Link-Time Static Analysis for Efficient Separate Compilation of Object-Oriented Languages

Jean Privat Roland Ducournau

LIRMM
CNRS/Université Montpellier II
France

Program Analysis for Software Tools and Engineering
Lisbon 2005

Outline

1. **Motivation**
2. **Global Techniques**
   - Type Analysis
   - Coloring
   - Binary Tree Dispatch
3. **Separate Compilation**
4. **Benchmarks**
   - Description
   - Results

Outline

1. **Motivation**
2. Global Techniques
    - Type Analysis
    - Coloring
    - Binary Tree Dispatch
3. Separate Compilation
4. Benchmarks
    - Description
    - Results

Software Engineering Ideal

Production of Modular Software

- Extensible software
- Reusable software components

⇒ Object-Oriented Programming (*inheritance* + *late binding*)

Production of Software in a Modular Way

- Small code modification → small recompilation
- Shared software components are compiled only once
- Software components can be distributed in a compiled form

⇒ Separate Compilation (*compile* components + *link*)

Compilation of OO Programs

Global Techniques

Knowledge of the whole program → more efficient implementation:

- Method invocation
- Access to attribute
- Subtyping test

The Problem

- Previous works use global technique with global compilation
- Global compilation is incompatible with modular production

Our Proposition

A Compromise

- A separate compilation framework
- that includes 3 global compilation techniques

How To?

⇒ Perform global techniques at link-time

Outline

1 Motivation

2 **Global Techniques**
- Type Analysis
- Coloring
- Binary Tree Dispatch

3 Separate Compilation

4 Benchmarks
- Description
- Results

Type Analysis

Problems

- Most method invocations are actually monomorphic
  → Implement them with a static direct call (no late binding)
- Many methods are dead
  → Remove them

How to?

Approximate 3 sets:

- Live classes and methods
- Concrete type of each expression
- Called methods of each call site

Many type analysis exist

Coloring

Problem

Overhead with standard VFT in multiple inheritance:

- Subobjects
- Many VFT (quadratic number, cubic size)

Solution

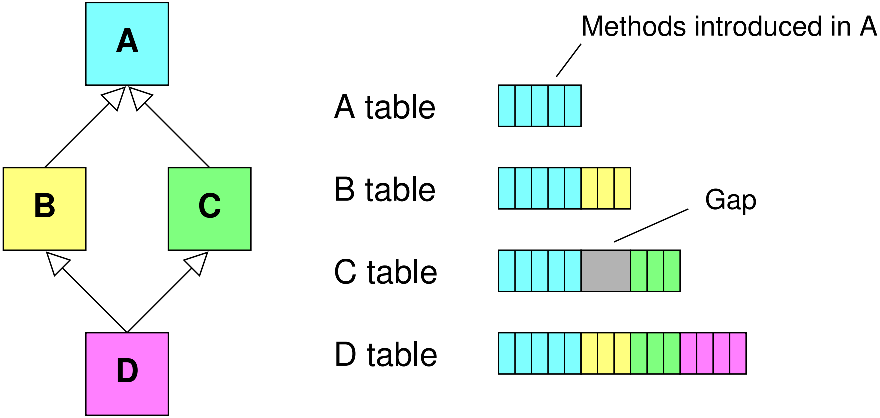
→ Simple inheritance implementation even in multiple inheritance

How to?

- Assign an identifier by class
- Assign a color (index) by class, method and attribute
- Minimize size of the tables

A NP-hard problem

Coloring (example)



Binary Tree Dispatch

Problem

Prediction of conditional branching of modern processors does not work with VFT

Solution

→ Use static jumps instead of VFT

How to?

- Perform a type analysis
- Assign an identifier by live class
- For each live call site, enumerate concrete type in a select tree

Binary Tree Dispatch (Example)

Compiling call site `x.foo`

- `id` is the class itentifier of the receiver `x`
- Concrete type of `x` is {A, B, C}

Class	A	B	C
Identifier	19	12	15
`foo` implementation	`A_foo`	`B_foo`	`C_foo`

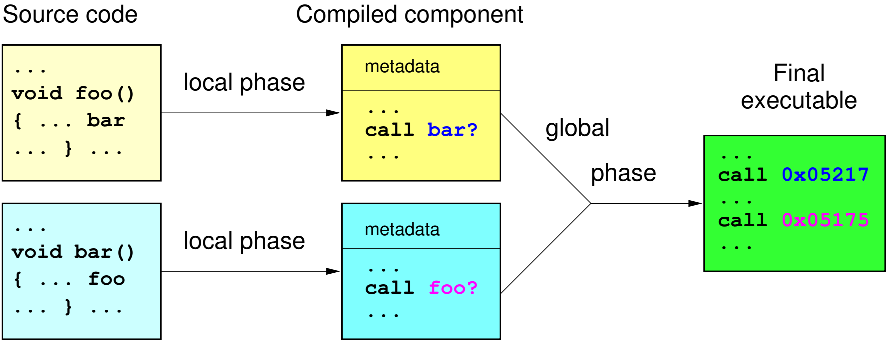
Generated Code

```
if id <= 15 then
        if id <= 12 then call B_foo
        else call C_foo
else call A_foo
```

Outline

1. Motivation
2. Global Techniques
    - Type Analysis
    - Coloring
    - Binary Tree Dispatch
3. **Separate Compilation**
4. Benchmarks
    - Description
    - Results

Separate Compilation



Source code

```
...
void foo()
{ ... bar
... } ...
```

```
...
void bar()
{ ... foo
... } ...
```

local phase

local phase

Compiled component

metadata

```
...
call bar?
...
```

metadata

```
...
call foo?
...
```

global

phase

Final executable

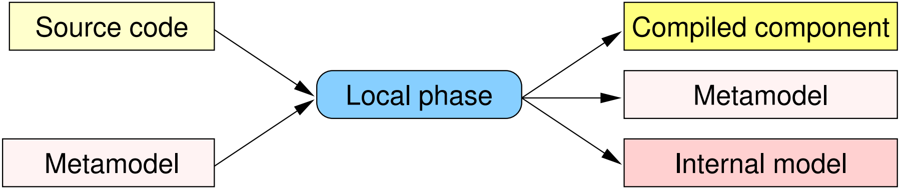
```
...
call 0x05217
...
call 0x05175
...
```

Two Phases

Local phase compiles independently of future use

Global phase links compiled components

Local Phase



Input

- Source code of a class
- Metamodel of required classes

Outputs

- Compiled version of the class (with unresolved symbols)
- Metadata : metamodel, internal model

Compiled Component

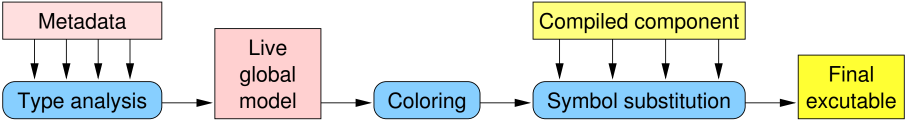
Method Call Site

- Assign a unique symbol by call site
- Compile into a direct call

Attribute Access and Subtype Test

- Assign a unique symbol by color and identifier
- Compile into a direct access:
    - in the instance for attribute access
    - in the subtyping table for subtype tests

Global phase



3 Stages

- Type analysis: based on the metadata
- Coloring: computes colors
- Symbol substitution: generates the final executable

Method Call Site Symbols

Substitute the address of:

- monomorphic → the invoked method
- polymorphic w/ BTD → a generated select tree
- polymorphic w/ VFT → a generated table access

Outline

1. Motivation
2. Global Techniques
    - Type Analysis
    - Coloring
    - Binary Tree Dispatch
3. Separate Compilation
4. **Benchmarks**
    - Description
    - Results

Benchmarks Description

Language and Compilers

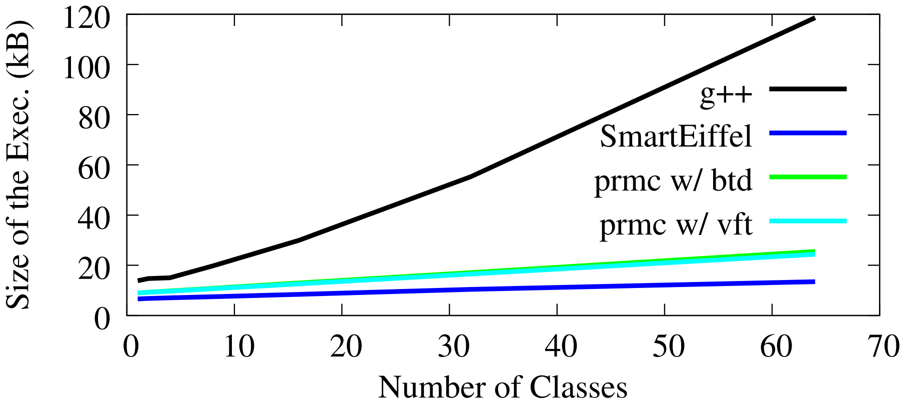
- g++: Separate + VFT w/ subobjects
- SmartEiffel: Global + Binary Tree Dispatch
- prmc w/ VFT: Separate + Coloring + VFT
- prmc w/ BTD: Separate + Coloring + BTD

Programs

Small programs are generated by a script

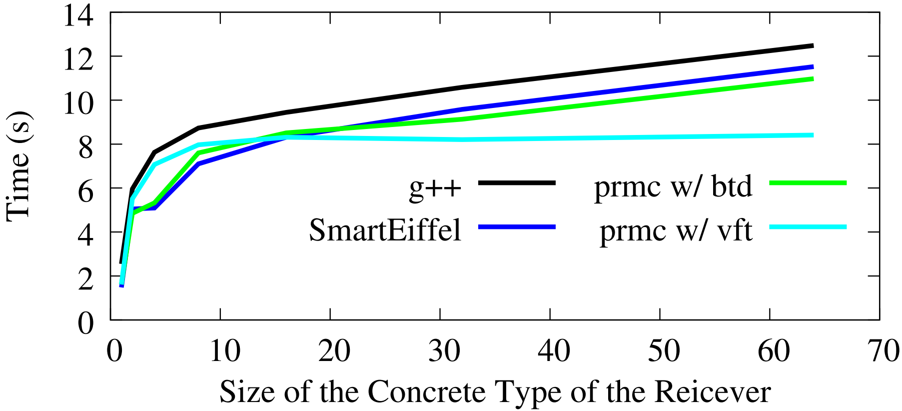
- The same programs for all language
- 1 OO mechanism per program

Size of Executables



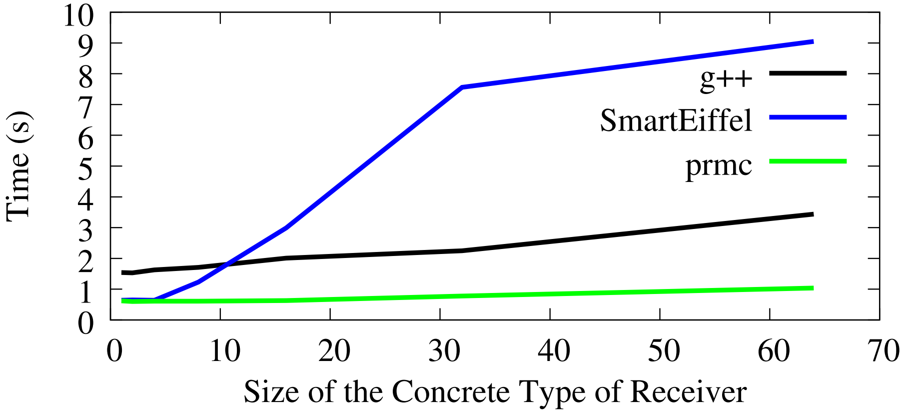
- Subobjects: many VTF → an important overhead
- prmc: BTD $\simeq$ VFT
- SmartEiffel: better dead code removal

Method Invocation



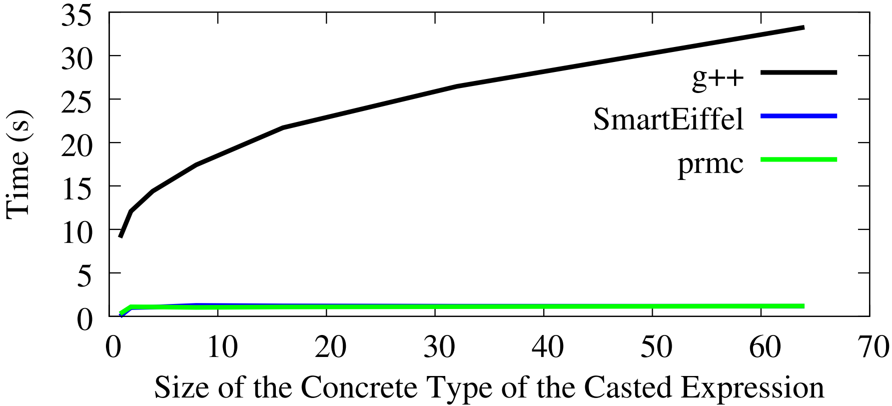
- Subobjects: constant overhead + cache misses
- BTD: better on oligomorphic calls
- Coloring: better on megamorphic calls

Attribute Access



- Subobjects: constant overhead
- Coloring: constant attribute access
- SmartEiffel: can degenerate

Subtype Test



- g++: bad performances
- Coloring and BTD: equivalent and mainly constant

Summary

Summary

A separate compilation framework with global techniques for statically typed class-based languages

- Better modularity than global compilers
- Better performance than other separate compilers

Outlook

- Shared libraries linked at load-time or dynamically loaded
- Time overhead of the global phase (link)