Automatically Checking the Correctness of Program Analyses and Transformations

Compilers have many bugs

Searched for “incorrect” and “wrong” in the gcc-bugs mailing list. Some of the results:

- [Bug middle-end/19650] New: miscompilation of correct code
- [Bug c++/19731] arguments incorrectly named in static member specialization
- [Bug rtl-optimization/13300] Variable incorrectly identified as a biv
- [Bug rtl-optimization/16052] strength reduction produces wrong code
- [Bug tree-optimization/19633] local address incorrectly thought to escape
- [Bug target/19683] New: MIPS wrong-code for 64-bit multiply
- [Bug c++/19605] Wrong member offset in inherited classes
- [Bug java/19295] (4.0 regression) Incorrect bytecode produced for bitwise AND

Total of 545 matches...
And this is only for January 2005!
On a mature compiler!

Compiler bugs cause problems

The focus: compiler optimizations

- A key part of any optimizing compiler

The focus: compiler optimizations

- A key part of any optimizing compiler
- Hard to get optimizations right
  - Lots of infrastructure-dependent details
  - There are many corner cases in each optimization
  - There are many optimizations and they interact in unexpected ways
  - It is hard to test all these corner cases and all these interactions

Goals

- Make it easier to write compiler optimizations
  – student in an undergrad compiler course should be able to write optimizations
- Provide strong guarantees about the correctness of optimizations
  – automatically (no user intervention at all)
  – statically (before the opts are even run once)
- Expressive enough for realistic optimizations
The Rhodium work
- A domain-specific language for writing optimizations: Rhodium
- A correctness checker for Rhodium optimizations
- An execution engine for Rhodium optimizations
- Implemented and checked the correctness of a variety of realistic optimizations

Broader implications
- Many other kinds of program manipulators: code refactoring tools, static checkers
  - My work is about program analyses and transformations, the core of any program manipulator
- Enables safe extensible program manipulators
  - Allow end programmers to easily and safely extend program manipulators
  - Improve programmer productivity

Outline
- Introduction
  - Overview of the Rhodium system
- Writing Rhodium optimizations
- Checking Rhodium optimizations
- Evaluation

Rhodium system overview
Written by me
Rhodium Execution engine  Checker

Written by programmer
Rdm Opt  Rdm Opt  ...  Rdm Opt

Rhodium system overview
Written by me
Rhodium Execution engine  Checker

Written by programmer
Rdm Opt  Rdm Opt  ...  Rdm Opt

Checker  Checker  ...  Checker
Rhodium system overview

The technical problem

- Tension between:
  - Expressiveness
  - Automated correctness checking

- Challenge: develop techniques
  - that will go a long way in terms of expressiveness
  - that allow correctness to be checked

Contribution: three techniques

1. Rhodium is declarative
   - declare intent using rules
   - execution engine takes care of the rest
Contribution: three techniques

1. Rhodium is declarative
   - declare intent using rules
   - execution engine takes care of the rest

2. Factor out heuristics
   - legal transformations
   - vs. profitable transformations

3. Split verification task
   - opt-dependent
   - vs. opt-independent
Contribution: three techniques

1. Rhodium is declarative
2. Factor out heuristics
3. Split verification task

Result:
- Expressive language
- Automated correctness checking

Outline

- Introduction
- Overview of the Rhodium system
  - Writing Rhodium optimizations
  - Checking Rhodium optimizations
  - Evaluation

MustPointTo analysis

MustPointTo info in Rhodium

MustPointTo info in Rhodium

Fact correct on edge if:
- whenever program execution reaches edge, meaning of fact evaluates to true in the program state
Propagating facts

\[ a = \& b \]

\[ a \rightarrow b \]

\[ c = a \]

\[ a \rightarrow b \]

\[ c \rightarrow d = \* c \]

\[ \ldots \]

\[ \text{define fact} \]

\[ \text{mustPointTo}(X: \text{Var}, Y: \text{Var}) \]

\[ \text{with meaning} \]

\[ X == \& Y \]

\[ \text{if} \]

\[ \text{currStmt} == \]

\[ [X = \& Y] \]

\[ \text{then} \]

\[ \text{mustPointTo}(X, Y)@\text{out} \]

\[ \text{mustPointTo}(a, b) \]

\[ \text{mustPointTo}(c, b) \]

\[ \ldots \]

Transformations

\[ a = \& b \]

\[ a \rightarrow b \]

\[ c = a \]

\[ a \rightarrow b \]

\[ c \rightarrow d = \* c \]

\[ \ldots \]

\[ \text{define fact} \]

\[ \text{mustPointTo}(X: \text{Var}, Y: \text{Var}) \]

\[ \text{with meaning} \]

\[ X == \& Y \]

\[ \text{if} \]

\[ \text{currStmt} == \]

\[ [X = \& Y] \]

\[ \text{then} \]

\[ \text{mustPointTo}(X, Y)@\text{out} \]

\[ \text{mustPointTo}(a, b) \]

\[ \text{mustPointTo}(c, b) \]

\[ \ldots \]
Transformations

\[ d = c \]
\[ a = b \]
\[ c = a \]
\[ a \rightarrow b \]
\[ c \rightarrow b \]

\[ d = b \]

Profitability heuristics

Profitability heuristic example 1

- Inlining
- Many heuristics to determine when to inline a function
  - compute function sizes, estimate code-size increase, estimate performance benefit
  - maybe even use AI techniques to make the decision
- However, these heuristics do not affect the correctness of inlining
- They are just used to choose which of the correct set of transformations to perform

Profitability heuristic example 2

- PRE as code duplication followed by CSE

\[ a := \ldots; \]
\[ b := \ldots; \]
\[ if (...) { \]
\[ \quad a := \ldots; \]
\[ \quad x := a + b; \]
\[ \quad \} else { \]
\[ \quad \ldots \]
\[ \} \]
\[ x := a + b; \]
Profitability heuristic example 2

- PRE as code duplication followed by CSE

```java
// Code duplication
// CSE
// self-assignment removal
a := ...;
b := ...;
if (...) {
a := ...;
x := a + b;
} else {
...}
x := a + b;
```

Profitability heuristic example 2

- Legal placements of \( x := a + b \)

Profitable placement

```java
// Code duplication
// CSE
// self-assignment removal
a := ...;
b := ...;
if (...) {
a := ...;
x := a + b;
} else {
...}
x := a + b;
```

Semantics of a Rhodium opt

- Run propagation rules in a loop until there are no more changes (optimistic iterative analysis)
- Then run transformation rules
- Then run profitability heuristics
- For better precision, combine propagation rules and transformations rules using our previous composition framework [POPL 02]

More facts

- Define fact `mustNotPointTo(X:Var,Y:Var)` with meaning \( X \neq Y \)
- Define fact `doesNotPointIntoHeap(X:Var)` with meaning \( \exists \exists \exists \exists Y:Var . X == \& Y \)
- Define fact `hasConstantValue(X:Var,C:Const)` with meaning \( X == C \)

More rules

```java
if currStmt == \[ X = *A \] ∧ mustNotPointToHeap(A)@in ∧ ∀∀∀∀ B:Var . mayPointTo(A,B)@in ⇒ mustNotPointTo(B,Y)
then mustNotPointTo(X,Y)@out
```

```java
if currStmt == \[ Y = I + BE \] ∧ varEqualArray(X,A,J)@in ∧ equalsPlus(J,I,BE)@in ∧ ¬ mayDef(X) ∧ ¬ mayDefArray(A) ∧ unchanged(BE)
then varEqualArray(X,A,Y)@out
```

More in Rhodium

- More powerful pointer analyses
  - Heap summaries
- Analyses across procedures
  - Interprocedural analyses
- Analyses that don’t care about the order of statements
  - Flow-insensitive analyses
Outline

- Introduction
- Overview of the Rhodium system
- Writing Rhodium optimizations
  - Checking Rhodium optimizations
- Evaluation

Rhodium correctness checker

![Diagram of Rhodium correctness checker process]

Checker

Rdm Opt

Checker

Rdm Opt

Checker

Rdm Opt

Checker

Rdm Opt

Exec

Rhodium optimization

define fact ... → if ... then ... → if ... then transform ... → Profitability heuristics

Checker

Automatic theorem prover

Checker

Automatic theorem prover
Dimensions of evaluation

- Ease of use
- Correctness guarantees
- Usefulness of the checker
- Expressiveness

Outline

- Introduction
- Overview of the Rhodium system
- Writing Rhodium optimizations
- Checking Rhodium optimizations

• Evaluation
Ease of use

• Joao Dias
  – third year graduate student in compilers at Harvard
  – less than 45 mins to write CSE and copy prop

• Erika Rice, summer 2004
  – only knowledge of compilers: one undergrad class
  – started writing Rhodium optimizations in a few days

• Simple interface to the compiler’s structures
  – pattern matching
  – “flow functions” familiar to compiler 101 students

Correctness guarantees

• Once checked, optimizations are guaranteed to be correct

• Caveat: trusted computing base
  – execution engine
  – checker implementation
  – proofs done by hand once by me

• Adding a new optimization does not increase the size of the trusted computing base

Usefulness of the checker

• Found subtle bugs in my initial implementation of various optimizations

```plaintext
define fact equals(X:Var, E:Expr)
with meaning
[ X == E ]
if currStmt == [ X = E ]
then equals(X,E)@out
```

```plaintext
define fact equals(x:Var, E:Expr)
with meaning
[ x == E ]
if currStmt == [ x = E ]
then equals(x,E)@out
```

Usefulness of the checker

• Found subtle bugs in my initial implementation of various optimizations

```plaintext
define fact equals(X:Var, E:Expr)
with meaning
[ X == E ]
if currStmt == [ X = E ] ∧ "X does not appear in E"
then equals(X,E)@out
```

```plaintext
define fact equals(x:Var, E:Expr)
with meaning
[ x == E ]
if currStmt == [ x = E ] ∧ "E does not use X"
then equals(x,E)@out
```

Usefulness of the checker

• Found subtle bugs in my initial implementation of various optimizations

```plaintext
define fact equals(X:Var, E:Expr)
with meaning
[ X == E ]
if currStmt == [ X = E ] ∧ "E does not use X"
then equals(X,E)@out
```

```plaintext
define fact equals(x:Var, E:Expr)
with meaning
[ x == E ]
if currStmt == [ x = E ] ∧ "x does not appear in E"
then equals(x,E)@out
```

Usefulness of the checker

• Found subtle bugs in my initial implementation of various optimizations

```plaintext
define fact equals(X:Var, E:Expr)
with meaning
[ X == E ]
if currStmt == [ X = E ] ∧ "X does not appear in E"
then equals(X,E)@out
```

```plaintext
define fact equals(x:Var, E:Expr)
with meaning
[ x == E ]
if currStmt == [ x = E ] ∧ "x does not appear in E"
then equals(x,E)@out
```

Rhodium expressiveness

• Traditional optimizations:
  – const prop and folding, branch folding, dead assignment elim, common sub-expression elim, partial redundancy elim, partial dead assignment elim, arithmetic invariant detection, and integer range analysis.

• Pointer analyses
  – must-point-to analysis, Andersen’s may-point-to analysis with heap summaries

• Loop opts
  – loop-induction-variable strength reduction, code hoisting, code sinking

• Array opts
  – constant propagation through array elements, redundant array load elimination
Rhodium expressiveness

- Traditional optimizations:
  - const prop and folding, branch folding, dead assignment elim, common sub-expression elim, partial redundancy elim, partial dead assignment elim, arithmetic invariant detection, and integer range analysis.
- Pointer analyses
  - must-point-to analysis, Andersen’s may-point-to analysis with heap summaries
- Loop opts
  - loop-induction-variable strength reduction, code hoisting, code sinking
- Array opts
  - constant propagation through array elements, redundant array load elimination

Expressiveness limitations

- May not be able to express your optimization in Rhodium
  - opts that build complicated data structures
  - opts that perform complicated many-to-many transformations (e.g.: loop fusion, loop unrolling)
- A correct Rhodium optimization may be rejected by the correctness checker
  - limitations of the theorem prover
  - limitations of first-order logic

Summary

- Rhodium system
  - makes it easier to write optimizations
  - provides correctness guarantees
  - is expressive enough for realistic optimizations

- Rhodium system provides a foundation for safe extensible program manipulators