Efficient Separate Compilation of Object-Oriented Languages

Jean Privat Floréal Morandat Roland Ducournau

LIRMM
CNRS/Université Montpellier II
France

Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems
ECOOP 2006
Nantes
Outline

1 Motivation

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

3 Separate Compilation
   - Local Phase
   - Global Phase
Outline

1 Motivation

2 Global Techniques
   • Type Analysis
   • Coloring
   • Binary Tree Dispatch

3 Separate Compilation
   • Local Phase
   • Global Phase
Ideal Software Engineering

Production of Modular Software

- Extensible software
- Reusable software components

⇒ Object-Oriented Programming \((\text{inheritance} + \text{late binding})\)

Production of Software in a Modular Way

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

⇒ Separate Compilation \((\text{compile components} + \text{link})\)
Compilation of OO Programs

Global Techniques
Knowledge of the whole program → more efficient implementation:
- Method invocation
- Access to attributes
- Subtyping test

The Problem
- Previous works use global technique with global compilation
- Global compilation is incompatible with modular production
Our Proposal

A Compromise

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

How?

⇒ Perform global techniques at link-time
Outline

1 Motivation

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

3 Separate Compilation
   - Local Phase
   - Global Phase
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?

Approximate 3 sets:

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

Many type analyses exist
Coloring

Problem

Overhead with standard VFT in multiple inheritance (e.g. C++):

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

Solution

→ Simple inheritance implementation even in multiple inheritance

How?

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

A NP-hard problem
Binary Tree Dispatch

**Problem**

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

**Solution**

→ Use static jumps instead of VFT

**How?**

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

1 Motivation

2 Global Techniques
   • Type Analysis
   • Coloring
   • Binary Tree Dispatch

3 Separate Compilation
   • Local Phase
   • Global Phase
Separate Compilation

Two Phases

Local phase: compiles independently from future use
Global phase: links compiled components
Local Phase

**Input**
- Source code of a class
- External model of required classes

**Output**
- Compiled version of the class (with unresolved symbols)
- Metadata: external model, 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

Replace symbols with the address of:

- monomorphmic → the invoked method
- oligomorphmic → a generated select tree (BTD)
- megamorphmic → a generated table access (VFT)
Complete Compilation Process

A Source code → Local phase → Compiled component → Type analysis
A External model → Live global model
A Internal model → Coloring

B Source code → Local phase → Compiled component → Symbol substitution
B External model → Final executable
B Internal model
A Simple Benchmark

![Graph showing the time (s) on the y-axis and the size of the concrete type of the receiver on the x-axis. The graph compares the performance of g++, SmartEiffel, and prmc.]
Summary

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

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

Future Work

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