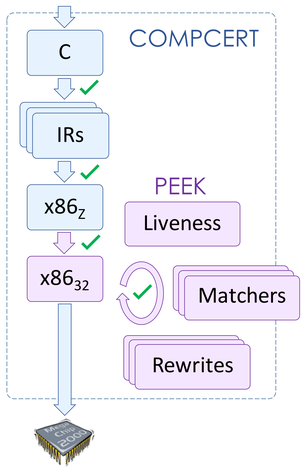
Proof Automation for Verified Peephole Optimizations

Daryl Zuniga, Eric Mullen, Zachary Tatlock, Dan Grossman

University of Washington

Peek

Running Peek



Peepholes

MOVs to XCHG

```
mov %eax, %ecx
mov %edx, %eax
mov %ecx, %edx
```
⇒
```
xchg %eax, %edx
```

Bit twidling

```
sub %eax, %ecx
mov %ecx, %eax
dec %eax
```
⇒
```
notl %eax
add %ecx, %eax
```

Removing a redundant load

```
mov %eax, -8(%ebp)
mov -8(%ebp), %ecx
```
⇒
```
mov %eax, -8(%ebp)
mov %eax, %ecx
```

Jump to conditional move

```
test %ecx, %ecx
je .L0
mov %edx, %ebx
.L0
```
⇒
```
test %ecx, %ecx
cmovene %edx %ebx
```

Verifying Peek

End-to-End Guarantee

- C program and assembly program exhibit same behavior
- Equivalence based on program trace
- CompCert's key contribution
- Peek maintains guarantee

Liveness Analysis

- Verified conservative liveness analysis
- Many peepholes use dead registers
- Allows state matching to be weaker than full equality

Local to Global

- Peek lifts local correctness into a global program transformation
- **Single-Entry** is a key property

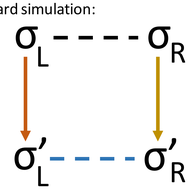
Execution Engine

- Verified by translation validation and simulation relations

Peephole Proof

Behavioral Equivalence

- Key local correctness property
- Forward simulation:



1. Use information about how the original ("**left**") program stepped
2. Construct step relations for the entire transformed ("**right**") program
3. Prove the final states of the left and right programs match

Making Peephole Proofs Easier

Challenge

Low-Level Details

- PC overflow
- Jump semantics
- Memory metadata and permissions
- Pointer values

Per-Peephole Effort

- Proofs required for every peephole
- Must be tractable for users
- Must compile fast
- Must be parameterized

Expressiveness

- Forward jumps
- Branching execution
- Memory operations
- Bit vector arithmetic

Solution for Peek

Unfolding the Left (1)

- Boring plumbing
- Highly similar across peepholes
- Well suited for tactic automation

Stepping the Right (2)

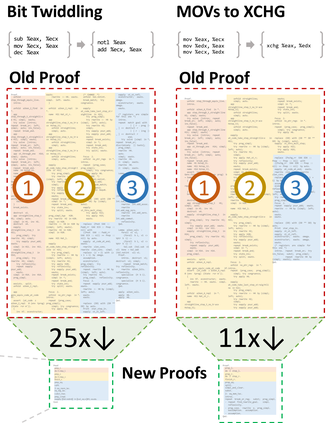
- Most difficult
- Highly similar across peepholes
- Good abstractions are the key
- E.g. straight-line vs jumps

Matching States (3)

- Most interesting
- Highly peephole specific
- Supported by rich library of lemmas and tactics

Example Proof

```
Proof.
  prep_l.
  do 3 step_l.
  prep_r.
  do 3 step_r.
  finish_r.
  prep_eq.
  split.
  2: eq_mem_tac.
  eq_reg_tac.
  subst_max.
  preg_simpl.
  eapply* [[Val.notint]] in [[val_eq x]]#1; eauto.
Qed.
```



General Strategy

Refactoring Proofs

- Get initial QEDs as fast as possible
- Refactor common tasks into tactics
- Identify patterns as candidates for abstraction

Structural Tactics

- Tactic library being developed at UW: https://github.com/uwplse/StructTact
- Make proofs robust to small changes
- Reduce explicit hypothesis naming

```
apply H1 in H2.
apply* [[my_term]] in [[my_rel]]#1.
```

- Reduce unification order dependencies

Tactics to Lemmas

- Often tactics can be replaced with lemmas
- Lemmas provide a strong interface
- Tactic failure is difficult to debug
- Lemma pre-conditions and post-conditions are difficult to get right

Better Abstractions

- Let users focus on essential differences
- Eliminates boilerplate code
- Smaller proof contexts

Evaluation

28 Peepholes Verified

- Interesting behavior including:
  - local jumps
  - memory operations
  - two's compliment arithmetic
  - conditional execution
  - more
- Useful reference for future peepholes

Lines of Proof

Performance improvements

- 4% speedup on verified SHA-256 [A. Appel, Verification of a Cryptographic Primitive, TOPLAS 2015]
- 0.7% speedup on SHA3 from CompCert benchmarks
- 81 peepholes fired across CompCert benchmarks