Fast, Effective Analysis and Optimization of C++ Programs

David F. Bacon and Peter F. Sweeney

IBM T.J. Watson Research Center
IBM C++ Optimization Group

- Michael Burke
- Michael Hind
- G. Ramalingam
- Harini Srinivasan
Introduction

- C++ has powerful features
  - Virtual Function Calls
  - Virtual Base Classes
  - Dynamic Casts

- But they cost
  - Time
  - Space
Optimization

- **Analysis**
  - ask the right questions

- **Transformation**
  - use the answers

- **Evaluation**
  - how well does it work?
Analysis Framework

- What classes are created?

- shape
  - triangle
  - rectangle
    - square
Analysis Framework

- What classes are created?

```
shape

triangle  rectangle

  square
```
Analysis Framework

- What are the classes of each value?

  object->draw();  { triangle }

  sign.reshape();  { rectangle, square }
Class Hierarchy Analysis

- What classes are created?
  - all classes

- What are the classes of each value?
  - the classes derived from the declared type
Class Hierarchy Analysis

- the classes derived from the declared type

```
shape
  triangle
  rectangle
  square

rectangle* r;
{rectangle,square}

square * s;
{square}
```
Rapid Type Analysis

- What classes are created?
  - the classes in the call graph

- What are the classes of each value?
  - the classes derived from the declared type
  - that have been created
Rapid Type Analysis

- the classes in the call graph

```cpp
new square;

build

class main

class rectangle

class square
```
Rapid Type Analysis

- the classes in the call graph: \{\texttt{square}\}

```cpp
def main()
    new square;
    build(square::draw);
```
What are the classes of each value?
- the classes derived from the declared type
- that have been created

rectangle* r;
{square}

square * s;
{square}
Transformation

- virtual base classes to non-virtual bases
- dynamic casts to static casts
- virtual calls to direct calls
Virtual Bases

- Is `shape` multiply inherited?
- Is that class live?
- If not, make `shape` a non-virtual base
Dynamic Casts

```cpp
foo(shape* p) {dynamic_cast<square> p;}
```

- What object types could `p` be?
- Is `square` the only possible type?
- If so, do a compile-time cast
Virtual Calls

\[ p->\text{draw}(); \]

- What object types could \( p \) be?
- What \texttt{draw()} functions are defined?
- If only 1 \texttt{draw()} function
  - resolve the call
Benefits of VFR

- **Time**
  - direct calls are faster
  - in-lining is possible

- **Space**
  - unused functions can be eliminated

- **Understandability**
  - programmer doesn’t see unused code
Evaluation

- Compare 3 analysis algorithms
  - Rapid Type Analysis
  - Class Hierarchy Analysis
  - Unique Name

- On virtual function resolution
Methodology

- Use real programs as benchmarks
  - 7 large programs 5000-20000 lines each
- Determine causes of
  - success
  - failure
- Evaluate against best possible algorithm
Dynamic Call Types

The chart illustrates the distribution of dynamic call types across various programs. Each bar is divided into segments representing different call types:

- Direct Method
- Indirect Function
- Direct Function

The programs listed from left to right are: sched, lxx, lcom, hotwire, simulate, idl, taldict, deltable, and richards.
Why RTA Wins: **ixx**

- RTA finds unused classes
- String ops are in inner loops
- Similar win for
  - taldict
  - hotwire
Why RTA Loses: sched

- True Polymorphism
  shape → □ → ○ → △

- Disjointed Polymorphism
  shape → □ → □ → □
  shape2 → △ → △ → △
Reduction in Code Size

The diagram shows the reduction in code size for several programs: sched, ixx, lcom, hotwire, simulate, idl, taldict, deltablue, and richards. The reduction is measured in percentages: 0%, 20%, 40%, 60%, 80%, and 100%. The graph uses different colors to represent different categories of code:

- **Live** (clear bars)
- **Not Eliminated/Unexecuted** (light red bars)
- **Eliminated by RTA** (dark blue bars)
- **Eliminated by CHA** (light blue bars)

Each program is represented by a set of bars, with the height of the bars indicating the percentage of code reduction in each category.
### Speed of Analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size (lines)</th>
<th>Analysis Time (s)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched</td>
<td>5,712</td>
<td>1.94</td>
<td>0.1%</td>
</tr>
<tr>
<td>ixx</td>
<td>11,157</td>
<td>5.22</td>
<td>1.4%</td>
</tr>
<tr>
<td>lcom</td>
<td>17,278</td>
<td>6.50</td>
<td>3.0%</td>
</tr>
<tr>
<td>hotwire</td>
<td>5,335</td>
<td>2.06</td>
<td>1.3%</td>
</tr>
<tr>
<td>simulate</td>
<td>6,672</td>
<td>2.75</td>
<td>5.6%</td>
</tr>
<tr>
<td>idl</td>
<td>30,288</td>
<td>6.42</td>
<td>1.4%</td>
</tr>
<tr>
<td>taldict</td>
<td>11,854</td>
<td>1.78</td>
<td>4.0%</td>
</tr>
<tr>
<td>deltablue</td>
<td>1,250</td>
<td>0.44</td>
<td>2.4%</td>
</tr>
<tr>
<td>richards</td>
<td>606</td>
<td>0.32</td>
<td>3.6%</td>
</tr>
</tbody>
</table>
Comparison: Alias Analysis

- Best precision from a static analysis
- Complex algorithm--expensive to implement
- Slow
  - between 0.4 and 55 source lines analyzed/second
  - RTA is 45 to 8250 times faster
Comparison: Alias Analysis

Program

- 9:deriv1
- 7:deriv2
- 3:family
- 5:garage
- 10:objects
- 1:office
- 12:primes
- 8:shapes
- 6:vcircle

Legend:
- Unresolved/Polymorphic
- Unresolved/Not Executed
- Unresolved/Monomorphic
- Resolved by RTA
- Resolved by CHA
Contributions

- Analysis Framework
- Rapid Type Analysis (RTA) algorithm
- Evaluation showing power of RTA
Rapid Type Analysis

- RTA is effective:
  - resolves 71% of virtual function calls
  - reduces code size by 25%
- RTA is fast:
  - analyzes 3300 lines per second
- RTA is often as good as alias analysis