Next In Line, Please!

Exploiting the Indirect Benefits of Inlining by Accurately Predicting Further Inlining
Next In Line, Please!
Exploiting the Indirect Benefits of Inlining by Accurately Predicting Further Inlining

Efficiently Estimating and Exploiting the Indirect Benefits of Inlining

Master-Thesis von Jannik Jochem
Mai 2011
Background: The Direct and Indirect Benefits of Inlining

Direct Benefits
- No stack frame creation
- No call/return overhead
- (Possibly) no dynamic dispatch

Indirect Benefits
- Constant folding
- Elimination of type checks
- Elimination of null checks
- Elimination of array bounds checks

Further, guardless Inlining
Background: The Direct and Indirect Benefits of Inlining

Direct Benefits

- No stack frame creation
- No call/return overhead
- (Possibly) no dynamic dispatch

Indirect Benefits

- Constant folding
- Elimination of type checks
- Elimination of null checks
- Elimination of array bounds checks

Further, guardless Inlining
### Background: The Direct and Indirect Benefits of Inlining

#### Direct Benefits
- No stack frame creation
- No call/return overhead
- (Possibly) no dynamic dispatch

#### Indirect Benefits
- Constant folding
- Elimination of type checks
- Elimination of null checks
- Elimination of array bounds checks
- Further, guardless Inlining
Background: The Direct and Indirect Benefits of Inlining

Direct Benefits
- No stack frame creation
- No call/return overhead
- (Possibly) no dynamic dispatch

Indirect Benefits
- Constant folding
- Elimination of type checks
- Elimination of null checks
- Elimination of array bounds checks
- Further, guardless Inlining
Background: When is Further, Guardless Inlining Possible?

Precise Arguments

class A1 {
    void m() {
        B b = ...;
        C c = new D();
        // Precise type of argument is D.
        b.n(c);
    }
}

class D extends C { ... }
Background: When is Further, Guardless Inlining Possible?

Precise Arguments

```java
class A1 {
    void m() {
        B b = ...
        C c = new D();
        // Precise type of argument is D.
        b.n(c);
    }
}

class D extends C { ...
}
```

Extant Arguments

```java
class A2 {
    void m(C c) {
        B b = ...
        // Argument exists before call
        // to A2.m(C).
        b.n(c);
    }
}
```
Background: When is Further, Guardless Inlining Possible? And When Obviously Not?

Precise Arguments

```java
class A1 {
    void m() {
        B b = ...
        C c = new D();
        // Precise type of argument is D.
        b.n(c);
    }
}
```

class D extends C {
    void n(C c) {
        this.f = c;
    }
}

Extant Arguments

```java
class A2 {
    void m(C c) {
        B b = ...
        // Argument exists before call
        // to A2.m(C).
        b.n(c);
    }
}
```
Background: The Inline Oracle of Jikes RVM
1. Reject @NoInline or native methods.
1. Reject @NoInline or native methods.
2. Accept trivial callees.
1. Reject @NoInline or native methods.
2. Accept trivial callees.
3. Identify targets in dynamic call graph.
Background: The Inline Oracle of Jikes RVM

1. Reject @NoInline or native methods.
2. Accept trivial callees.
3. Identify targets in dynamic call graph.
4. For each dynamic target, estimate its size and decide whether to inline.
Background: The Inline Oracle of Jikes RVM

1. Reject @NoInline or native methods.
2. Accept trivial callees.
3. Identify targets in dynamic call graph.
4. For each dynamic target, estimate its size and decide whether to inline.
5. Choose appropriate guards.
1. Reject @NoInline or native methods.
2. Accept trivial callees.
3. Identify targets in dynamic call graph.
4. For each dynamic target, estimate its size and decide whether to inline.
5. Choose appropriate guards.
## Problem: Jikes RVM’s Assumed Size Reductions

<table>
<thead>
<tr>
<th>Reduction</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference argument of precise type</td>
<td>15%</td>
</tr>
<tr>
<td>Reference argument pre-exists method call</td>
<td>5%</td>
</tr>
<tr>
<td>Non-null object constant</td>
<td>10%</td>
</tr>
<tr>
<td>null constant</td>
<td>10%</td>
</tr>
<tr>
<td>Integer constant</td>
<td>5%</td>
</tr>
<tr>
<td>Array argument of precise type</td>
<td>5%</td>
</tr>
<tr>
<td>No aastore check required</td>
<td>2%</td>
</tr>
<tr>
<td>Reduction</td>
<td>Reduction</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Reference argument of precise type</td>
<td>15 %</td>
</tr>
<tr>
<td>Reference argument pre-exists method call</td>
<td>5 %</td>
</tr>
<tr>
<td>Non-null object constant</td>
<td>10 %</td>
</tr>
<tr>
<td>null constant</td>
<td>10 %</td>
</tr>
<tr>
<td>Integer constant</td>
<td>5 %</td>
</tr>
<tr>
<td>Array argument of precise type</td>
<td>5 %</td>
</tr>
<tr>
<td>No astore check required</td>
<td>2 %</td>
</tr>
</tbody>
</table>
Proposed Solution: Award Size Reductions Only When Further Inlining Likely

<table>
<thead>
<tr>
<th>Reduction</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference argument of precise type</td>
<td>0%</td>
</tr>
<tr>
<td>Reference argument pre-exists method call</td>
<td>0%</td>
</tr>
<tr>
<td>Non-null object constant</td>
<td>10%</td>
</tr>
<tr>
<td>null constant</td>
<td>10%</td>
</tr>
<tr>
<td>Integer constant</td>
<td>5%</td>
</tr>
<tr>
<td>Array argument of precise type</td>
<td>5%</td>
</tr>
<tr>
<td>No aastore check required</td>
<td>2%</td>
</tr>
</tbody>
</table>
Proposed Solution: How to Determine Likelihood of Further Inlining?

1. Reject @NoInline or native methods.
2. Accept trivial callees.
3. Identify targets in dynamic call graph.
4. For each dynamic target, estimate its size and decide whether to inline.
5. Choose appropriate guards.
Proposed Solution: Identify Precise- and Extant-Induced Edges

class A1 {
    void m() {
        B b = ...
        C c = new D();
        // Precise type of argument is D.
        b.n(c);
    }
}

// Extant-induced edges work
Proposed Solution: Identify Precise- and Extant-Induced Edges

class A1 {
    void m() {
        B b = ...
        C c = new D();
        // Precise type of argument is D.
        b.n(c);
    }
}

November 3, 2011 | Andreas Sewe, Jannik Jochem, Mira Mezini | 8
Proposed Solution: Identify Precise- and Extant-Induced Edges

class A1 {
    void m() {
        B b = ...
        C c = new D();
        // Precise type of argument is D.
        b.n(c);
    }
}

Extant-induced edges work similarly.
Proposed Solution: Identify Precise- and Extant-Induced Edges

class A1 {
    void m() {
        B b = ...
        C c = new D();
        // Precise type of argument is D.
        b.n(c);
    }
}

Extant-induced edges work similar.
Evaluation: The Prediction Problem of Further Inlining

Problem
Given the decision to inline B.n(C) into A.m(), predict whether further inlining of C.o() into B.n(C) will occur—and only then award a size reduction.
Evaluation: The Prediction Problem of Further Inlining

Problem
Given the decision to inline \texttt{B.n(C) into A.m()}, predict whether further inlining of \texttt{C.o()} into \texttt{B.n(C)} will occur—and only then award a size reduction.
Problem
Given the decision to inline B.n(C) into A.m(), predict whether further inlining of C.o() into B.n(C) will occur—and only then award a size reduction.
Evaluation: Per-Decision Quality of Inlining Heuristics (Precision)

true positives

true positives + false positives
Evaluation: Per-Decision Quality of Inlining Heuristics (Recall)

true positives
true positives + false negatives
Evaluation: Per-Decision Quality of Inlining Heuristics (F1-Measure)

\[
2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]
Evaluation: Performance Measurements with Replay-Compilation

Proposed Inlining Heuristic saves compile time

Proposed Inlining Heuristic (sometimes) increases program time

10 Compilation Plans
10 Compilation Plans
• Proposed Inlining Heuristic saves compile time
Evaluation: Performance Measurements with Replay-Compilation

- 10 Compilation Plans
- Proposed Inlining Heuristic saves compile time
- Proposed Inlining Heuristic (sometimes) increases program time
Evaluation: Performance Measurements with and without Replay-Compilation

![Graphs showing performance measurements with and without replay- compilation.](image-url)
Evaluation: Speed-up on different architectures
(AMD Athlon 64, Intel Core i7)

With replay

Without replay
Open Questions and Future Work

Assumption \( x \)-induced edge \( \Rightarrow \) call on \( x \) argument valid?
And is gathering exact information worth it?
Replay compilation right methodology?
How to account for other indirect benefits (checkcast, etc.)
How to better spend compile time saved?

```java
Collections.sort(list, new Comparator() {
    int compare(Object lhs, Object rhs) {
        ...
    }
});
```
Open Questions and Future Work

- Assumption “x-induced edge $\Rightarrow$ call on x argument” valid?
Open Questions and Future Work

- Assumption “x-induced edge ⇒ call on x argument” valid? And is gathering exact information worth it?
Open Questions and Future Work

- Assumption “x-induced edge ⇒ call on x argument” valid? And is gathering exact information worth it?
- Replay compilation right methodology?
Open Questions and Future Work

- Assumption “x-induced edge \Rightarrow call on x argument” valid? And is gathering exact information worth it?
- Replay compilation right methodology?
- How to account for other indirect benefits (checkcast, etc.)
Open Questions and Future Work

- Assumption “x-induced edge \(\Rightarrow\) call on x argument” valid? And is gathering exact information worth it?
- Replay compilation right methodology?
- How to account for other indirect benefits (checkcast, etc.)
- How to better spend compile time saved?
Open Questions and Future Work

- Assumption “x-induced edge ⇒ call on x argument” valid? And is gathering exact information worth it?
- Replay compilation right methodology?
- How to account for other indirect benefits (checkcast, etc.)
- How to better spend compile time saved?

```java
Collections.sort(list, new Comparator() {
    int compare(Object lhs, Object rhs) {
        ...
    }
});
```