Atomicity Refinement for Verified Compilation

David Pichardie  Vincent Laporte  Gustavo Petri  Jan Vitek
Suresh Jagannathan

INRIA, Purdue University, Université Rennes 1

January 25th, 2013
Context: Verified Compilation

CompCert (Leroy 2006)

- Optimizing compiler from C to assembly
- Fully verified in the Coq proof assistant
CompCert-TSO (Ševčík et al. 2011)

- TSO semantics for all intermediate languages from x86 to C

- Low-level concurrency primitives: threads, compare-and-swap
This Work

Towards higher-level languages
- general synchronization mechanisms
- garbage collection

Strong formal guarantees
- mechanized proofs

Methodology
- Refinement: replace the implementation by a high-level specification
Running Example: Lock

Lock L

\textit{critical section}

Unlock L
Running Example: Lock

Lock L

critical section

Unlock L

Proof technique: backward simulation

What? Prove that any trace of the target is a trace of the source.

How? Match every state in the target trace with a state in the source.

→ Too hard because one has to consider too many interleavings (and relaxed behaviors): compilation introduces memory accesses and loops.
Running Example: Lock

```
old ← CAS(L, 0 → 1)
curr ← old
if (curr = 0)
curr ← L
if (old ≠ 0)
```

Proof technique: backward simulation

**What?** Prove that any trace of the target is a trace of the source.

**How?** Match every state in the target trace with a state in the source.
Running Example: Lock

Lock L
    critical section
Unlock L

Proof technique: backward simulation

What? Prove that any trace of the target is a trace of the source.
How? Match every state in the target trace with a state in the source.

→ Too hard because one has to consider too many interleavings (and relaxed behaviors): compilation introduces memory accesses and loops.
Compiler Architecture

Source

RTL

→ Compilation step

→ Simulation proof

Intractable brute-force proof

Lock L
Compiler Architecture

Repeat
   old ← CAS(L, 0 → 1)
   curr ← old
   While curr = 0
      curr ← L
   Until old = 0

→ Compilation step
····→ Simulation proof
Compiler Architecture

Lock L

Repeat
old ← CAS(L, 0 → 1)
curr ← old
While curr = 0
curr ← L
Until old = 0

→ Compilation step
→ Simulation proof
≺ Refinement

Atomic{ Assume(L = 0); L ← 1 }
### Refinement Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trans</strong></td>
<td>( s_1 \preceq s_2 \quad s_2 \preceq s_3 \rightarrow s_1 \preceq s_3 )</td>
</tr>
<tr>
<td><strong>Repeat</strong></td>
<td>( \text{repeat } s \text{ until } c \preceq \text{ loop } (s; \text{assume}(\neg c)); s; \text{assume}(c) )</td>
</tr>
<tr>
<td><strong>IfBranch</strong></td>
<td>( \text{if then } s_1 \text{ else } s_2 \preceq \text{ branch } (\text{assume}(c); s_1), (\text{assume}(\neg c); s_2) )</td>
</tr>
<tr>
<td><strong>IfAtomic</strong></td>
<td>( \text{if } c \text{ then (atomic } s_0 \text{) else (atomic } s_1 \text{) } \preceq \text{ atomic (if } c \text{ then } s_0 \text{ else } s_1 )</td>
</tr>
<tr>
<td><strong>Cas-fail</strong></td>
<td>( \text{cas}(d, r, o, n); \text{assume}(o \neq d) \preceq \text{ fence; load}(d, r) )</td>
</tr>
<tr>
<td><strong>Cas-success</strong></td>
<td>( \text{cas}(d, r, o, n); \text{assume}(o = d) \preceq \text{ atomic (load}(d, r); \text{assume}(o = d); \text{store}(n, r)) )</td>
</tr>
<tr>
<td><strong>SwapAssume</strong></td>
<td>( \text{defines}(s) \cap \text{uses}(c) = \emptyset ) \rightarrow s; \text{assume}(c) \preceq \text{ assume}(c); s )</td>
</tr>
<tr>
<td><strong>Dead</strong></td>
<td>( s_1 \text{ is dead} \rightarrow s_1; s_2 \preceq s_2 )</td>
</tr>
<tr>
<td><strong>FenceAtomic</strong></td>
<td>( \text{fence} \preceq \text{ atomic skip} )</td>
</tr>
<tr>
<td><strong>FenceElim</strong></td>
<td>( \text{fence} \preceq \text{ skip} )</td>
</tr>
<tr>
<td><strong>AfterAbort</strong></td>
<td>( \text{abort; } s \preceq \text{ abort} )</td>
</tr>
<tr>
<td><strong>GrowAtomicLocal</strong></td>
<td>( s_0 \in {\text{load}<em>{\text{Loc}}(r, d), \text{store}</em>{\text{Loc}}(d, r)} ) \rightarrow s_0; \text{atomic } s_1 \preceq \text{ atomic } (s_0; s_1) )</td>
</tr>
<tr>
<td><strong>MakeStoreAtomic</strong></td>
<td>( s \in {\text{store}(r, d), \text{release store}(r, d)} \rightarrow s; \text{fence} \preceq \text{ atomic } s )</td>
</tr>
<tr>
<td><strong>MakeLoadAtomic</strong></td>
<td>( \text{fence; load}(r, d) \preceq \text{ atomic load}(r, d) )</td>
</tr>
</tbody>
</table>
Interactive Refinement Proof

**Repeat**
- \( \text{old} \leftarrow \text{CAS}(L, 0 \rightarrow 1) \)
- \( \text{curr} \leftarrow \text{old} \)
- While \( \text{curr} = 0 \)
  - \( \text{curr} \leftarrow L \)
- Until \( \text{old} = 0 \)

**Loop**
- \( \text{old} \leftarrow \text{CAS}(L, 0 \rightarrow 1) \)
- \( \text{curr} \leftarrow \text{old} \)
- While \( \text{curr} = 0 \)
  - \( \text{curr} \leftarrow L \)
- Assume(\( \text{old} \neq 0 \))
- \( \text{old} \leftarrow \text{CAS}(L, 0 \rightarrow 1) \)
- \( \text{curr} \leftarrow \text{old} \)
- While \( \text{curr} = 0 \)
  - \( \text{curr} \leftarrow L \)
- Assume(\( \text{old} = 0 \))

\[
\text{Repeat s Until c} \Leftrightarrow \text{Loop } \{ \text{s; Assume}(\neg c) \}; \text{s; Assume}(c)
\]
Interactive Refinement Proof

Loop
old ← CAS(L, 0 → 1)
curr ← old
While curr = 0
curr ← L
Assume(old ≠ 0)
old ← CAS(L, 0 → 1)
curr ← old
While curr = 0
curr ← L
Assume(old = 0)
defines(s) ∩ uses(c) = ∅
s; Assume(c) ⇐ Assume(c); s
Interactive Refinement Proof

Loop

old ← CAS(L, 0 → 1)
Assume(old ≠ 0)
curr ← old
While curr = 0
    curr ← L
old ← CAS(L, 0 → 1)
Assume(old = 0)
curr ← old
While curr = 0
    curr ← L

≈

Loop

Fence
old ← L
curr ← old
While curr = 0
    curr ← L
old ← CAS(L, 0 → 1)
Assume(old = 0)
curr ← old
While curr = 0
    curr ← L

o ← CAS(d,e → n); Assume(o ≠ e) ≼ Fence; o ← d
Interactive Refinement Proof

Loop
  Fence
  old ← L
  curr ← old
  While curr = 0
    curr ← L
  old ← CAS(L, 0 → 1)
  Assume(old = 0)
  curr ← old
  While curr = 0
    curr ← L

o ← CAS(d, e → n); Assume(o = e) ≼

Atomic { o ← d; Assume(o = e); d ← n }
Interactive Refinement Proof

Loop
Fence
old ← L
curr ← old
While curr = 0
curr ← L

Atomic
old ← L
Assume(old = 0)
L ← 1
curr ← old
While curr = 0
curr ← L

Atomic
old ← L
Assume(old = 0)
L ← 1

\[ s_1 \text{ is dead} \]
\[ s_1; s_2 \preceq s_2 \]
Summary

- Compiler for a high-level concurrent language written in Coq
- Proof that program refinement implies behaviour inclusion
- Sound refinement rules
- Case studies: spin lock and concurrent garbage collector (using a permission system to handle such complex examples)
Summary

- Compiler for a high-level concurrent language written in Coq
- Proof that program refinement implies behaviour inclusion
- Sound refinement rules
- Case studies: spin lock and concurrent garbage collector (using a permission system to handle such complex examples)

Questions?