Future Challenges in Dynamic Interprocedural Analysis and Optimization

Vivek Sarkar
IBM T.J. Watson Research Center
vsarkar@us.ibm.com
Outline

1. Motivation
2. Dynamic Optimistic Interprocedural Type Analysis (DOIT)
3. Immutability Analysis Opportunities for Dynamic IPA
4. Future Challenges
Acknowledgments / References

- Discussions with Jikes RVM team members on interprocedural extensions to type analysis, load/store elimination, and register allocation
Motivation

- Interprocedural analysis (IPA) is essential for compiler-driven performance
  - especially when optimizing object-oriented languages
- Static IPA optimizations:
  - limited precision due to impact of methods that may not be executed
  - scalability limitations in analyzing static “whole program”
- Dynamic intra-procedural optimizations:
  - Significant advances, with inlining, to address interprocedural optimization opportunities
  - reaching point of diminishing returns
- Dynamic IPA:
  - Opportunity to get best of both worlds
Dynamic Interprocedural Analysis Scenario

- supports dynamic class loading, adaptive optimization, optimistic assumptions about unanalyzed code
Invalidation Scenario

- Support for invalidation is necessary, to handle case when optimistic assumption proves to be incorrect

\[ M_k \text{ (re)} \]

Use IPA information, making optimistic assumptions

Method \( M_k \) compiled with optimization

New classes get loaded

First execution of new method \( M_x \) (unoptimized)

Analyze \( M_x; M_x \) violates assumptions for \( M_k \)

Invalidate optimized compilation of \( M_k \)
Static vs. Dynamic Application Characteristic:
(Number of Methods)

Ratio of Dynamic methods to Semi-static methods ~ 12% - 50%
Static vs. Dynamic Application Characteristics:
(Number of Fields containing object references)

Ratio of Dynamic fields to Semi-static fields ~ 31% - 92%
Outline

1. Motivation
2. *Dynamic Optimistic Interprocedural Type Analysis (DOIT)*
3. Immutability Analysis Opportunities for Dynamic IPA
4. Future Challenges
DOIT Phases

- Initialization
- Analysis
  - analyzes each method on first invocation
  - incorporates method summary into Value Graph
- Optimization
  - traverses Value Graph to identify types
  - uses type information in optimization
  - registers verification actions for type info used
  - registers invalidations for optimized method
Value Graph

- **Node** \( n \) denotes a set of types, \((n)\)
- **Location nodes**
  - Local variable
  - Field
  - Array element
  - Constant type e.g., \( T_1 \)
- **Edges** represent flow of types
  - graph may be cyclic

- **Operator nodes**
  - Closure: \((\ast)\)
  - Subscript: \( ([]) \)
  - Union: \((\bullet)\)
  - Intersection: \((\ mundane)\)
Local Value Graph Example

\[ T_1 \ M(); \]
\[ \ldots \]
\[ S1: \ k = A.a; \]
\[ \ldots \]
\[ S2: \ k = M(); \]
\[ \ldots \]
\[ S3: \ A.a[0] = k; \]

\[ B1.1: \ \text{getstatic} \ A.a \]
\[ B1.2: \ \text{astore} \ k \]
\[ \ldots \]
\[ B2.1: \ \text{invokestatic} \ M \]
\[ B2.2: \ \text{astore} \ k \]
\[ \ldots \]
\[ B3.1: \ \text{getstatic} \ A.a \]
\[ B3.2: \ \text{iconst}_0 \]
\[ B3.3: \ \text{aload} \ k \]
\[ B3.4: \ \text{aastore} \]
Computing Local and Global Value Graphs

- **Local Value Graph**
  - Abstract interpretation of bytecodes
    - propagates types symbolically through stack
  - Represents type flow in method

- **Global Value Graph**
  - Local Value Graph is compressed after method is analyzed
    - Local variable nodes can be bypassed and removed
  - Local Value Graph is spliced into Global Value Graph
Global Value Graph

Global Type Constants

$T_1$

Object[]

$T_1[]$

$T_0[]$

Method $M_1$

$k$

$[]$

Method $M_2$

$l$

Method Summaries

Global Locations

(A.a)

A.a
Computing Type Information

- For use in optimization
- Determine the type of a given location
  - on-demand traversal of the Value Graph
    - reverse-DFS starting at location
    - types are propagated along the edges
Experimental Setup

- Prototyped using Jikes RVM
  - type-based optimizations of calls
  - recompile after first run at highest opt level
- Benchmarks:
  - SPECjvm98, Hyper/J, Xerces (DOMCount)
- Measurements
  - Dynamic counts of virtual and interface calls
  - Execution times
  - Value Graph sizes and traversal statistics
  - Value Graph construction times
Experimental Results

Impact of DOIT Analysis on Interface Calls

% of all interface invocations

Virtualized and unguarded
Virtualized and guarded
Static guarded inline
Virtualized
Interface dispatch

Pessimistic (CHA)
Optimistic (DOIT)
Experimental Results

Speedup from using interprocedural type info

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>1.6%</td>
</tr>
<tr>
<td>jess</td>
<td>3.6%</td>
</tr>
<tr>
<td>gb</td>
<td>7.5%</td>
</tr>
<tr>
<td>javac</td>
<td>0.8%</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>0.3%</td>
</tr>
<tr>
<td>mtrt</td>
<td>0.5%</td>
</tr>
<tr>
<td>jack</td>
<td>2.4%</td>
</tr>
<tr>
<td>HyperJ</td>
<td>3.4%</td>
</tr>
<tr>
<td>DOMCount</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>3.1%</td>
</tr>
</tbody>
</table>
Experimental Results
Value Graph Traversal Statistics

Average nodes/edges visited

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack
- HyperJ
- DOMCount

Compress
Jess
Db
Javac
Mpegaudio
Mtrt
Jack
HyperJ
Domcount
Experimental Results
Analysis Rates (bytecode bytes/ms)

Average Analysis Rate (bcb/ms)

Baseline Compiler
DOIT Analysis

Analysis rate (bcb/ms)

2.53× 4.49× 2.81× 4.87× 2.80× 5.40× 4.20× 3.53×

compress jess db javac mpegaudio mtrt jack Average
Outline

1. Motivation
2. Dynamic Optimistic Interprocedural Type Analysis (DOIT)
3. Immutability Analysis Opportunities for Dynamic IPA
4. Future Challenges
Immutability Analysis: Motivation

- Immutability information can be used interprocedurally to enhance:
  - Load elimination and register allocation
    - Load of immutable value cannot be changed across a procedure call
  - Array dependence analysis, pointer alias analysis
    - Target of a store instruction cannot be aliased with target of a load instruction
  - Value Numbering / CSE / PRE
    - Load of an immutable value can be treated similarly to read of an unmodified local variable to enable optimization of derived expressions (including null pointer, type checks, array bounds checks)
  - Data transformations
    - object inlining, splitting, replication, caching
  - Parallelization
    - Immutable locations cannot interfere with parallelization
Immutability Properties

- Dimensions of Immutability:
  - Lifetime
    - *e.g.*, whole program, after a certain point, in method call
  - Reachability
    - *e.g.*, reference, object, full reachability, arbitrary shape
  - Context
    - *e.g.*, all instances, instances within a method, *etc.*

- Existing language mechanisms provide limited support for these dimensions
  - *e.g.*, Java `final`, C++ `const`

- How can immutability properties be obtained?
  1. Specified by user
  2. Inferred (optimistically) by dynamic optimization system
     
     Opportunity for Dynamic IPA
Dimensions: Lifetime

- whole program
- after a certain program point
  - *e.g.*, after an object has been initialized
- in a method call
- *etc.*
Dimensions: Reachability

- reference (=final)
- object
- full reachability
- arbitrary shape
Simple Example

class MyString {
    /* assume deep immutability for S*/
    final char[] S;
    final int count;

    int foo() {
        int c1 = S[0];
        bar();
        int c2 = S[0]; // c2 must be same as c1
        return c1 + c2;
    }
}

Limit Study: Immutability Ratio

- Define *Immutability Ratio* as
  \[
  IR = \frac{\text{# of read operations after last write}}{\text{total # of read operations}}
  \]

- **IR actual**
  - Obtained by counting last write separately for each dynamic object instance

- **IR uniform**
  - Obtained by assuming that writes are uniformly distributed among reads
    - Hypothetical “expected” value of IR
Limit Study: Experimental Setup

• Instrument Jikes RVM to generate traces
  • all read and write accesses
• Benchmarks
  • Jikes RVM regression tests
    • bytecodeTests, reflect, threads, utf8, opttests
  • CaffeineMark
  • SPECjvm98 (input size = 10%)
    • _200_check, _202_jess, _209_db, _213_javac
  • Xerces (DomCount)
• Goal: measure Immutability Ratio for benchmarks
Limit Study: Abstract Locations

- Abstract location = static representative for set of dynamic locations
  - Each declared field is a distinct abstract location
  - Each declared array type is a distinct abstract location
- Coarse-grained immutability: measured by merging all dynamic instances of the same abstract location
- Goals:
  - Measure gap between fine-grained and coarse-grained immutability
  - Determine how immutable reads are distributed across abstract locations
Distribution of immutable reads across abstract locations: _202_jess

Fraction of abstract locations

Number of Immutable reads

Fine-grained Immutability
Coarse-grained Immutability
Distribution of immutable reads across abstract locations: _209_db
Distribution of immutable reads across abstract locations: _213_javac

- Fine-grained Immutability
- Coarse-grained Immutability
Distribution of immutable reads across abstract locations: DOMcount

![Graph showing the distribution of immutable reads across fraction of abstract locations]

- **Fine-grained Immutability**
- **Coarse-grained Immutability**
Invalidation Issues in Dynamic IPA

- **Correctness**: must always be possible to undo the optimization
  - need recovery procedure; may limit scope of optimization
- **Efficiency**: cost; depends on
  - what optimization is performed, *e.g.*,
    - preexistence based inlining only needs recompilation
    - dead store elimination needs on-stack replacement
    - object inlining needs data structure rewriting
  - when optimization is performed
    - delaying optimization may avoid need for invalidation
Integrating Dynamic IPA into Adaptive Optimization Framework

- Invalidation cost supplied to adaptive system
  - which uses cost-benefit model
- Optimization considered worthwhile if cost of invalidation less than potential benefit
  - invalidation cost may vary dynamically
- Optimizations may be more profitable for long-running programs
Adaptive Optimization System w/ Adaptive Inlining

- Method Samples
- Cell Edge Samples
- Hot Method Organizer
- Decay Organizer
- Collector Organizer
- Inlining Organizer
- Dynamic Call Graph
- Inlining Rules
- Method, %Hot, Boost Factor
- Event Queue
- Controller
- Compilation Queue
- AOS Database
- CPT Compiler
- Compilation Thread
- Executing Code
- Take Sample
- New Code
- Executing Code
Outline

1. Motivation
2. Dynamic Optimistic Interprocedural Type Analysis (DOIT)
3. Immutability Analysis Opportunities for Dynamic IPA
4. Future Challenges
Future Challenges

- Integrating Dynamic IPA into Adaptive Optimization and Invalidation
- Automatic inference of Dynamic IPA properties of interest
- Application of Dynamic IPA to verification
- Refining granularity of Dynamic IPA from methods to basic blocks