Type Specialization in Dynamic Weakly Typed Languages

Dr David Siegwart    IBM Hursley UK

applied to IBM’s PHP (P8) Engine
presented by Ian Gartley    IBM Toronto Canada
Contents

- **Background**
- Performance Motivation
- How the P8 PHP Engine Works
- Algorithm and Performance Results
- Conclusions
Dynamic Weakly Typed Languages

- **Dynamic**: types may change dynamically.

- **Weakly Typed**: operations that would otherwise fail or give other results in other languages:
  - “3.7” + 1 double
  - “xyz” + 1 null
  - “4” + 2 long

- **Dynamic Weakly Typed Languages**: PHP, Perl

- **Strongly Typed Languages**: Java, Pascal, ADA.

- **We focus on IBM’s PHP P8 Engine, but techniques are generic**
Zend, PHP and IBM’s P8

<table>
<thead>
<tr>
<th>PHP Engine</th>
<th>Written in</th>
<th>PHP Code is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zend</td>
<td>C</td>
<td>Interpreted</td>
</tr>
<tr>
<td>IBM P8</td>
<td>Java</td>
<td>Interpreted, Dynamically Compiled</td>
</tr>
</tbody>
</table>

P8 uses an internal Intermediate Language (IL) Representation.
Contents

- Background
- **Performance Motivation**
- How the P8 PHP Engine Works
- Algorithm and Performance Results
- Conclusions
Top Method of SugarCRM Benchmark

- SugarBean::process_union_list_query(…)

- Salient Feature: while loop
  - with integer induction variable and bounds check:

```php
function loop()
{
  /* locals: $row */
  /* integer: $q */
  $q = 0;
  $index = $this->index;
  $row = $this->arow[$index];
  while ($row)
  {
    if ($index < $this->row_offset + $this->max_per_page)
    {
      $q += strlen($row) + $index;
    }
    $index++;
    $row = $this->arow[$index];
  }
  $this->q = $q;
}
```
Performance Motivation

<table>
<thead>
<tr>
<th></th>
<th>throughput (ops/usec)</th>
<th>% gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original loop</td>
<td>1.97</td>
<td></td>
</tr>
<tr>
<td>generic type → longs</td>
<td>2.47</td>
<td>25%</td>
</tr>
</tbody>
</table>

JVM: IBM Java 5 SR11 for Windows  
sMash: n/a, CLI  
P8: windows-ia32-20100107023555 (Skinny)  
VM opts: -Xjit:optLevel=warm -Xgcpolicy:gencon -Xms512M -Xmx512M -Xmn96M  
P8 opts: -d track_errors=false [debug_replace=bean::simple_loop=<replacement class>]  
woodytwo.hursley.ibm.com: 2x(Quad Core Intel Xeon Clovertown, 2.67GHz, 2x4MB shared L2, 32k I/32k D L1, 1333MHz), Windows 2003 Server x64 SP2  
D.Siegwart, 15 Jan 2010, Preliminary Results, single runs using adapted p8bench.php.
**Literature**

- \textit{type lattice}
  - 1974: Tenenbaum

- \textit{global constant propagation}
  - Kildell 1973
  - Wegbreit 1975
    - + conditional
    - + sparse (SSA form)
      - Cytron et al. 1989/91

- Sparse Conditional Type Specialization 2010
Contents

- Background
- Performance Motivation
- How the P8 PHP Engine Works
- Algorithm and Performance Results
- Conclusions
### Simplified P8 IL

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY [param]</td>
<td>Enter function body [with given parameter]. May be multiple.</td>
</tr>
<tr>
<td>RETURN</td>
<td>Return from function body.</td>
</tr>
<tr>
<td>UNSET var</td>
<td>Set the value of the local variable to “unassigned”.</td>
</tr>
<tr>
<td>PHI var</td>
<td>φ function for local variable var. Used for SSA Form.</td>
</tr>
<tr>
<td>PUSH type(val)</td>
<td>Push the given value of given type onto operand stack.</td>
</tr>
<tr>
<td>DROP</td>
<td>Pop top of stack and discard it.</td>
</tr>
<tr>
<td>STORE var</td>
<td>Pop top of stack and store it as local variable var.</td>
</tr>
<tr>
<td>LOAD var</td>
<td>Load given local variable and pushes it onto stack.</td>
</tr>
<tr>
<td>INC var</td>
<td>Increments given local variable in place.</td>
</tr>
<tr>
<td>ADD [var]</td>
<td>Pop top two values; push sum, [and store in local var].</td>
</tr>
<tr>
<td>ECHO</td>
<td>Pop top of stack and echo it to the terminal.</td>
</tr>
<tr>
<td>CMPLE</td>
<td>Pop top value b and next value a. Push (a &lt;= b) onto stack.</td>
</tr>
<tr>
<td>branch ±offset</td>
<td>Branch relative by offset. Unconditional (BRANCH) or</td>
</tr>
<tr>
<td></td>
<td>conditional (BRFALSE) which pops condition from stack.</td>
</tr>
</tbody>
</table>
Simple Example

```php
function foo($n)
{
    $k = false;
    $i = 1;
    $j = 2;
    while ($i <= $n)
    {
        $j += 2;
        $k = true;
        $i++;
    }
    if ($k)
    {
        echo $j;
    }
    else
    {
        $i++;
    }
    return;
}
```

P8 IL Generated

```
0: ENTRY
1: ENTRY this
2: ENTRY n
3: UNSET i
4: UNSET k
5: UNSET j
6: PUSH bool(false)
7: STORE k
8: PUSH long(1)
9: STORE i
10: PUSH long(2)
11: STORE j
12: LOAD i
13: LOAD n
14: CMPLen
15: BRFALSE +8
16: PUSH long(2)
17: LOAD j
18: ADD
19: STORE j
20: PUSH bool(true)
21: STORE k
22: INC i
23: BRANCH -12
24: LOAD k
25: BRFALSE +3
26: LOAD j
27: ECHO
28: BRANCH +1
29: INC i
30: RETURN
```
PHPValue – State Transitions

<table>
<thead>
<tr>
<th>specialized type</th>
<th>assignment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHPValue</td>
<td>$v_1</td>
<td>$v = &amp; $x;</td>
</tr>
<tr>
<td>long</td>
<td>$v_2</td>
<td>$v = 20;</td>
</tr>
<tr>
<td>long</td>
<td>$v_3</td>
<td>$v += 5;</td>
</tr>
<tr>
<td>PHPValue</td>
<td>$v_4</td>
<td>$v += 2.3;</td>
</tr>
<tr>
<td>PHPValue</td>
<td>$v_5</td>
<td>$v *= 7;</td>
</tr>
<tr>
<td>bool</td>
<td>$v_6</td>
<td>$v = ($v &gt; 6);</td>
</tr>
<tr>
<td>bool</td>
<td>$v_7</td>
<td>$v = true;</td>
</tr>
<tr>
<td>PHPValue</td>
<td>$v_8</td>
<td>$v = a$[0][1];</td>
</tr>
<tr>
<td>long</td>
<td>$v_9</td>
<td>$v = (long) $v;</td>
</tr>
</tbody>
</table>

Without type specialization

With bool, long specialization
Evaluation of P8 IL ADD

- As an example, unspecialized, ADD is evaluated like this:

  \[
  \text{jhADD}(\text{PHPValue } a, \text{PHPValue } b)
  \]

  which adds two values \(a\) and \(b\) of generic type. The JIT cannot do better than this.

- Specialized to longs, ADD can be evaluated like this in bytecode:

  \[
  \text{lload a} \\
  \text{lload b} \\
  \text{ladd}
  \]

  which the JIT can optimize to something like this:

  \[
  \text{ADD RAX, RCX}
  \]

so how can we carry out the type specialization of the P8 IL ADD and other instructions?
Contents

- Background
- Performance Motivation
- How the P8 PHP Engine Works
- Algorithm and Performance Results
- Conclusions
Local Analysis (Simplified)

\[\text{in: } \{q: \text{long}, z: \text{PHPValue}\}\]

\[
\begin{align*}
&a = 1 + 5; \\
&b = q; \\
&c = 'abc'; \\
&a = c; \\
&c = c + z;
\end{align*}
\]

\[\text{assigned types on BB entry}\]

\[
\begin{align*}
&a: \text{long} \\
&b: \text{long} \\
&c: \text{str} \\
&a: \text{PHPValue} \\
&c: \text{PHPValue}
\end{align*}
\]

\[\text{out: } \{q: \text{long}, z: \text{PHPValue}, a: \text{PHPValue}, b: \text{long}, c: \text{PHPValue}\}\]
Type Lattice

$\top$ unassigned type

{bool, str} bool long str ... double {double, long}

$\bot$ cannot be constant type (many)
Local Analysis (Lattice)

\[
\begin{align*}
\text{in}(BB): \{&q: \text{long}, \ z: \text{many}\} \\
&\{ \\
&a = 1 + 5; \\
&b = q; \\
&c = 'abc'; \\
&a = c; \\
&c = c + z; \\
\} \\
\text{out}(BB): \{&q: \text{long}, \ z: \text{many}, \ a: \text{many}, \\
&b: \text{long}, \ c: \text{many}\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Locals Type Lattices</th>
<th>$q$</th>
<th>$z$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>long</td>
<td>many</td>
<td>many</td>
<td>long</td>
<td>many</td>
</tr>
</tbody>
</table>
Type Expression Rule for ADD for PHP

First need to convert the input operands into a number

<table>
<thead>
<tr>
<th>var</th>
<th>num(var)</th>
</tr>
</thead>
<tbody>
<tr>
<td>long, null, bool, object, or resource</td>
<td>long</td>
</tr>
<tr>
<td>double, array</td>
<td>unchanged</td>
</tr>
<tr>
<td>str</td>
<td>{long, double, null}</td>
</tr>
</tbody>
</table>

Symmetric Type Expression Rules for ADD

<table>
<thead>
<tr>
<th>num(operand 1)</th>
<th>num(operand 2)</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>double</td>
<td>{double, long}</td>
<td>double</td>
</tr>
<tr>
<td>array</td>
<td>array</td>
<td>array</td>
</tr>
<tr>
<td>otherwise</td>
<td></td>
<td>null (err)</td>
</tr>
</tbody>
</table>
Sparse Conditional Type Specialization

- For each P8 IL op define a type specialization rule
- **Build Structural Components of the code to be specialized**
- Solve the Data Flow Equations
- Determine which expressions may be type specialized
- Generate Specialized ByteCode
Introduction to SSA (Static Single Assignment) Form

A simple example illustrates:

\[
\begin{align*}
x & \leftarrow a \\
x & \leftarrow b \\
y & \leftarrow c + x
\end{align*}
\]

SSA

\[
\begin{align*}
x_1 & \leftarrow a_1 \\
x_2 & \leftarrow b_1 \\
y_1 & \leftarrow c_1 + x_2
\end{align*}
\]
SSAEdges

\[ x \leftarrow \ldots \]
\[ y \leftarrow c + x \]

SSA

\[ x_2 \leftarrow \ldots \]
\[ y_1 \leftarrow c_1 + x_2 \]
Control Flow: FlowEdges

**Straight line**

```
LOAD x
ADD
```

- incoming edges = 1
- outgoing edges = 1

**Branch**

```
LOAD v
PUSH long(3)
BRFALSE
```

- outgoing edges > 1

```
PUSH long(5)
```

**Join**

```
LOAD v
STORE x
```

- incoming edges > 1
SSA Form for Control Flow

What happens at a join? Which definition of a variable should be used?

```
a_1 \leftarrow 0
\downarrow
\hline
c_1 \leftarrow 1
\downarrow
\leftarrow \text{if } c_1 < 1 \text{ then }
\hline
Y
\downarrow
a_2 \leftarrow 5
\hline
N
\downarrow
a_3 \leftarrow 2
\downarrow
\hline
J
\downarrow
\text{if } c_2 \leftarrow c_1 + a_2 \text{ then }
\hline
\text{return } c_2
```
SSA Form for Control Flow

```
\begin{align*}
a_1 & \leftarrow 0 \\
c_1 & \leftarrow 1 \\
c_1 \&< 1 \\
a_2 & \leftarrow 5 \\
a_3 & \leftarrow 2 \\
\end{align*}
```

- If $c_1 < 1$, then:
  - $a_4 \leftarrow \varphi(a_2, a_3)$
  - $c_2 \leftarrow c_1 + a_4$
  - return $c_2$

- If not $c_1 < 1$, then:
  - return $c_2$
Simple Example of $\phi$ Function Insertion

```
0: ENTRY
1: ENTRY this
2: ENTRY n
3: UNSET i
4: UNSET k
5: UNSET j
6: PUSH bool(false)
7: STORE k
8: PUSH long(1)
9: STORE i
10: PUSH long(2)
11: STORE j
12: LOAD i
13: LOAD n
14: CMPLE
15: BRFALSE +8
16: PUSH long(2)
17: LOAD j
18: ADD
19: STORE j
20: PUSH bool(true)
21: STORE k
22: INC i
23: BRANCH -12

24: LOAD k
25: BRFALSE +3
26: LOAD j
27: ECHO
28: BRANCH +1

29: INC i
30: RETURN
```

```c
0: ENTRY
1: ENTRY this
2: ENTRY n
3: UNSET i
4: UNSET k
5: UNSET j
6: PUSH bool(false)
7: STORE k
8: PUSH long(1)
9: STORE i
10: PUSH long(2)
11: STORE j
12: PHI j
13: PHI k
14: PHI i
15: LOAD i
16: LOAD n
17: CMPLE
18: BRFALSE +8
19: PUSH long(2)
20: LOAD j
21: ADD
22: STORE j
23: PUSH bool(true)
24: STORE k
25: INC i
26: BRANCH -15

27: LOAD k
28: BRFALSE +3
29: LOAD j
30: ECHO
31: BRANCH +1
32: INC i
33: PHI i
34: RETURN
```
Sparse Conditional Type Specialization

- For each P8 IL op define a type specialization rule
- Build Structural Components of the code to be specialized
- **Solve the Data Flow Equations**
- Determine which expressions may be type specialized
- Generate Specialized ByteCode
Solve The Data Flow Equations

- Perform symbolic execution of the program using the FlowEdges and the SSA graph to transmit lattice type information along their edges.
  - Assign initial values of each P8 IL instruction in the SSA form code to the value unassigned (top, T).
  - Follow FlowEdges through the program visiting each instruction once, at each instruction:
    - **[visit]**: symbolically execute the type expression rule for that instruction, and update any SSA variables.
  - When the lattice type of an SSA variable changes, symbolic re-execute each of its uses as in [visit].

- Continue until all instructions have been executed once, and no more SSA variables are changing.
Sparse Conditional Type Specialization (SCTS)

- For each P8 IL op define a type specialization rule
- Build Structural Components of the code to be specialized
- Solve the Data Flow Equations
- **Determine which expressions may be type specialized**
- **Generate Specialized ByteCode**
Type Specialization Example

```plaintext
function loop_int() {
  $q = 0;
  $index = 0;
  $row_offset = 100;
  $max_per_page = 500;
  $strlen = 1;
  while ($index < 1000)
  {
    if ($index < $row_offset + $max_per_page)
    {
      $strlen++;
      $q += $strlen + $index;
    }
    $index++;
  }
  return $q;
}
```

```
0: PUSH int(0)
1: STORE q
2: PUSH int(0)
3: STORE index
4: PUSH int(100)
5: STORE row_offset
6: PUSH int(500)
7: STORE max_per_page
8: PUSH int(1)
9: STORE strlen
10: LOAD index
11: PUSH int(1000)
12: CMPLT
13: BRFALSE +15
14: LOAD index
15: LOAD max_per_page
16: LOAD row_offset
17: ADD
18: CMPLT
19: BRFALSE +7
20: INC strlen
21: ADD
22: LOAD strlen
23: ADD
24: LOAD q
25: ADD q
26: DROP
27: INC index
28: BRANCH -19
29: LOAD q
30: RETURN
```

Type Specialization Example

Unspecialized
operators are on PHPValues

Specialized
operators are on longs

PHPValue $q, $strlen, $index;
...
Op.jhADD_INPLACE($q,
Op.jhADD($strlen, $index));

long $q, $strlen, $index;
...
$q = ADD_long($q,
ADD_long($strlen, $index));

$q += $strlen + $index;

21: LOAD index
22: LOAD strlen
23: ADD
24: LOAD q
25: ADD q

24: <long > LOAD index
25: <long > LOAD strlen
26: <long > ADD
27: <long > LOAD q
28: <long > ADD q
Performance Results

<table>
<thead>
<tr>
<th></th>
<th>throughput (ops/usec)</th>
<th>speed up factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unspecialized</td>
<td>28.11</td>
<td></td>
</tr>
<tr>
<td>Specialized</td>
<td>392.63</td>
<td>14.0</td>
</tr>
</tbody>
</table>

JVM: IBM J9 VM SR15 (build 2.4, JRE 1.6.0 IBM J9 2.4 Windows 7 amd64-64)
Lenovo W530 Intel i7-3720QM 2.60GHz, Windows 7
D.Siegwart, 3 Nov 2014, Preliminary Results, single runs, each loop repeated 200,000 times to give one iteration. 50 iterations discarded, next 10 iterations averaged.
Conclusions

- Type Specialization is necessary for performance of Server-side code written in dynamic weakly typed languages.

- Tight loops that can benefit from type specialization.

- Hand-specializing of one of these loops gave a 25% gain.

- For a strictly numeric-only loop based on this loop, and using SCTC we were able to achieve a speedup of a factor of 14 over non-specialized P8.

- Thus the work has huge speed up for purely arithmetic code, and potential for gains on server-side performance too.
References


also


References (cont.)


Additional Slides
SSA Locals Map

The variable $x_2$ may be retrieved from the local map by using SSAEdge.

For instance suppose we are looking at the edge that enter the operand of $op$. We get the name of the variable “$x$” from the assignment instruction then use the assignment itself and name as a key to look up the SSA variable in the SSA Local Map. We never need a label for $x$, because we use the assignment itself as the label.
Type Lattice (With Meet Join operators)

\[ \top \text{ unassigned type} \]

\{bool, str\} \quad \text{bool} \quad \text{long} \quad \text{str} \quad \cdots \quad \text{double} \quad \{\text{double, long}\}

\[ \bot \text{ cannot be constant type (many)} \]

for \( a \neq b \):

- meet: \( a \land b = \bot \)
- join: \( a \lor b = \top \)
- \( a \land a = a \lor a = a \)

other relationships are as for a standard lattice.
Data Flow Analysis

- Transfer Function $F_B$ for basic block $B$:
  
  $$\text{out}(B) = F_B(\text{in}(B))$$

  - for instance for the example on slide 2, a transfer function $F$ transforms $\text{in}(B) = (\text{long}, \bot, \top, \top, \top)$ to $\text{out}(B) = (\text{long}, \bot, \top, \text{long}, \top)$.

- $\text{out}(B)$ flows down all outgoing edges $e_o$ as $J(e_o) := \text{out}(B)$ emanating from $B$.

- Similarly, the flow into $B$ originates from the flow $J(e_i)$ from all the incoming edges $e_i$, thus given as the meet of those flows:
  
  $$\text{in}(B) = \bigwedge_{e \in e_i(B)} J(e)$$

- These flow equations form a dynamical system over the blocks $B$. The transfer functions are monotonically increasing, and therefore the system converges to a fixed point. By iterating the flow equations, the solution of them $\text{out}$(exit) gives the variables that have constant type.
LatticeTypes for Local Assignment Ops/SSALocals

- At Each Local Assignment Op:
  - the `targetType` of the operation is evaluated (from its `TypeExpressionRules`).
  - the simulated type $t_v$ of the local $v$ is maintained.
  - the type is updated using $t_v \leftarrow \text{meet}(t_v, \text{targetType})$.

- At the end of the analysis of the straight line sequence, the `ssaLocalMap` contains references to the final types, that can then be used during bytecode generation.
Type Specialization to long for P8 IL ADD

For ADD first one has to convert the operands according to this table:

<table>
<thead>
<tr>
<th>operand 1 ⊆ {long, null, bool, object, resource}</th>
<th>operand 2 ⊆ {long, null, bool, object, resource}</th>
<th>convert to Java long</th>
</tr>
</thead>
<tbody>
<tr>
<td>otherwise</td>
<td>otherwise</td>
<td>do not convert</td>
</tr>
</tbody>
</table>

Then generate

- specialized ADD if result is \{long\}
- unspecialized ADD if result is not long.

To get the operands in the correct form there may be some “boxing” or “unboxing” required to and from the PHPValue to the primitive Java long, as the specialized ADD will act on Java longs. Similarly will need to generate specialized lload and lstore for the long SSA variables.
Boxing and Unboxing

During type specialization we often need to convert variables from PHPValues to longs (unboxing) and from longs to PHPValues (boxed).

To do this we generate calls in the bytecode to simple methods like:

```java
PHPValue convertTypeTo_PHPValue(long value) {
    return PHPValue.boxLong(value);
}
```

```java
long convertTypeTo_long(PHPValue value) {
    return value.getLong();
}
```

before we use the primitive types directly. These will get inlined. Similarly our specialized operations are often simple methods, which will be inlined readily eg:

```java
long ADD_returns_long(long v1, long v2) { return v1 + v2; }
```