Directing Function
Inlining with Post-Inlining
Benefits

Erick Ochoa
echoa@ualberta.ca

Andrew Craik
ajcraik@ca.ibm.com

Karim Ali
karim@ualberta.ca

J. Nelson Amaral
jamaral@ualberta.ca
Function Inlining

System.out.println("Hello, " + person.getName());
Function Inlining

System.out.println("Hello, " + person.getName());
Function Inlining

System.out.println("Hello, \" + person.getName());

23: aload_1
24: invokevirtual #9 // Person.getName:()Ljava/lang/String;
System.out.println("Hello, " + person.getName());

23: aload_1
24: invokevirtual #9 // Person.getName():()Ljava/lang/String;
System.out.println("Hello, " + person.getName());

23: aload_1
24: invokevirtual #9 // Person.getName:()Ljava/lang/String;

public java.lang.String getName();

Code:
0: aload_0
1: getfield
4: areturn
Function Inlining

System.out.println("Hello, " + person.getName());

23: aload_1
24: invokevirtual #9 // Person.getName():()Ljava/lang/String;

public java.lang.String getName();
    Code:
    0: aload_0
    1: field
    4: areturn
Function Inlining

```java
public java.lang.String getName();
```

```java
public java.lang.String getName();
Code:
0: aload_0
1: getfield
4: areturn
```

```java
System.out.println("Hello, " + person.getName());
```

23: aload_1
24: invokevirtual #9 // Person.getName():()Ljava/lang/String;

```java
23: aload_1
24: invokevirtual #9 // Person.getName():()Ljava/lang/String;
```
Function Inlining

System.out.println("Hello, " + person.getName());

23: aload_1
24: invokevirtual #9

public java.lang.String getName();
Code:
  0: aload_0
  1: getfield
  4: areturn
Function Inlining

System.out.println("Hello, " + person.name);
Function Inlining

System.out.println("Hello, " + person.name);

23: aload_1
24: invokevirtual #9

23: aload_1
24: invokevirtual #9

public java.lang.String getName();
Code:
  0: aload_0
  1: getfield
  4: areturn
Function Inlining

```java
public java.lang.String getName();
```

```java
System.out.println("Hello, " + person.name);
```

```
23: aload_1
24: invokevirtual #9
```

```
23: aload_1
24: invokevirtual #9
```

```
public java.lang.String getName();
Code:
  0: aload_0
  1: getfield
  4: areturn
```
Benefits vs Costs

Minimizes call instructions

Code:
0: aload_0
1: getfield
4: areturn

public java.lang.String getName();

23: aload_1
24: invokevirtual #9
Benefits vs Costs

Minimizes call instructions
May increase Code size

23: aload_1
24: invokevirtual #9

public java.lang.String getName()
Code:
  0: aload_0
  1: invokevirtual #9
  23: aload_1
  24: invokevirtual #9

23: aload_1
24: invokevirtual #9

May increase
Code size
Greedy Inlining Strategy

- Focuses on direct benefits of inlining

```java
23: aload_1
24: invokevirtual #9

public java.lang.String
getName();

Code:
0: aload_0
1: getfield
4: a return
```
Greedy Inlining Strategy

- Focuses on direct benefits of inlining
- Inlines smallest methods first

```java
23: aload_1
24: invokevirtual #9

public java.lang.String
getName();

Code:
0: aload_0
1: getfield
4: a return
```
Greedy Inlining Strategy

- Focuses on direct benefits of inlining
- Inlines smallest methods first

Is inlining solved?
Greedy Inlining Strategy

- Focuses on direct benefits of inlining
- Inlines smallest methods first

Is inlining solved?
No!
Takeaway: small methods wrap around the computational expensive method

```java
public int startsWithAReturnsLength(String example) {
    boolean starts = example.startsWith("A");
    return starts.length();
}
```
Example

Takeaway: small methods wrap around the computational expensive method

```java
public int startsWithAReturnsLength(String example) {
    boolean starts = example.startsWith("A");
    return starts.length();
}

public boolean startsWith(String prefix) {
    return startsWith(prefix, 0);
}

public boolean regionMatches(int thisStart, String string, int start, int length) {
    /* ... */
}

public int length() {
    return this.length;
}
```
public int startsWithAReturnsLength(String example) {
    boolean starts = example.startsWith("A");
    return starts.length();
}

public boolean startsWith(String prefix) {
    return startsWith(prefix, 0);
}

public boolean startsWith(String prefix, int start) {
    return regionMatches(start, prefix, 0, prefix.count);
}

public int length() {
    return this.length;
}

Takeaway: small methods wrap around the computational expensive method
Example

Takeaway: small methods wrap around the computational expensive method

```java
public int startsWithAReturnsLength(String example) {
    boolean starts = example.startsWith("A");
    return starts.length();
}

public boolean startsWith(String prefix) {
    return startsWith(prefix, 0);
}

public boolean startsWith(String prefix, int start) {
    return regionMatches(start, prefix, 0, prefix.count);
}

public boolean regionMatches(int thisStart, String string, int start, int length) {
    /* ... */
}

public int length() {
    return this.length;
}
```
Greedy Inliner

Budget = 40
Remaining = 40

startsWithAReturnLength

startsWith1

length

startsWith2

regionMatches
Greedy Inliner

Budget = 40
Remaining = 36

startsWithAReturnLength

6
startsWith1

11
startsWith2

20
regionMatches

4 ✓
Greedy Inliner

Budget = 40
Remaining = 30

- startsWithAReturnLength
  - startsWith1
    - startsWith2
      - regionMatches
  - length
    - 4
  - 6

Greedy Inliner

Budget = 40
Remaining = 19

startsWithAReturnLength

startsWith1

length

startsWith2

regionMatches

6 ✓

11 ✓

4 ✓

20
Greedy Inliner

Budget = 40
Remaining = 19

startsWithAReturnLength

6 ✓

startsWith1

11 ✓

startsWith2

20 ×

regionMatches

4 ✓

length
Problem?

Greedy inliner → Minimizes call instruction overhead
Problem?

Greedy inliner

Minimizes call instruction overhead

What we want

Minimizes execution time

```java
23: aload_1
24: invokevirtual #9

public java.lang.String
getName();

Code:
0: aload_0
1: getfield
4: a return
```
Benefits vs Costs

Benefit

• Avoids call instruction overhead
Benefits vs Costs

Benefit

• Avoids call instruction overhead

• Improves dataflow analyses by providing additional context
Benefits vs Costs

Benefit

• Avoids call instruction overhead

• Improves dataflow analyses by providing additional context

• Other compiler transformations benefit from additional context
Benefits vs Costs

Benefit

- Avoids call instruction overhead
- Improves dataflow analyses by providing additional context
- Other compiler transformations benefit from additional context
- Inlined method specializes to its calling context
## Benefits vs Costs

<table>
<thead>
<tr>
<th>Benefit</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Avoids call instruction overhead</td>
<td></td>
</tr>
<tr>
<td>• Improves dataflow analyses by providing additional context</td>
<td></td>
</tr>
<tr>
<td>• Other compiler transformations benefit from additional context</td>
<td></td>
</tr>
<tr>
<td>• Inlined method specializes to its calling context</td>
<td></td>
</tr>
</tbody>
</table>
## Benefits vs Costs

<table>
<thead>
<tr>
<th>Benefit</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Avoids call instruction overhead</td>
<td>• Code growth</td>
</tr>
<tr>
<td>• Improves dataflow analyses by providing additional context</td>
<td></td>
</tr>
<tr>
<td>• Other compiler transformations benefit from additional context</td>
<td></td>
</tr>
<tr>
<td>• Inlined method specializes to its calling context</td>
<td></td>
</tr>
</tbody>
</table>
## Benefits vs Costs

<table>
<thead>
<tr>
<th>Benefit</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Avoids call instruction overhead</td>
<td>• Code growth</td>
</tr>
<tr>
<td>• Improves dataflow analyses by providing additional context</td>
<td>• Potential negative cache effects</td>
</tr>
<tr>
<td>• Other compiler transformations benefit from additional context</td>
<td></td>
</tr>
<tr>
<td>• Inlined method specializes to its calling context</td>
<td></td>
</tr>
</tbody>
</table>
Benefits vs Costs

**Benefit**

- Avoids call instruction overhead
- Improves dataflow analyses by providing additional context
- Other compiler transformations benefit from additional context
- Inlined method specializes to its calling context

**Costs**

- Code growth
- Potential negative cache effects
- Increase compile time and analysis time
Problem?

Greedy inliner → Minimizes call instruction overhead
Benefits

• Avoids call instruction overhead
Benefits

- Avoids call instruction overhead
- Improves dataflow analyses by providing additional context
- Other compiler transformations benefit from additional context
- Inlined method specializes to its calling context
Towards Better Inlining Decisions Using Inlining Trials

Jeffrey Dean and Craig Chambers
Department of Computer Science and Engineering
University of Washington

Abstract
Inlining trials are a general mechanism for making better automatic decisions about whether a routine is profitable to inline. Unlike standard source-level inlining heuristics, an inlining trial captures the effects of optimizations applied to the body of the inlined routine when calculating the costs and benefits of inlining. The results of inlining trials are stored in a persistent database to be reused when making future inlining decisions at similar call sites. Type group analysis can determine the amount of available static information exploited during compilation, and the results of analyzing the compilation of an inlined routine help decide when a future call site would lead to substantially the same generated code as a given inlining trial. We have implemented inlining trials and type group analysis in an optimizing compiler for SLEL, and by making wise inlining decisions we were able to cut compilation time and compiled code space with virtually no loss of execution speed. We believe that inlining trials and type group analysis could be applied effectively to many high-level languages where procedural or functional abstraction is used heavily.

1 Introduction
Inlining is an important implementation technique for reducing the performance costs of language abstraction mechanisms. Inlining (also known as procedure instantiation and unfolding) not only confers the direct benefit of eliminating the procedure call and return sequences but also facilitates optimizing the body of the called routine in the context of the caller site; sometimes these indirect potential benefits dwarf the direct benefits. Inlining has long been applied to languages like C and Fortran, but it may be even more beneficial in the context of higher-level languages. Functional languages such as Scheme and ML, (Rosen & Clinger 86, Milner et al. 90), pure object-oriented languages such as Smalltalk and Eiffel (Goldberg & Robson 83, Meyer 92), and reflective systems such as CLU55 and SchemeXerox (Brotherhood et al. 88, Adams et al. 93) encourage programmers to write general, reusable routines and solve problems by composing existing functions, leading to programs with very high call frequencies. Compilers and partial evaluators, such as Simula and Scheme (Bosshard 91, Corner 93), can exploit inlining to reduce the cost of these abstraction mechanisms and thereby foster better programming styles.

Inlining is possible only when the compiler can determine statically the single target routine invoked by a call, in functional and object-oriented languages, this determination can require sophisticated analysis (Shavers 88, Hall & Kennedy 92, Chambers & Ungur 90, Paulson & Schwartzbach 91). But even if the call site is potentially inlinable, inlining may not be profitable. Care must be taken not to inline too much, or compilation time and compiled code could swell prohibitively. Inlining should only be applied where the benefits obtained by inlining outweigh the costs.

In many systems, the profitability of inlining a particular routine is hard-wired into the compiler. For example, the Smalltalk-80 compiler hard-wires the definition and optimized implementation of several basic functions from its standard library, and the Haskell standard prelude is fixed so that compilers can implement the functions in the standard library more efficiently (Hofstad et al. 90). A drawback of the hard-wiring approach is that built-in routines usually run much faster than user-defined routines, discouraging programmers from defining and using their own abstractions. Other systems, including C++, Modula-2, T-Scheme, SchemaXerox, Common Lisp, Simula, and Scheme (Streeter 91, Nelson 91, Slade 87, Adams et al. 93, Steele 90), allow programmers to indicate explicitly which routines are profitable to inline. While granting programmers fine control over the compilation process, this approach requires programmers to have a fair understanding of the language's implementation issues (an assumption becoming less likely as implementations become more sophisticated) and can be tedious if inlining must be applied heavily to get good performance. Additionally, most explicit declaration-based mechanisms do not allow programmers to specify that inlining is profitable only in certain contexts, or that inlining should only take place at particular high-frequency calls of some routine.

Our research investigates techniques for automatically deciding when inlining is profitable. Making good inlining decisions depends crucially on accurately assessing the costs and benefits of inlining. Previous automatic decision makers used simple techniques for estimating costs based on an examination of the target routine's source code (or unoptimized intermediate code), and consequently they failed to take into account the effect of post-inlining optimization of the target routine. Our work corrects this deficiency, leading to more accurate cost and benefit estimates and therefore better inlining decisions.

Our system assesses the costs and benefits of inlining by first experimentally inlining the target routine, in the process measuring the actual costs and benefits of that particular inlining. Then it optimizes the cost of the experiment (called an inlining trial) across future calls to that routine by storing the results of the trial in a persistent database. Because the indirect costs and benefits of inlining can depend greatly on the amount of the static information available at the call site (e.g., the static value or class of an argument, our system performs type group analysis to determine the amount of available call-site-specific static information that was exploited during optimization. Each database entry is guarded with a type group information, restricting reuse of the information derived from an inlining trial to those call sites that would generate substantially the same compiled code.

We implemented and measured our approach in the context of an optimizing compiler for SLEL (Unger & Smith 87, Chambers & Ungur 91), a pure object-oriented language similar to Smalltalk but without any hard-wired operations or control structures. The SLEL
Problem?

Greedy inliner

What we want

Cost-Benefit Inliner

Minimizes call instruction overhead

Minimizes execution time

Maximizes Benefit
Steps

1. Call Graph Construction
2. Transfer static information
3. Calculate the benefit metric
4. Obtain inlining plan
Call Graph Construction

```
startsWithAReturnLength
  └── startsWith1
      └── startsWith2
          └── regionMatches
```

```
public int startsWithAReturnsLength(String example) {
    boolean starts = example.startsWith("A");
    return starts.length();
}
public boolean startsWith(String prefix) {
    return startsWith(prefix, 0);
}
public boolean startsWith(String prefix, int start) {
    return regionMatches(start, prefix, 0, prefix.length);
}
public boolean regionMatches(int thisStart,
    String string,
    int start,
    int length) {

    if (start < 0 || string.count - start < length) return false;
    if (thisStart < 0 || count - thisStart < length) return false;
    if (length <= 0) return true;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}
Post Inlining Benefit

```java
public boolean regionMatches(int thisStart,
   String string,
   int start,
   int length) {

    if (start < 0 || string.length - start < length) return false;
    if (thisStart < 0 || string.length - thisStart < length) return false;
    if (length <= 0) return true;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}
```
public boolean regionMatches(int thisStart,
    String string,
    int start,
    int length) {

    if (string.count - start < length) return false;
    if (thisStart < 0 || count - thisStart < length) return false;
    if (length <= 0) return true;
    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }
    return true;
}
public boolean regionMatches(int thisStart,
    String string,
    int start,
    int length) {

    if (string.count - start < length) return false;
    if (thisStart < 0 || count - thisStart < length) return false;
    if (length <= 0) return true;

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}
public boolean regionMatches(int thisStart,  
    String string,  
    int start,  
    int length) {

    if (start < 0 || string.count - start < length) return false;  
    if (thisStart < 0 || count - thisStart < length) return false;  
    if (length <= 0) return true;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])  
            return false;
    }

    return true;
}
public boolean regionMatches(int thisStart, 
        String string, 
        int start, 
        int length) {

    if (thisStart < 0 || count - thisStart < length) return false;
    if (length <= 0) return true;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}
public boolean regionMatches(int thisStart,
       String string,
       int start,
       int length) {

    if (thisStart < 0 || count - thisStart < length) return false;
    if (length <= 0) return true;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}
public boolean regionMatches(int thisStart, 
    String string, 
    int start, 
    int length) {

    if (count - thisStart < length) return false;
    if (length <= 0) return true;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}
public boolean regionMatches(int thisStart,
    String string,
    int start,
    int length) {

    if (count < 1) return false;
    if (length <= 0) return true;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }
    return true;
}
public boolean regionMatches(int thisStart,
String string,
int start,
int length) {

if (count < 1) return false;
if (length <= 0) return true;

/* ... */

int end = length - 1;
for (int i = 0; i < end; ++i) {
    if (source[o1 + i] != target[o2 + i])
        return false;
}
return true;
}
public boolean regionMatches(int thisStart,
   String string,
   int start,
   int length) {

    if (count < 1) return false;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}
public boolean regionMatches(int thisStart,
    String string,
    int start,
    int length) {

    if (count < 1) return false;

    /* ... */
    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}

Benefit in method = 0.5

*Still work in progress
If end was different to 0, then we could unroll the loop.
public boolean regionMatches(int thisStart,
    String string,
    int start,
    int length) {

    if (start < 0 || string.count - start < length) return false;
    if (thisStart < 0 || count - thisStart < length) return false;
    if (length <= 0) return true;

    /* ... */

    int end = length - 1;
    for (int i = 0; i < end; ++i) {
        if (source[o1 + i] != target[o2 + i])
            return false;
    }

    return true;
}

If end was different to 0, then we could unroll the loop.
Benefit Analysis

Budget = 40

Cost/Benefit
Benefit Analysis

Budget = 40
Remaining = 36

Cost/Benefit
Benefit Analysis

Budget = 40
Remaining = 34

(startsWithLength) →

(startsWith1)  length  (4/1)

(startsWith2) →

(regionMatches) (20/5)

Cost/Benefit
Benefit Analysis

Budget = 40
Remaining = 30

startsWithAReturnLength

- startsWith1
- length

- startsWith2
- regionMatches

Cost/Benefit
Benefit Analysis

Budget = 40
Remaining = 23

Cost/Benefit
Benefit Analysis

Budget = 40
Remaining = 19

Cost/Benefit

startsWithAReturnLength

 startsWith1
 startsWith2
 regionMatches

6/1
11/1
20/5

4/1

✓
✓
✓
Benefit Analysis

Budget = 40
Remaining = 3

✓ 6/1

✓ 11/1

✓ 20/5

startsWithA\rightarrow length

startsWith1

startsWith2

regionMatches

Cost/Benefit
Comparison

Greedy inlining plan

- startsWithAReturnLength
  - startsWith1
  - length
  - startsWith2

Cost benefit analysis inlining plan

- startsWithAReturnLength
  - startsWith1
  - startsWith2
  - regionMatches
Some Challenges

- Virtual functions (do not consider all targets, prioritize heavily called targets.)
Some Challenges

• Virtual functions (do not consider all targets, prioritize heavily called targets.)

• Limiting big search space (profiling information + updating IDT while doing analysis)
Some Challenges

- Virtual functions (do not consider all targets, prioritize heavily called targets.)

- Limiting big search space (profiling information + updating IDT while doing analysis)

- Determining adequate trade-off between compile-time and analysis-time (approximate analysis)
Some Challenges

- Virtual functions (do not consider all targets, prioritize heavily called targets.)

- Limiting big search space (profiling information + updating IDT while doing analysis)

- Determining adequate trade-off between compile-time and analysis-time (approximate analysis)

- What other optimizations to consider? (escape analysis)
Benefits vs Costs

23: aload_1
24: instanceof

public java.lang.String getName():
  Code:
  1:  aload_0
  2:  getfield
  3:  invokevirtual

Minimizes call instructions
Benefits vs Costs

Minimizes call instructions

Cost-Benefit Inliner

Maximizes Benefit

```
23: aload_1
24: getfield

public java.lang.String getName()
Code:
0: aload_0
1: getfield
2: invokevirtual

startsWithAReturnLength

startsWith1

length

startsWith2

regionMatches
```
Benefits vs Costs

length = 1
start = 0
string = “A”
thisStart = 0
this = Type: String

Cost-Benefit Inliner → Maximizes Benefit

Minimizes call instructions

23: aload_1
24: getfield

startsWithA

startsWith1

length

startsWith2

regionMatches

public java.lang.String getName()

Code:
0: aload_0
1: getfield
2: getfield
length = 1
start = 0
string = “A”
thisStart = 0
this = Type: String
length = 1

start = 0

string = “A”

thisStart = 0

this = Type: String