THE HIPHOP VIRTUAL MACHINE

Keith Adams
Facebook
2/6/13
Why PHP?
Facebook’s PHP Engines

• 2004-2009: Zend
  • Switch-on-bytecode C interpreter

• 2009-2012: HipHop
  • Ahead-of-time compiler
  • Intermediate: a C++ program
  • Output: big fat ELF binary
  • Devs use interpreter

• 2013-: HipHop VM
  • Just-in-time compiler
  • Dev/prod unified
Overall, HipHop's throughput over this period improved by the Zend cluster was changed to use the baseline HipHop HipHop after August 2010. This comparison was performed left side of Figure 5, this throughput data is plotted relative cluster achieve throughput 2.5 of requests-per-second (RPS), demonstrated that the HipHop utilized. The results of this experiment, measured in number vide enough traffic to keep the CPUs in both clusters 100% servers in its cluster were underutilized. We also compared two sets received equal traffic. The average CPU time serv-

From “The HipHop Compiler for PHP,” Zhao et al., OOPSLA 2012
HipHop Learnings

• Compiler taught us that *type inference* is the most important optimization available.

• Many types cannot be inferred at compile-time!
Dynamic Types

goldbach_conjecture() ? 3.14159 : “string”

mysql_fetch_row($result)[0]

123.2 / $divisor
Claim: Types are homogenous in practice

- $a / b$ could produce boolean but almost never does
- Database row types rarely change
- Goldbach conjecture can only be true or false
- Etc.
HHVM Vision

• Observe types at runtime before generating code
• Discover types that cannot be inferred
Aside: HipHop Bytecode

- Originally developed to support a fast interpreter
- Executable serialization of PHP

```
lval:$c
+ _______ 
|        |
| +      |
rval:$a     rval:$b
```

PushL 1  Add  SetL 2
Example program

```php
<?
function mymax($a, $b) {
    return $a > $b ? $a : $b;
}
...
```
Type polymorphism

```php
function mymax($a, $b) {
    return $a > $b ? $a : $b;
}
```

mymax(1, 2)                   => 2
mymax(1.3, 3.2) => 3.2        => 3.2
mymax(true, false)            => true
mymax(array(1,2), array(4))   => array(1,2)
mymax(array(1,2), array(2,1)) => array(2,1)
mymax("Felix", "Bob")         => "Felix"
In HHBC

function mymax($a, $b) {
    return $a > $b ? $a : $b;
}

PushL 1
PushL 0
Gt
JmpZ 1f
PushL 0
Jmp 7 2f
1:  PushL 1
2:  RetC
HHVM’s Strategy

- Basic-block at a time
- Type specialization
- Goal: incremental *type discovery*
Tracelets

• Our compilation unit: the tracelet

• Sequence of control-flow-free bytecode instructions, annotated with type information

• We construct tracelets the moment before running them, translate them to machine code, and chain them together in the code cache
Tracelet construction

- `mymax(10, 333);

```
PushL 1
PushL 0
Gt
JmpZ 1f
PushL 0
Jmp 7 2f
1: PushL 1
2: RetC
```
Tracelet construction: trimming

- `mymax(10, 333);`

```plaintext
PushL 1
PushL 0
Gt
JmpZ 1f - Break at branch
PushL 0
Jmp 7 2f
1: PushL 1
2: RetC
```
Tracelet construction

- mymax(10, 333);

```
PushL 1
PushL 0
Gt
JmpZ X
```
Tracelet construction

- `mymax(10, 333);

  PushL 1 – Relies on locals 0 and 1
  PushL 0
  Gt
  JmpZ X
Tracelet construction: guards

- `mymax(10, 333);

Local0 :: Int
Local1 :: Int
PushL 1
PushL 0
Gt
JmpZ X
Tracelet construction: data flow

• $\text{mymax}(10, 333)$;

\begin{align*}
\text{Local0} &:: \text{Int} \\
\text{Local1} &:: \text{Int} \\
\text{PushL} &\ 1 \\
\text{PushL} &\ 0 \\
\text{Gt} &\ (\text{Int}, \text{Int}) \rightarrow \text{Bool} \\
\text{JmpZ} &\ X (\text{Bool}) \rightarrow ()
\end{align*}
Tracelet construction: machine code

- `mymax(10, 333);

```assembly
Local0 :: Int
Local1 :: Int
PushL 1
PushL 0
Gt
JmpZ X

cmpl $0x3,-0x4(%rbp)
jne <retranslate>
cmpl $0x3,-0x14(%rbp)
jne <retranslate>
mov -0x20(%rbp),%rax
mov -0x10(%rbp),%r13
mov %r13,%rcx
cmp %rax,%rcx
jle <translateSuccessor0>
jmpq <translateSuccessor1>
```
Tracelet construction

- `mymax(10, 333);`

    ```
    PushL 1
    PushL 0
    Gt
    JmpZ 1f — Taken!
    PushL 0
    Jmp 7 2f — Skipped
    ```

1: PushL 1
2: RetC
Logical view of code cache

A

$a :: \text{Int},$
$b :: \text{Int}$
$a > b ?$

C

$a :: \text{Int},$
$b :: \text{Int}$
return $b$

A: PushL 1
PushL 0
Gt
JmpZ 1f

B: PushL 0
Jmp 7 2f

C: 1: PushL 1
2: RetC

Retranslate A
Retranslate B
Retranslate C

Program Flow
Guard Flow
Call mymax(“a”, “z”)
Call mymax(“z”, “a”)
Risk: Code explosion

- $N$ inputs, each takes on $t$ types
  - will yield $t^N$ separate translations!
- Solution: truncate tracelet chain at 12 items
- Fall back to interpreter.
- Applies to 0.0066% of chains
Function types

- PHP methods and functions return untyped data
- Late-bound
- Early approach: using return values *breaks tracelets*

```php
function hotFunc() {
    ... foreach ($obj in $objs) {
        $s = $a[$obj->getName()]; // what type?
    }
}
```
class C {
    ...
    function getName() { return $this->name; }
}

class D {
    ...
    function getName() { return "D"; }
}

function hotFunc() {
    ... foreach ($obj in $objs) {
        $s = $a[$obj->getName()]; // what type?
    }
Type profiling

• Special case of value profiling\[1\]
• Typical value-profiling records values encountered per-call site

• Problems
  • Too much information
  • Requires long calibrations

Fun trick: profile names

• Run first $N$ requests in interpreter
• Remember *method name* -> *return type* info
• System learns facts like “methods named getName() tend to return strings”
  • Works for callsites, classes, methods, etc. we haven’t seen in warmup
  • Exploits the ways that humans naturally write code
• Our codebase has millions of call sites, but only 13K unique method names
• Accuracy ~99%
## Example name predictions

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>Null</td>
</tr>
<tr>
<td>getTimer</td>
<td>Int</td>
</tr>
<tr>
<td>get_loggedin_user</td>
<td>Int</td>
</tr>
<tr>
<td>is_fb_employee</td>
<td>Bool</td>
</tr>
<tr>
<td>array_pull</td>
<td>Array</td>
</tr>
<tr>
<td>IPAddress</td>
<td>Object</td>
</tr>
<tr>
<td>ends_with</td>
<td>Bool</td>
</tr>
<tr>
<td>vqsprintf</td>
<td>String</td>
</tr>
<tr>
<td>html2txt</td>
<td>String</td>
</tr>
</tbody>
</table>
Perf
HHVM vs. Zend
Perf: Wordpress

Wordpress RPS

- Zend
- HHVM Interp
- HPHPc
- HHVM Jit
- HHVM Repo Jit

RPS
Facebook CPU idle (1/27/2013)
Road ahead

- Low-hanging fruit
- Memory management
- Broadened compilation scope
- Improved compilation quality
- Science fiction: compilation as big-data
Conclusions

• HHVM can run your PHP
  • very fast
  • in both prod and dev
• Lots of cool work left
Team HHVM
Thanks!

- https://github.com/facebook/hiphop-php/
- Our group blog: http://www.hiphop-php.com/wp/
HHBC is a platform, not an IR

- Bytecode can be serialized
- Runs without source
- Natively supports PHP datatypes
- Other front-ends, languages could produce HHBC
Ahead: Compilation at scale

- Facebook: many thousands of machines running the same app

- Almost *any* profiling/compilation overhead can be amortized

- Challenge: publish local compilation results
Ahead: HackIR

- SSA-based IR
- Supports classic compiler optimization suite, multiple machine targets
- Recently checked in
Ahead: Larger compilation scope

• Stitch together a method’s tracelets

• HHIR gets larger scope for optimization

• Tracelet body counts
  • Provide both control-flow and type profiling
HipHop Pipeline

$c = $a + $b; // php

```php
$lval: \$c = \$a + \$b surveill\n```

```cpp
_c.assign(_a.add(_b)); // C++
```
Problems with AoT

- PHP development environment suffers
  - Slow
  - New class of bug
  - Perf work different from “normal” work
  - Operational headaches deploying ELF binary

- Missed perf opportunities
  - Runtime has more info than compile-time
  - Type inference
  - Method dispatch
Important related work

• Other dynamic language VMs
  • Smalltalk, Self
  • ECMAscript: v8, JSC, Tamarin, *Monkey, …
  • Other: LuaJIT, PyPy

• Dynamic binary translators
  • Shade
  • Embra
  • VMware’s BT monitor
  • DynamoRIO
HipHop

GeoMean of 5.6x faster than Zend

From “The HipHop Compiler for PHP,” Zhao et al., to appear in OOPSLA 2012
PHP Perf Challenges

- What method/function is being called here?
- What types do these data have?
- Automatic memory management
- High language “surface area”
Prod: Tracelet Chain length
Risk: Warmup

- Possible weak point of JIT vs. AoT: warmup latency
- We start with an empty code cache
- Goal: reach steady state quickly
Warmup: Production requests/second
Code size over time

[Graph showing code size over time with two lines: one for hot code and one for cold code. The y-axis represents bytes ranging from 0 to 130,000,000, and the x-axis represents minutes since boot ranging from 0 to 65.]
Jit output / time
The HipHop VM (HHVM)

• New execution engine for PHP
• ~5x faster than standard interpreter
• No compromises
  • Interactive use same as interpreter
  • Real warts-and-all PHP language
Why PHP?

- Large base of software
- Mediawiki, Drupal, Wordpress
- Large opportunities for performance improvement
Background: PHP

- Server-side
- Single-threaded
- Stateless
  - Web requests run in isolation
  - Request starts with empty heap, runs to completion
  - No thread-to-thread memory sharing
- Untyped
- Highly dynamic
- Facebook
  - > 18 MLOC
  - Significant open source PHP
  - Site pushed twice daily
Dynamic binding: functions

```php
<?

switch(date('l')) {
    case 'Wednesday':
        function foo($a, $b) {
            return "Wednesday!!!" . (32 * $a - $b);
        }
        break;
    default:
        function foo($a, $b) {
            return date('l') . ($a + $b) . "\n";
        }

    echo foo(1, 2);
```