Supporting Objects in Run-Time Bytecode Specialization

Reynald Affeldt, Hidehiko Masuhara, Eijiro Sumii, Akinori Yonezawa
University of Tokyo
Run-Time Specialization (RTS)

RTS optimizes program code at run-time

More precisely:

\[
\text{static input} + \text{original code} \xrightarrow{\text{RTS}} \text{residual code}
\]

Typical applications:

- computations done:
  - repeatedly with similar inputs
  - with an unfortunate timing
- input not available at compile-time
Motivation

Optimize object-oriented (OO) programs by RTS

OO programs are typically slower than imperative programs:
  • they are more generic
  • object-orientation is costly

RTS is well adapted:
  • specialization trades genericity for performance
  • it is a general optimization technique
  • RTS has proved to be efficient for several languages
Contributions

Design and implement RTS for an OO language, namely Java:

- efficient residual code regarding OO overheads
  - elimination of dynamic allocation
  - elimination of memory accesses
    (including destructive updates)
  - elimination of virtual dispatches
- better automation of the specialization process
  - as few annotations by the user as possible
- correctness statement

We hope it can lead ultimately to:

- a system easier to use
- favoring extensive residual code reuse
Outline

1. **Effectiveness of OO Specialization**
2. Potential Problems with Objects
3. Techniques for Correctness and Efficiency
4. Generalization and Formalization
5. Preliminary Experiments
6. Conclusion and Future Work
Complex Arithmetic

A class for complex numbers:

```java
class Complex {
    float re, im;
    Complex mul (Complex z) {
        return new Complex (...);
    }
    Complex add (Complex c) {
        return new Complex (...);
    }
}
```

A complex function:

```java
// f(z, c) = z · z + c
Complex f (Complex z, Complex c) {
    Complex prod = z.mul (z);
    return prod.add (c);
}
```
Computation of an array of complex numbers:

```c
for (int i = 0; i < n; n++) {
    c[i] = f(a[i], b[i]);
}
```

Assume that a[i] happens to be always i

⇒ Optimization by specialization of f w.r.t. its first argument
Off-Line Specialization

\[ z \text{ static, } c \text{ dynamic} \]

```java
Complex f (Complex z, Complex c) {
    Complex prod = z.mul(z);
    return prod.add(c);
}

Complex mul (Complex z) {
    return new Complex
        (re * z.re - im * z.im,
         re * z.im + im * z.re);
}

Complex add (Complex c) {
    return new Complex
        (re + c.re, im + c.im);
}
```

\[ z = i \]

```java
// f(res(c)) = -1 + c
Complex f_res (Complex c) {
    return new Complex
        (-1 + c.re, 0 + c.im);
}
```

The residual code features:
- less calculations
- less object creations
- less method calls

\( \Rightarrow \) OO specialization is effective
Outline

1. Effectiveness of OO Specialization
2. Potential Problems with Objects
3. Techniques for Correctness and Efficiency
4. Generalization and Formalization
5. Preliminary Experiments
6. Conclusion and Future Work
One-Dimensional Geometry

A class for one-dimensional points:

```java
class Point {
    int x = 0;
    void update (int a) { x = x + a; }
    static Point make (int s, int d) {
        Point p = new Point();
        p.update (s);
        p.update (d);
        p.update (s);
        return p;
    }
}
```
Original Application

Computation of two one-dimensional points:

```java
int u = Console.getInt();
Point a = Point.make(u, 7);
Point b = Point.make(u, 11);
int v = a.x + b.x;
int w = a == b;
```

⇒ Specialization of make w.r.t. u
Naive and Incorrect Off-Line Specialization

static Point make (int s, int d) {
    Point p = new Point ();
    p.update (s);
    p.update (d);
    p.update (s);
    return p;
}

s = 42

static Point make_res (int d) {
    _p.update (d);
    _p.update (42);
    return _p;
}

(_p is the point created during specialization; we say it is stored in the specialization store)
Problems with Objects

The original application cannot be simply rewritten:

```c
int u = Console.getInt ();
Point a = Point.\textcolor{red}{\textbf{make}} (u, 7);
Point b = Point.\textcolor{red}{\textbf{make}} (u, 11);
int v = a.x + b.x;  // 91 + 95
int w = a == b;    // false
```

```c
\textcolor{blue}{\textbf{Original cause: Application, specializer and residual code share the same heap}}
```

```c
int u = \textcolor{blue}{\textbf{Console.getInt ()};}
Point a = \textcolor{blue}{\textbf{make_res}} (7);
Point b = \textcolor{blue}{\textbf{make_res}} (11);
int v = a.x + b.x;  // 144 + 144
int w = a == b;    // true
```
Approaches

Immediate approaches:

- perform over-specialization
- require annotations by the user
- enforce residualization

⇒ None is satisfactory

Our approach:

- as few annotations as possible
- efficiency achieved by improving specialization rules
Outline

1. Effectiveness of OO Specialization
2. Potential Problems with Objects
3. Techniques for Correctness and Efficiency
4. Generalization and Formalization
5. Preliminary Experiments
6. Conclusion and Future Work
About Specialization Rules (1/2)

Main idea: distinguish operations in terms of staticness

For instance, memory accesses as in statements of the form:

\[ \text{lhs} = p.x; \]

- if \( p.x \), then the memory access can be evaluated during specialization
- if \( p.x \), then the memory access must be residualized during specialization

But in general, this static/dynamic dichotomy is not sufficient
Key idea: distinguish operations in terms of visibility

For instance, (static) object creations as in statements of the form:

\[ \text{lhs} = \text{new class\_name}(\ldots); \]

or (static) destructive updates as in statements of the form:

\[ \text{p.x} = \text{rhs}; \]

- if \text{visible}, residualization and evaluation during specialization
- if \text{invisible}, evaluation during specialization
“If Visible, Residualization and Evaluation”

s static, d dynamic

```
static Point make(int s, int d) {
    Point p = new Point();
    p.update(s);
    p.update(d);
    p.update(s);
    return p;
}
```

```
static Point make_res(int d) {
    Point p = new Point();
    p.x = 42 + d;
    p.x = p.x + 42;
    return p;
}
```

- Enforced residualization guarantees **correctness**
- Evaluation during specialization enables **efficient** residual code
“If Invisible, Evaluation” (1/2)

Extraction of small segments:

```java
Set set = new Set();
for (int i = 0; i < n; i++) {
    if (areClose(a[i], b[i]))
        set.add(new Segment(a[i], b[i]));
}
```

Assume that a[i] happens to be always 42

⇒ Optimization by specialization of areClose w.r.t. its first argument
“If Invisible, Evaluation” (2/2)

\[
s \text{ static, } d \text{ dynamic}
\]

boolean `areClose` (int s, int d) {
    Point a = new \text{INVIS} Point ();
    Point b = new \text{INVIS} Point ();
    a.update (s);
    b.update (d);
    return a.distance (b) < 10;
}

boolean `areClose_res` (int d) {
    _b.update (d);
    return _a.distance (_b) < 10;
}

\(_b \text{ and } _a \text{ are the points stored in the specialization store}\)

- Reuse of objects yield more \textit{efficient} residual code
- Specialization of destructive updates does not infringe \textit{correctness}
Outline

1. Effectiveness of OO Specialization
2. Potential Problems with Objects
3. Techniques for Correctness and Efficiency
4. Generalization and Formalization
5. Preliminary Experiments
6. Conclusion and Future Work
Correctness Statement for RTS

Two components:

1. **valid code replacement** :
   the residual code may substitute for the original code
   whenever the static input is used

2. **valid specialization usage** :
   RTS may happen
   as soon as the static input is available
Valid Code Replacement

Mix equation (reminder):

\[ t = f(s, d); \quad \leftrightarrow \quad t = f_s(d); \]
Valid Code Replacement

Mix equation (extended with heaps):

\[
(t, H_t) = f(s, H_s, d, H_d);
\]

\[
(t, H_t) = f_{s, H_s}(d, H_d);
\]

⇒ Describe arguments and results in terms of:

- **heap equivalence** (including a notion of *reachability*)

- additional requirements for the values of references
  - because of *reference lifting*
  - because references can be compared

(see the paper for more details)
Valid Code Replacement

Example:

```
Point a = Point.make(s, d);
```

```
Point a' = make_res(d);
```

Condition on arguments:
s is expected to be indeed 42

Condition on results:
Points a and a’ must have the same coordinate

Additional requirement:
a and a’ must be fresh references
Valid Specialization Usage

Informally:

\[
\begin{align*}
\text{statement}_1; \\
f_s &= \text{spec} (f, s); \\
\text{statement}_2; \\
\text{statement}_3; \\
t &= f_s (d);
\end{align*}
\]

⇒ Specify the interactions between specialization and the application:

- *specialization* cannot break the semantics of the *application*
- the *application* cannot break the semantics of *specialization*
Valid Specialization Usage

Example:

\[
\text{statement}_1; \\
\text{make\_res} = \text{spec} (\text{make, s}); \\
\text{statement}_2; \\
\text{statement}_3; \\
\text{Point a} = \text{Point. make\_res} (d); \\
\]

\[
\text{statement}_1; \\
\text{statement}_2; \\
\text{make\_res} = \text{spec} (\text{make, s}); \\
\text{statement}_3; \\
\text{Point a'} = \text{make\_res} (d); \\
\]

Condition on the interaction:

\text{spec} \text{ cannot perform \textit{visible} side-effects}
Outline

1. Effectiveness of OO Specialization
2. Potential Problems with Objects
3. Techniques for Correctness and Efficiency
4. Generalization and Formalization
5. Preliminary Experiments
6. Conclusion and Future Work
Implementation Strategy

Based on Masuhara and Yonezawa’s BCS:

- RTS for the Java bytecode language
- end-to-end bytecode-level approach:
  - type-based binding-time analysis
  - cogen-by-hand approach
  - run-time code generation

Extended to:

- an OO subset of the Java bytecode language
- new rules for binding-time analysis and code generation
- interface with compile-time analyses
Implementation Overview

Original code
- Compile-time analyses
- Binding-time specification
- Annotated method
  - Code generator
  - Specializer
  - BCS

Binding-time analysis

Off-line

Original application
- Static values
- Dynamic values
- Results

Rewritten application
- Specializer
- Residual code

Run-time
Performance Measurements

Test Programs:

Object-oriented version of standard applications:

- Power function
- Mandelbrot sets drawer
- Ray tracer

Environment for Experiments:

Standard virtual machines with Just-in-time compilation
## Power Function

<table>
<thead>
<tr>
<th></th>
<th>Speed-up</th>
<th>Recursive</th>
<th>Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UltraSparc</strong></td>
<td>Hotspot (Sun 1.3)</td>
<td>5.4</td>
<td>1.5</td>
</tr>
<tr>
<td><strong>Intel x86</strong></td>
<td>Hotspot (Sun 1.3)</td>
<td>1.9</td>
<td>1.3</td>
</tr>
<tr>
<td><strong>Intel x86</strong></td>
<td>Classic (IBM 1.3)</td>
<td>5.9</td>
<td>4.4</td>
</tr>
</tbody>
</table>

## Mandelbrot Sets Drawer

<table>
<thead>
<tr>
<th></th>
<th>Speed-up</th>
<th>eval</th>
<th>eval_res</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UltraSparc</strong></td>
<td>Hotspot (Sun 1.3)</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td><strong>Intel x86</strong></td>
<td>Hotspot (Sun 1.3)</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td><strong>Intel x86</strong></td>
<td>Classic (IBM 1.3)</td>
<td>1.05</td>
<td></td>
</tr>
</tbody>
</table>
## Ray Tracer

<table>
<thead>
<tr>
<th>Speed-up</th>
<th>Overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specialization</td>
</tr>
<tr>
<td></td>
<td>Subject method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler Version</th>
<th>Speed-up</th>
<th>Subject Method Overhead</th>
<th>Residual Code Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>UltraSparc</td>
<td>Hotspot (Sun 1.3)</td>
<td>1.18</td>
<td>10</td>
<td>196</td>
</tr>
<tr>
<td>Intel x86</td>
<td>Hotspot (Sun 1.3)</td>
<td>1.25</td>
<td>7</td>
<td>115</td>
</tr>
<tr>
<td>Intel x86</td>
<td>Classic (IBM 1.3)</td>
<td>1.26</td>
<td>6</td>
<td>208</td>
</tr>
</tbody>
</table>

### Break-even points

<table>
<thead>
<tr>
<th>Compiler Version</th>
<th>No JIT overhead</th>
<th>JIT overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotspot (Sun 1.3)</td>
<td>5,646 ~ 138,421</td>
<td>&lt; 0 ~ 9,755</td>
</tr>
<tr>
<td>Classic (IBM 1.3)</td>
<td>277,582</td>
<td>174,939</td>
</tr>
</tbody>
</table>
Measurements’ Summary

Speed-ups are comparable to related work:

- compile-time specialization for Java
- run-time specialization for C++

The environment for experiments complicates interpretation:

- unfriendly environment:
  - dynamic compilation → more overhead
  - small time window → less optimizations

- overlapping optimizations
- behavior hard to predict
Outline

1. Effectiveness of OO Specialization
2. Potential Problems with Objects
3. Techniques for Correctness and Efficiency
4. Generalization and Formalization
5. Preliminary Experiments
6. Conclusion and Future Work
Related Work: Compile-time Techniques

Compile-time specialization for C:

- C-Mix [Andersen93]
- Tempo [Consel & Noël96]

Specialization and object-orientation:

- Elimination of virtual dispatches [Lea90, Dean et al.94]
- Partial evaluation formalization and implementation [Schultz99-01]

Partial evaluation during interpretation:

- Correctness and experiments [Asai01]
Related Work: Run-time Techniques

Run-time specialization for imperative languages:
- Tempo [Consel & Noël96]
- DyC [Grant et al.97]
- BCS [Masuhara & Yonezawa01]

Run-time specialization for object-oriented languages:
- C++ [Fujinami98]
- Specialization classes [Volanschi et al.97]
Conclusion

Design RTS for an OO subset of Java:

- efficient residual code regarding OO operations
- better automation of the specialization process
- correctness statement

Experimental implementation:

- end-to-end bytecode-level approach
- effective in practice (e.g., 26% speed-up for a ray tracer)
Future Work

Complete the implementation:

- access modifiers, constructors, ...

Increase effectiveness:

- selective inlining
- allow visible side-effects during specialization

Reuse of objects in the specialization store as presented here:

- is not thread-safe
- may withhold many objects

Formal proof of correctness