Proving Optimizations Correct using Parameterized Program Equivalence

Sudipta Kundu
Zachary Tatlock
Sorin Lerner

University of California, San Diego
Compiler Correctness

- Difficult to develop reliable compilers:
  - large, complex programs
  - take a long time to mature

- Consequence: buggy compilers, but also …
  - hinders development of new languages, architectures
  - discourages user from extending compiler

- Focus on correctness of compiler optimizations
  - many intricate opts, unexpected interactions
  - turning off optimizations often no longer an option
### Existing Automated Techniques

<table>
<thead>
<tr>
<th>Translation Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>prove equivalence</td>
</tr>
<tr>
<td>for each transformation</td>
</tr>
<tr>
<td>every compiler execution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a priori Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>prove correctness</td>
</tr>
<tr>
<td>before compiler runs</td>
</tr>
<tr>
<td>once and for all</td>
</tr>
</tbody>
</table>

- [Pnueli et al.]
- [Necula 00]
- TVOC [Zuck et al.]
- Verified TV [Tristan et al.]
- CompCert [Leroy et al.]
- Rhodium [Lerner et al.]
Focus on Automated Techniques

Key Insight:
Adapt Translation Validation to once-and-for-all setting

Our Approach:
complex loop opts + once-and-for-all correctness

Scope of Guarantee
Verify Compiler Run
Verify Compiler

Scope of Guarantee

Expressive Power
Complex Loop Optns
One-to-one Rewrites

[Neuca 00]
Rhodium [Lerner et al.]
Generalize to Parameterized Programs

\[
\begin{align*}
&k := 0 \\
&\text{while } k < 100 \{ \\
&\quad a[k] += k \\
&\quad k++ \\
&\}
\end{align*}
\]
Generalize to Parameterized Programs

I := 0
while (I < 100) {
    a[k] += I
    I++
}

I := 0
while (I < 100) {
    a[k] += I
    I++
}

Generalize to Parameterized Programs

Input
PProg

Output
PProg

Equivalence Checking

Output
Prog

Input
Prog
Generalize to Parameterized Programs

```
I := 0
while(I < E){
a[k] += I
I++
}
```
I := 0
while(I < E) {
  S
  I++
}

Equivalence Checking
Generalize to Parameterized Progs

Optimization

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>PProg</td>
<td>PProg</td>
</tr>
</tbody>
</table>

Parameterized Equivalence Checking

Output

Generalize to Parameterized Programs

Optimization Instance

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog</td>
<td>Prog</td>
</tr>
</tbody>
</table>

Equivalence Checking

Instance
Contributions

1. Optimization
   - Input PProg
   - Output PProg

Parameterized Equivalence Checking (PEC)
- Proves opts correct statically and automatically
- Can reason about many-to-many opts

2. Parameterized Equivalence Checking (PEC)
   - Proves opts correct statically and automatically
   - Can reason about many-to-many opts

3. Expressed and proved a variety of opts correct
   - Which Rhodium could not have proven correct
   - Software pipelining and other complex loop opts
Parameterized Rewrite Rules

- Loop Peeling
  - move iter out of loop

- ids range over:
  - \( I \): variable
  - \( E \): expression
  - \( S \): statement

- Shift final iteration after loop

- Side conditions indicate when rewrite safe

\[
\begin{align*}
I & := 0 \\
\text{while}(I < E) \{ \\
S & \\
I++ & \\
\}
\end{align*}
\]

\[
\begin{align*}
I & := 0 \\
\text{while}(I < E-1) \{ \\
S & \\
I++ & \\
\}
\end{align*}
\]

where:
- \( E > 0 \)
- \( S \) does not modify \( I, E \)
Enable loop unrolling

Apply Rewrite
1. Match parameters
2. Check side conditions
3. Rewrite

where:
- \( 100 > 0 \)
- \( a[k] += k \) does not modify \( k, 100 \)

Divisible by 3
- Directly unroll by 3

Not divisible by 3
- Hard to unroll by 3
Checking Correctness

Parameterized Equivalence Checking

OPT
Checking Correctness

Generalization of [Necula 00]

Relate

Generalization of [Zuck et al. 04]

OPT

Permute

OR

✓

✗

✓

✗
Checking Rewrite Rules

- Programs equivalent:
  - Consider CFGs
  - Start in equal states
  - End in equal states

\[ \sigma_1 = \sigma_2 \]
Checking Rewrite Rules

- Relate Executions:
  1. Find synchronization points
  2. Generate invariants
  3. Check invariants preserved
     - Use auto theorem prover
     - Each invariant must imply successor invariants
     - Strengthen if inv too weak
1. Find Synchronization Points

- Traverse in lockstep
- Stop at stmt params
- Prune infeasible paths
  - From Path: $E \leq 0$
  - Side Conds: $E > 0$
  - ∴ Path never executes
2. Generate Invariants

- Invariants:
  - predicates over $\sigma_1, \sigma_2$

- Gen initial invariant:
  - $\sigma_1 = \sigma_2$ AND
  - strongest post cond

$$\begin{array}{|c|c|}
\hline
A(\sigma_1, \sigma_2) & \sigma_1 = \sigma_2 \wedge \text{eval(} \sigma_1, I < E \text{)} \wedge \text{eval(} \sigma_2, I < E - 1 \text{)} \\
\hline
B(\sigma_1, \sigma_2) & \sigma_1 = \sigma_2 \wedge \text{eval(} \sigma_1, I < E \text{)} \wedge \text{eval(} \sigma_2, I \geq E - 1 \text{)} \\
\hline
\end{array}$$
3. Check Invariants

- Each invariant must imply successor invariants

- Query Theorem Prover

<table>
<thead>
<tr>
<th>Entry</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Exit</td>
<td></td>
</tr>
</tbody>
</table>

| A($\sigma_1, \sigma_2$) | $\sigma_1=\sigma_2 \land \text{eval}(\sigma_1, I < E)$ $
\land \text{eval}(\sigma_2, I < E-1)$ |
|------------------------|----------------------------------------------------------------------------------|
| B($\sigma_1, \sigma_2$) | $\sigma_1=\sigma_2 \land \text{eval}(\sigma_1, I < E)$ $
\land \text{eval}(\sigma_2, I \geq E-1)$ |
3. Check Invariants

ATP Query:

∀ σ_1 σ_2 . 

\( A(σ_1, σ_2) \land \sigma_1' = \text{step}(σ_1, S; I++; I < E) \land \) 

\( \sigma_2' = \text{step}(σ_2, S; I++; I \geq E - 1) \) 

\( \rightarrow B(σ_1', σ_2') \)

<table>
<thead>
<tr>
<th></th>
<th>A(σ_1, σ_2)</th>
<th>B(σ_1, σ_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>σ_1=σ_2 \land \text{eval}(σ_1, I &lt; E) \land \text{eval}(σ_2, I &lt; E - 1)</td>
<td>\text{σ_1=σ_2 \land eval}(σ_1, I &lt; E) \land \text{eval}(σ_2, I \geq E - 1)</td>
</tr>
</tbody>
</table>
3. Check Invariants

ATP Query:

\[ \forall \sigma_1, \sigma_2 . \]
\[ A(\sigma_1, \sigma_2) \land \]
\[ \sigma_1' = \text{step}(\sigma_1, S; I++; I < E) \land \]
\[ \sigma_2' = \text{step}(\sigma_2, S; I++; I \geq E-1) \]

\[ B(\sigma_1', \sigma_2') \]

<table>
<thead>
<tr>
<th>ATP Query:</th>
<th>[ A(\sigma_1, \sigma_2) ]</th>
<th>[ \sigma_1 = \sigma_2 \land \text{eval}(\sigma_1, I &lt; E) \land \text{eval}(\sigma_2, I &lt; E-1) ]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ B(\sigma_1', \sigma_2') ]</td>
<td>[ \sigma_1' = \sigma_2' \land \text{eval}(\sigma_1', I &lt; E) \land \text{eval}(\sigma_2', I \geq E-1) ]</td>
</tr>
</tbody>
</table>
3. Check Invariants

ATP Query:

\[
\forall \sigma_1 \sigma_2 . \quad A(\sigma_1, \sigma_2) \land \\
\sigma_1' = \text{step}(\sigma_1, S; I++; I < E) \land \\
\sigma_2' = \text{step}(\sigma_2, S; I++; I \geq E-1) \\
\rightarrow B(\sigma_1', \sigma_2')
\]

Strengthen A if the theorem prover fails
3. Check Invariants

- Each invariant must imply successor invariants

- Query Auto Theorem Prover

<table>
<thead>
<tr>
<th>Entry</th>
<th>A</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>B</td>
<td>Exit</td>
<td>✓</td>
</tr>
</tbody>
</table>

\[
\sigma_1 = \sigma_2 \\
A(\sigma_1, \sigma_2) \quad \sigma_1 = \sigma_2 \land \text{eval}(\sigma_1, I < E) \\
\land \text{eval}(\sigma_2, I < E-1) \\
B(\sigma_1, \sigma_2) \quad \sigma_1 = \sigma_2 \land \text{eval}(\sigma_1, I < E) \\
\land \text{eval}(\sigma_2, I \geq E-1)
\]
Checking Correctness

Works well for structure-preserving optimizations

Generalization of [Necula 00]

Relate

OR

Permute

Generalization of [Zuck et al. 04]

Works well for non structure-preserving optimizations
Permute Module

Loop Interchange

where: (side condition)

• Generate correspondence between loop indices
• Ask ATP to show that (side condition) implies:
  • correspondence is one-to-one
  • For all $I, I' \in R_1$ and $J, J' \in R_2$
    $S[I, J]$ commutes with $S[I', J']$
Contributions

1. Optimization
   - Input PProg
   - Output PProg

Parameterized Equivalence Checking

2. Parameterized Equivalence Checking (PEC)
   - Proves opts correct statically and automatically
   - Can reason about many-to-many opts

3. Expressed and proved a variety of opts correct
   - Which Rhodium could not have proven correct
   - Software pipelining and other complex loop opts
# Optimizations Proved Correct

<table>
<thead>
<tr>
<th>Category 1:</th>
<th>Copy propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEC and Rhodium formulation equivalent</td>
<td>Constant propagation</td>
</tr>
<tr>
<td></td>
<td>Common sub-expression elim</td>
</tr>
<tr>
<td></td>
<td>Partial redundancy elim</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category 2:</th>
<th>Loop invariant code hoisting</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEC formulation easier, more general</td>
<td>Conditional speculation</td>
</tr>
<tr>
<td></td>
<td>Speculation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category 3:</th>
<th>Software pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressible in PEC</td>
<td>Loop unswitching</td>
</tr>
<tr>
<td>No Rhodium formulation possible</td>
<td>Loop unrolling</td>
</tr>
<tr>
<td></td>
<td>Loop peeling</td>
</tr>
<tr>
<td></td>
<td>Loop splitting</td>
</tr>
<tr>
<td></td>
<td>Loop alignment</td>
</tr>
<tr>
<td></td>
<td>Loop interchange</td>
</tr>
<tr>
<td></td>
<td>Loop reversal</td>
</tr>
<tr>
<td></td>
<td>Loop skewing</td>
</tr>
<tr>
<td></td>
<td>Loop fusion</td>
</tr>
<tr>
<td></td>
<td>Loop distribution</td>
</tr>
</tbody>
</table>
Parameterized Equivalence Checking (PEC)
- proves opts correct statically and automatically
- can reason about many-to-many opts

Expressed and proved a variety of opts correct
- which Rhodium could not have proven correct
- software pipelining and other complex loop opts