your Zend Engine



The Zend Engine



Zend Optimizer

The Zend Optimizer is a 'peephole' optimizer that hooks into the Zend Engine to perform code optimisation during the first processing pass. The result is opcodes which are more efficient than those normally generated.

Performance improvements of 0-50% can be expected on most code, but on some code it can actually slow you down. Most benefit is when you have lots of tight loops.

Beware: Zend Optimizer prevents you using any third-party opcode caches.

Opcode Caching

An opcode cache grabs a copy of the output of the first processing pass by the Zend Engine and stores it for later reuse. There are quite a few options:

- Zend Accelerator
www.zend.com
- Turck-mmcache
turck-mmcache.sourceforge.net
- APC (Alternative PHP Cache)
apc.communityconnect.com
- ionCube PHP Accelerator (wave to Flame)
www.php-accelerator.co.uk
- AfterBurner Cache
afterburner.bware.it

HTML Caching

Why regenerate pages if you don't have to? Install an HTML caching system to avoid executing PHP altogether.

Some templating systems such as Smarty incorporate a caching mechanism.

PEAR Cache is a set of caching classes that help you cache HTML and images. To cache HTML you use the output buffering class to accumulate the output from your script and store it for later re-use.

Cache_Lite is a newer caching class than PEAR Cache and it's a bit faster.

Output Compression

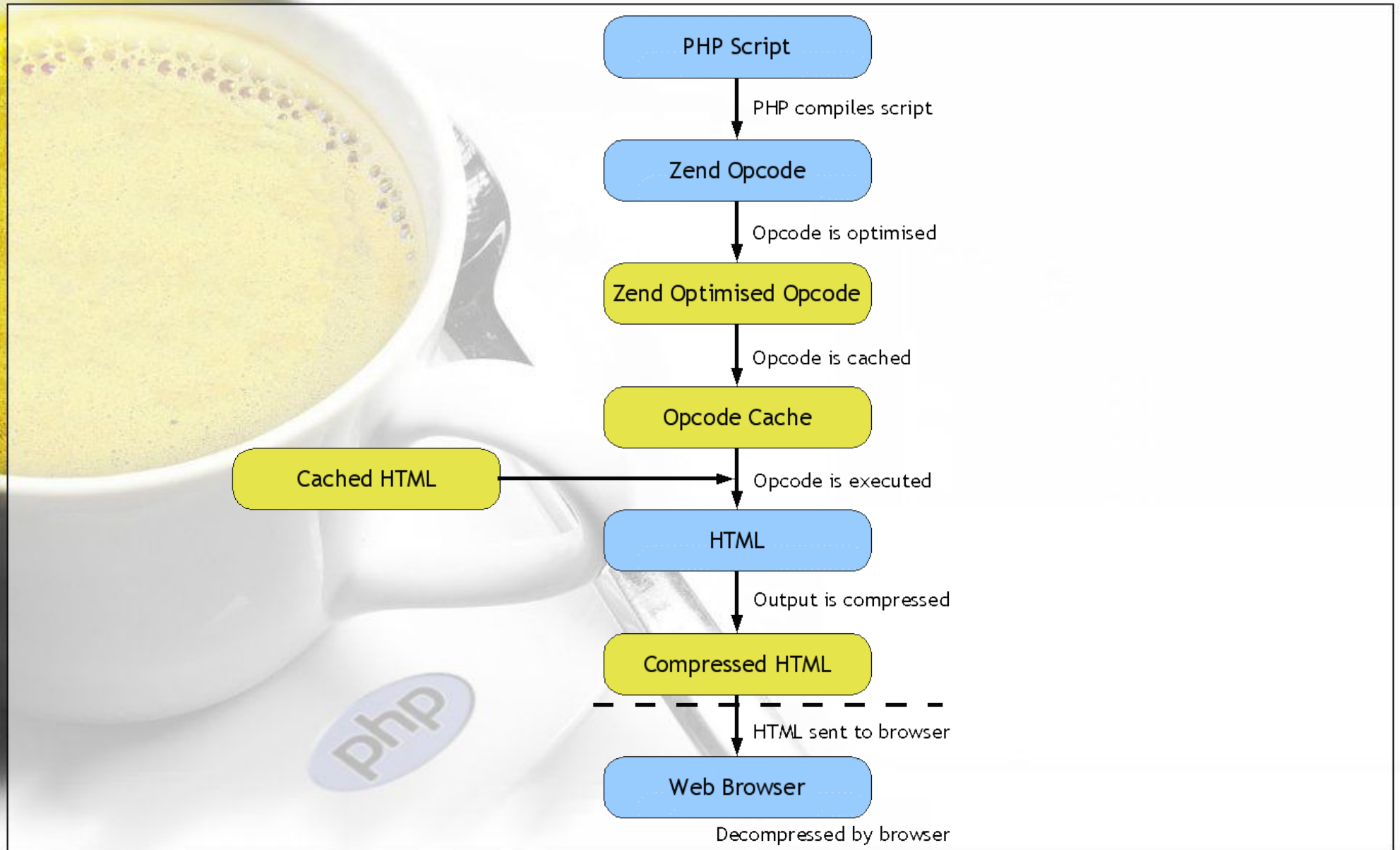
If the net is the slowest link in the chain and we've got CPU cycles to spare, let's try to squeeze more through by compressing the HTML we send down the wire.

Create a compressed output buffer using `ob_gzhandler` or, better still, activate the `zlib.output_compression` module in your `php.ini` file if you run your own server.

There are also a bunch of third-party classes out there which do a very similar thing.

Output compression can often give you a 1000% increase in throughput, provided you've got the CPU grunt to spare.

Know Your Zend Engine



Don't Blow Your Output Buffers

The TCP/IP stack in the Linux kernel has a buffer to hold data from applications that want to send it out over the network. If you're outputting more data than the buffer can hold it makes the kernel do it in several passes, which slows things down.

Load the biggest page on your site (in terms of HTML generated) and see how many bytes it takes up.

Then set the kernel buffer size to a slightly bigger value like this:

```
echo "65536" > /proc/sys/net/core/wmem_max
```

Do it in the Apache startup script so you don't forget. Oh, and set `SendBufferSize` in `httpd.conf` while you're at it.

Respect Bandwidth

Network bandwidth is probably the biggest bottleneck in any web app.

Think about it: if you send out web pages which average 40kB, and you have a 10Mbit net connection, how many pages can you send per second? Only about 25.

That's pathetic.

Look at ways to reduce your output volume: prune your HTML, use separate CSS files instead of inlining your CSS, don't send Javascript functions in your header unless that page needs it, optimize your images, compress your output.

And just send less stuff!

Watch Your Filesystem

Memory is fast. Disk is sloooooooooooooooooooooooooooooow.

In case you missed it the first time, I'll repeat it: disk is slow.

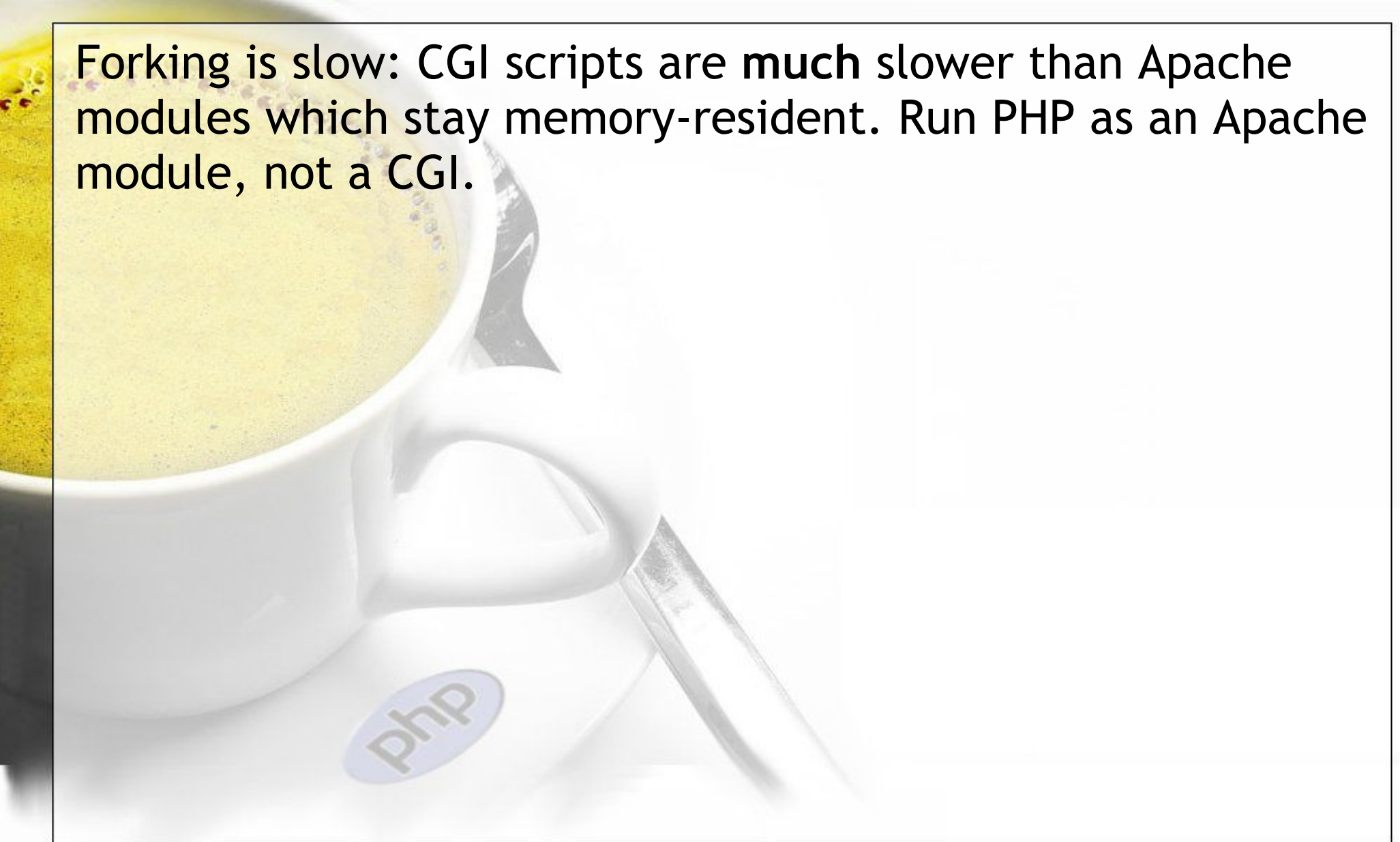
Put plenty of RAM in your server to prevent hits to virtual memory and allow output caching. RAM is dirt cheap. Buy plenty.

Symlinks are slow. Avoid them.

Default Linux installations set up the disk controllers using a lowest common denominator approach for maximum compatibility. Don't settle for that: tune your disk with hdparm.

Prune Your Environmental Processes

Forking is slow: CGI scripts are **much** slower than Apache modules which stay memory-resident. Run PHP as an Apache module, not a CGI.



Prune Your Internal Processes

Don't invoke system calls unless you really, really need them. Use internal PHP functions where available. Both of these do the same thing:

```
<?php
    `mkdir /tmp/myworkingdir1`;
    mkdir("/tmp/myworkingdir2");
?>
```

but the second is just a smidge faster.

OK, I lied. It's not just a smidge faster. It's 20+ times faster.

Prune Your External Processes

We don't need no stinkin' X. And we certainly don't need no stinkin' 3D screensaver that uses up 100% CPU time.

Get rid of anything your server doesn't **really** need. You should be able to prune these and more:

- telnetd
- inetd
- atd
- lpd
- samba
- portmap
- xfs / xinit / X

Databases: Friend Or Foe?

Databases are an essential part of almost all complex web apps. They're very powerful, but they're also a big potential time-sink. A single badly-structured SELECT can do more damage to app performance than almost any other factor.

- Make sure your data is structured efficiently
- Simplify queries and only query when you have to
- Offload the database to another box
- Scale up the database using replication

Data Structure Matters

Don't try to mix different kinds of data together: use a schema that structures it neatly with defined relationships between each table. In DBA-techie-speak this is part of a process known as “data normalisation”, which involves 4 levels of practical structural optimisation based on rules which have been devised by set-theory mathematicians, plus a 5th level that only a theoretical mathematician could love.

It sounds scary but it's not, and it'll help you impress your boss and pick up girls and save the whales. And stuff. So hold on tight, here we go...

Zero'th Normal Form: Totally Abnormal

Not really a level of data normalisation at all, this is the baseline: a table that holds everything. The process of normalisation is a matter of eliminating redundancy and inconsistent dependencies in and between your tables.

users				
Name	Company	Address	Url1	Url2
Billy Bob	Big Co	1 Main Road	google.com	msn.com
Amelia Ann	Little Store	23 Lilac Street	yahoo.com.au	slashdot.org

First Normal Form

Achieving the first normal form involves applying some rules:

- *Eliminate repeating groups in individual tables*
- *Create a separate table for each set of related data*
- *Identify each set of related data with a primary key*

users				
userid	name	company	address	url
1	Billy Bob	Big Co	1 Main Road	google.com
1	Billy Bob	Big Co	1 Main Road	msn.com
2	Amelia Ann	Little Store	23 Lilac Street	google.com
2	Amelia Ann	Little Store	23 Lilac Street	slashdot.org

Second Normal Form

Rules for the second normal form:

- *Create separate tables for sets of values that apply to multiple records*
- *Relate these tables with a foreign key*

users			
userid	name	company	address
1	Billy Bob	Big Co	1 Main Road
2	Amelia Ann	Little Store	23 Lilac Street

urls		
urlid	reluserid	url
1	1	google.com
2	1	msn.com
3	2	google.com
4	2	slashdot.org

Third Normal Form

The third normal form has just one rule, but if you get this far your data will be in pretty good shape:

- *Eliminate fields that do not depend on the key*

companies		
compid	company	address
1	Big Co	1 Main Road
2	Little Store	23 Lilac Street

users		
userid	name	relcompid
1	Billy Bob	1
2	Amelia Ann	2

urls		
urlid	reluserid	url
1	1	google.com
2	1	msn.com
3	2	google.com
4	2	slashdot.org

Fourth Normal Form

The fourth normal form is concerned with the most complex inter-data relationships: many-to-many joins.

- In a many-to-many relationship, independent entities can not be stored in the same table*

companies		
compid	company	address
1	Big Co	1 Main Road
2	Little Store	23 Lilac Street

users		
userid	name	relcompid
1	Billy Bob	1
2	Amelia Ann	2

urls_users		
relid	reluserid	relurlid
1	1	1
2	2	1
3	1	2
4	2	3

urls	
urlid	url
1	google.com
2	msn.com
3	slashdot.org

Fifth Normal Form: Ultimate Normality?

The fifth normal form is way too theoretical for any of us to really care:

- *The original table must be reconstructed from the tables into which it has been broken down*

Basically this is a kind of sanity-check to make sure you didn't stuff up while normalising your tables, and that no extraneous columns etc have been added.

Simplify Queries

Let's put it in caveman-speak: Simpler queries good. Fewer queries good. Fewer, simpler queries very good. Ug ug.

All the usual stuff: don't join across non-indexed columns, turn on indexing for columns you query a lot but only if the index maintenance won't kill you, specify SELECT fields, etc.

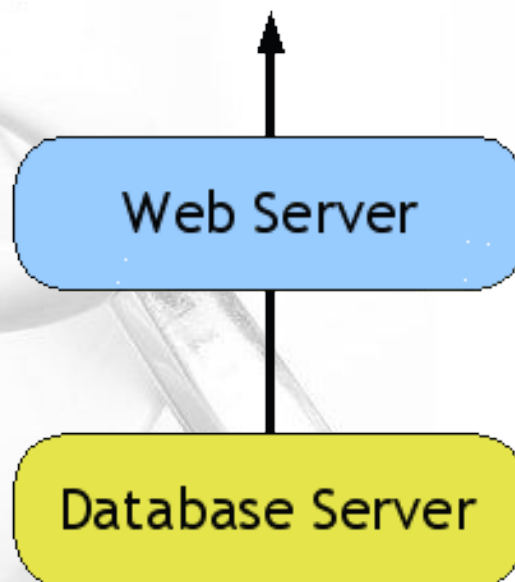
Plus more interesting stuff: don't run sub-queries, it's usually much faster to build a temporary table and reference it in a second query. And learn some SQL other than INSERT, SELECT, UPDATE and DELETE. There are a world of time-saving operations like LOAD DATA INFILE, which lets you bulk-import data in one query rather than doing 50 million INSERTs.

If necessary, pre-process data to suit your schema.

Offload The Database

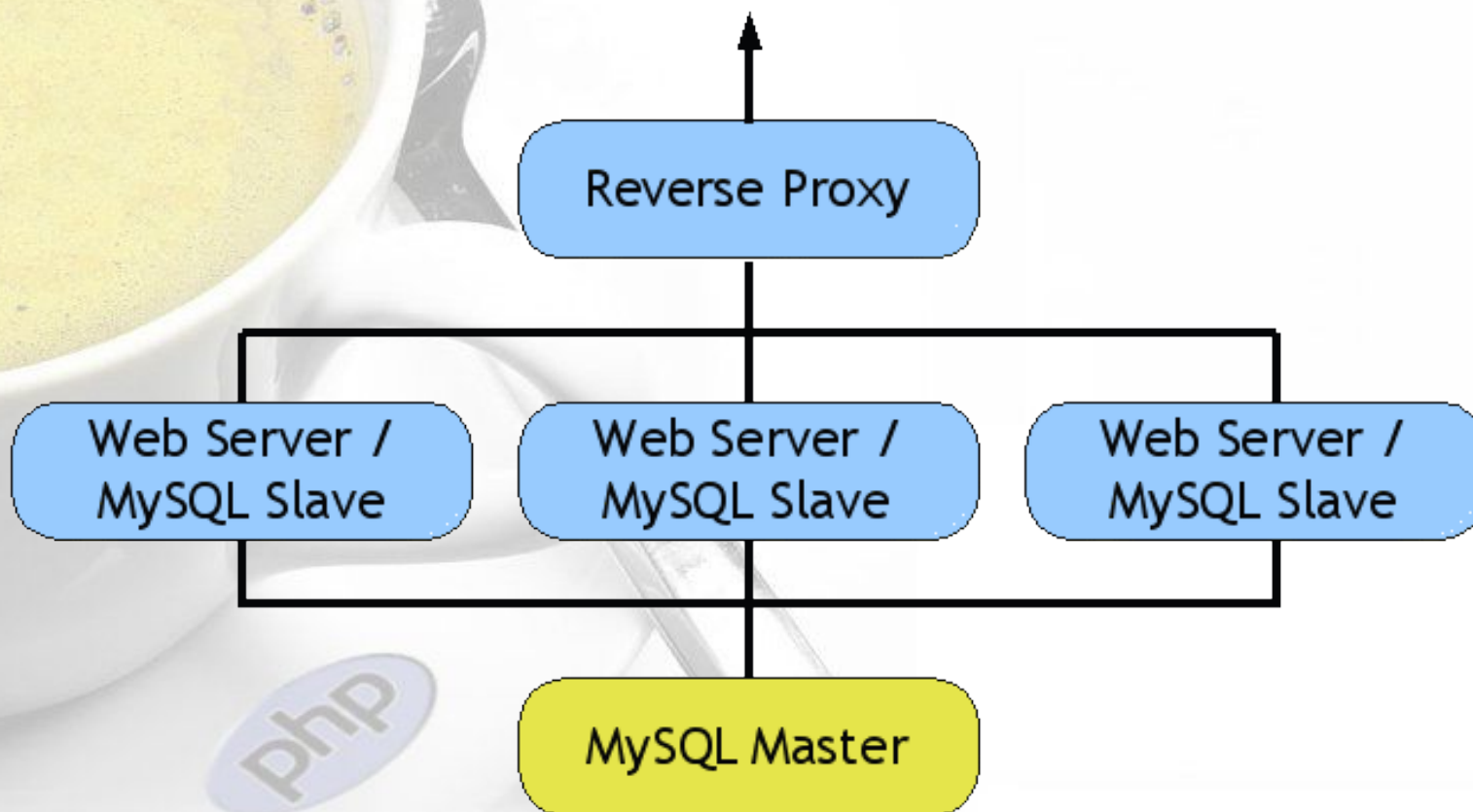
Having the database on your web server can be convenient, but it makes your server do two very high-pressure jobs at once.

First step to scaling up your deployment platform is to put the database on a dedicated server behind your web server.



Imagine A Beowulf Cluster Of...

If one web server won't cut the mustard, use two. Or three. Or seventeen. Use Squid in reverse proxy mode to act as a load balancer, but watch out for session management issues.



Loops

From a structural standpoint loops are the first place you should look in your code when trying to squeeze out cycles. Any optimisation you can do in a loop is multiplied by the number of iterations.

Especially avoid putting things that don't change inside a loop (jargon-speak: “loop-invariant optimisation”).

Sometimes things sit on the border of a loop and it's not obvious you're actually calling it many times:

```
for ($pos=0; $pos<sizeof($array); $pos++)  
    echo $array[$pos].'<br>';
```

Better to do the sizeof() first.

Libraries And Classes

Libraries can be very useful things, but take care of how you use them: make sure functions are logically grouped. It really sucks to load an extra include and instantiate a 10,000 line class just to call a single method that you could replicate in 5 lines of code.

OOP Subtlety

- Calling object methods is twice as slow as calling a function
- Incrementing a local variable in a method is very fast
- Incrementing a global variable is 2 times slower than a local variable
- Incrementing an object property (eg: `$this->$property++`) is 3 times slower than a local variable
- Incrementing an undefined local variable is 9 times slower than a pre-initialised variable.
- Declaring a global variable without using it is as slow as incrementing a local variable

Don't Reinvent The Wheel

PHP has an *amazing* range of built-in functions for things you wouldn't even dream of doing. Built-in functions are much faster than implementing it yourself. Spend some time on www.php.net reading up on them.

Before implementing something from scratch check whether it's already in PHP as a native function or an extension.

Passing By Reference

Passing by reference means that you don't send a copy of a data structure to a function, you just send a pointer to it.

So instead of doing:

```
$result = myfunkyfunc($myobject, $myarray, $myint);
```

Do this:

```
$result = myfunkyfunc(&$myobject, &$myarray, $myint);
```

Can save a heap of memory as well as CPU cycles, but can trip you up if you're used to passing by value. It's also slower (!) for simple data structures since a reference counter must be incremented.

In PHP5 objects are passed by reference by default. Yay!

Substring Search And Replace

When searching for a string in a string, ranked fastest to slowest:

- `strpos()`
- `preg_match()`
- `ereg()`

Similarly for substring substitutions, fastest to slowest:

- `str_replace()`
- `preg_replace()`
- `ereg_replace()`

String Concatenation

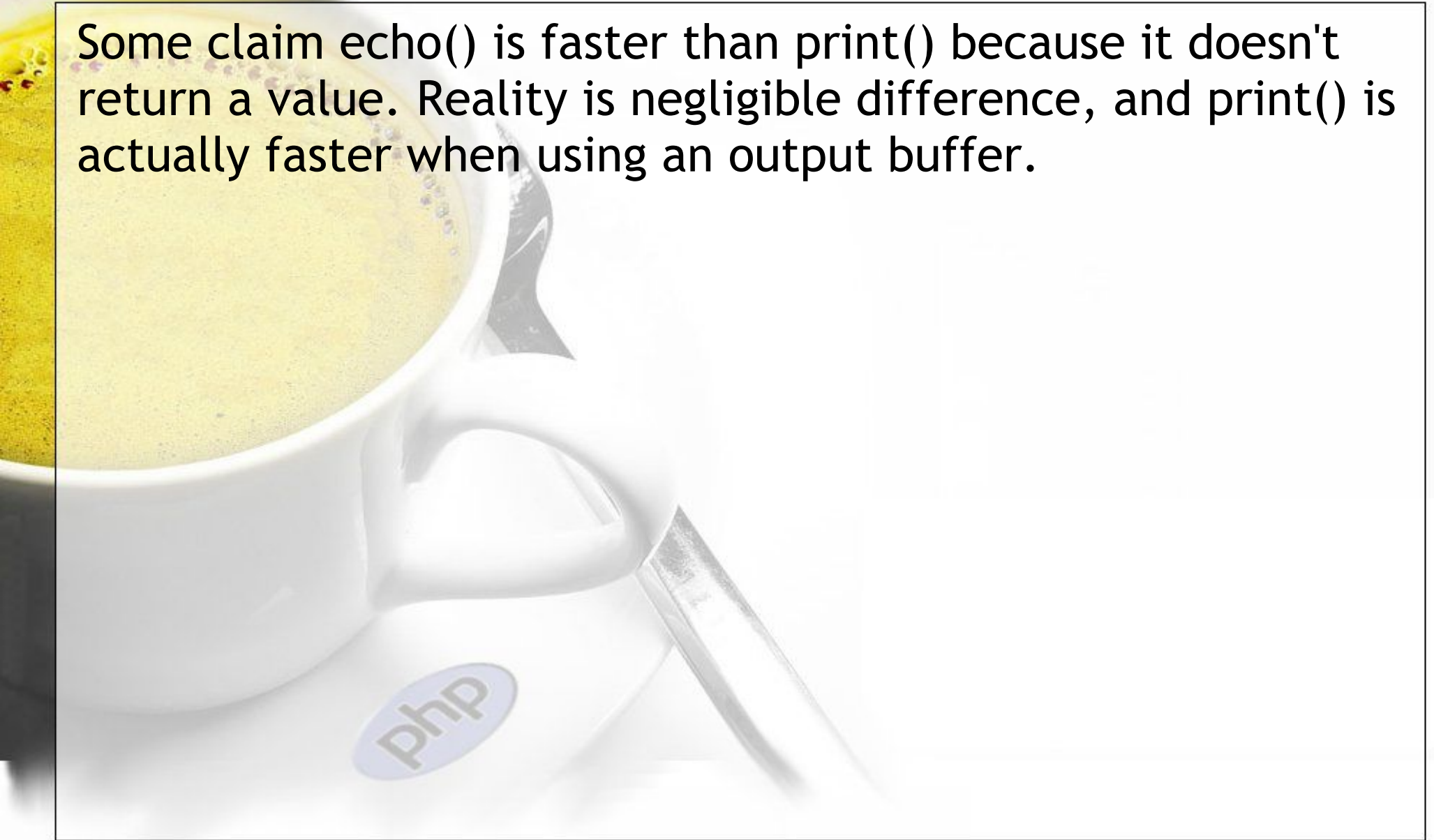
Concatenating strings can be slow: the killer is memory allocation within the Zend Engine, which is done on a per-string basis.

Instead, create an output buffer using `'ob_start()'` and echo into the buffer. Then collect the buffer at the end using `'ob_get_contents()'`. Only worthwhile if you have lots of small strings though.

Output buffers are given a generous 40k allocation on creation, then grow in 10k chunks.

Myth: echo() And print()

Some claim echo() is faster than print() because it doesn't return a value. Reality is negligible difference, and print() is actually faster when using an output buffer.



Myth: Comments Slow Code

PHP used to be interpreted one line at a time and comments slowed things down. Opcode caches totally negate this effect, since comments are stripped anyway.

Don't leave your code uncommented just to gain a bit of speed, it's not worth it.

Myth: Single vs Double Quotes

Single quoted strings are treated as literal strings: double quoted strings are parsed for variable substitution. It used to be the case that this caused a performance hit so it was preferable to assign variables with single quotes when there are no internal variables:

Good: `$mystring = 'What should I say?';`

Bad: `$mystring = "What should I say?";`

The Zend Engine now handles internal variable substitution so efficiently that the effect is immeasurably small - at least I can't measure it, and I've tried fairly hard :-)

Convert PHP To C

Sometimes nothing you do will make your PHP fast enough and the only solution is to re-implement core logic in C.

Writing your own PHP extension allows you to dramatically speed up performance-critical code, but at the cost of harder maintenance and less portability.

The definitive reference for creating PHP extensions is “Building Custom PHP Extensions” by Blake Schwendiman.

More Information



These slides are online at
jon.oxer.com.au/talks