"Super" optimization

CSE 501

Spring 15

Course Outline

- Static analysis
- Language design
- Program Verification
- Dynamic analysis
- New compilers
  - superoptimizers ⟵ We are here
  - synthesis-based translation

Announcements

- HW1 scores out
- HW2 due on June 9th
  - Post on forum if you have questions
- Project presentations next Thursday
  - 10 min presentation for each group
  - Signup will be posted
- Project final report due on June 9th

Why compilers

- Utilize everything we learned in this class
  - Static analysis
  - Verification
  - Testing
- What this class was originally about!

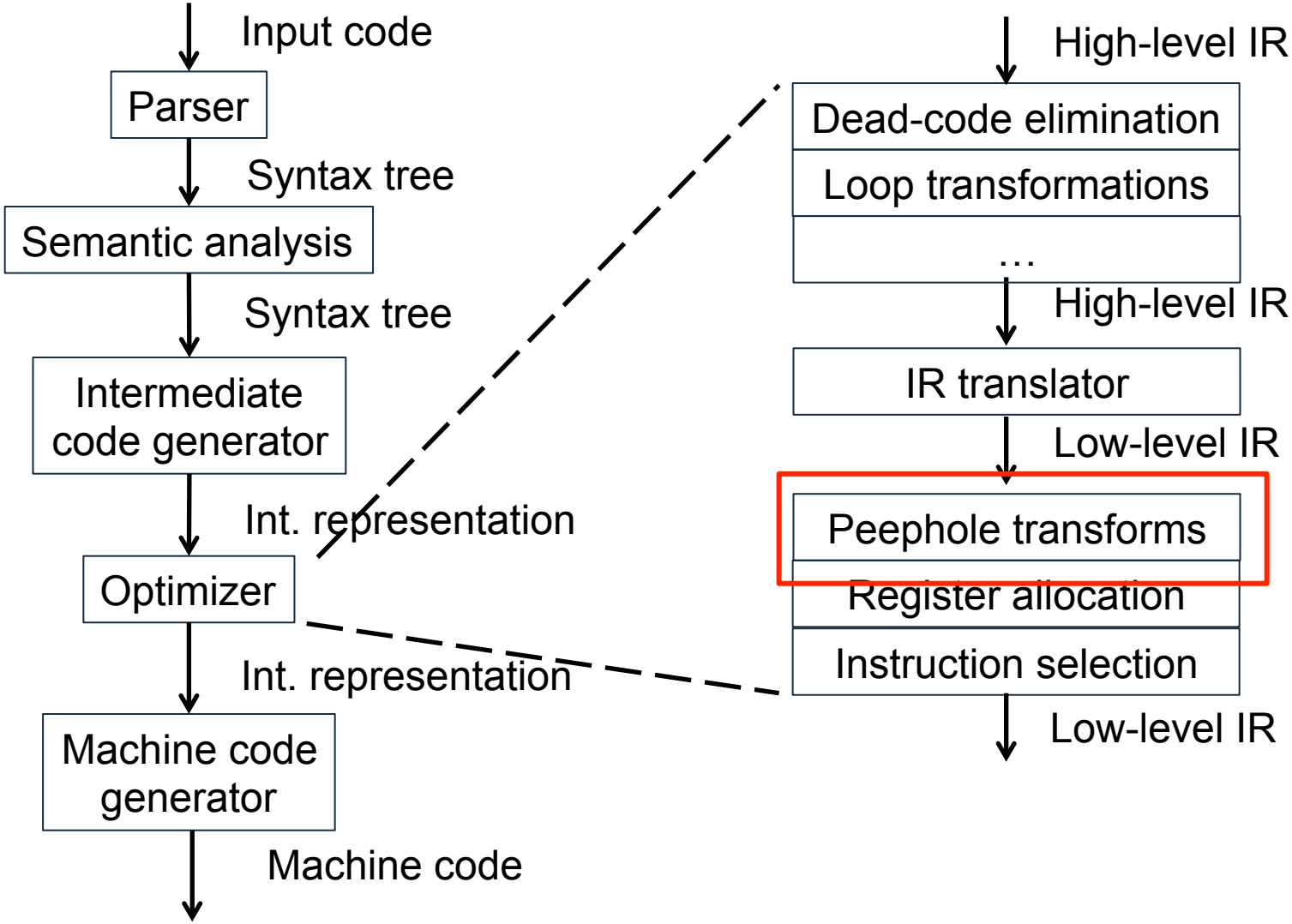
Outline for today

- Classical optimizing compilers
- Superoptimization
  - High-level idea
  - Algorithms
  - Examples
    - Massalin
    - Denali
    - STOKE

Optimizing compilers

- Tools that generate optimal code
  - Smallest executable size
  - Shortest runtime
  - Smallest footprint
- Issues to consider
  - Soundness
  - Compilation time
  - Optimality

Optimizing Compilers



Peephole optimization

- Purely syntactic driven transformation rules
  - Usually done on low-level IR
- Rules have the form:
  - If instructions match pattern then apply rewrite
- "grep" over instruction sequence

Example: eliminate redundant stores

```
LD a, R0      load address a into R0
ST R0, a      store contents of R0 to address a
```

```
LD a, R0
```

In general:

{ LD %x, %y ; ST %y, %x } → { LD %x, %y }

- But store instruction must not have a label

Example: control-flow optimizations

```
goto L1                    goto L2
…                  ⟶       …
L1: goto L2                L1: goto L2
```

```
goto L1                    if c goto L2
…                  ⟶       goto L3
L1: if c goto L2
L3:                        L3:
```

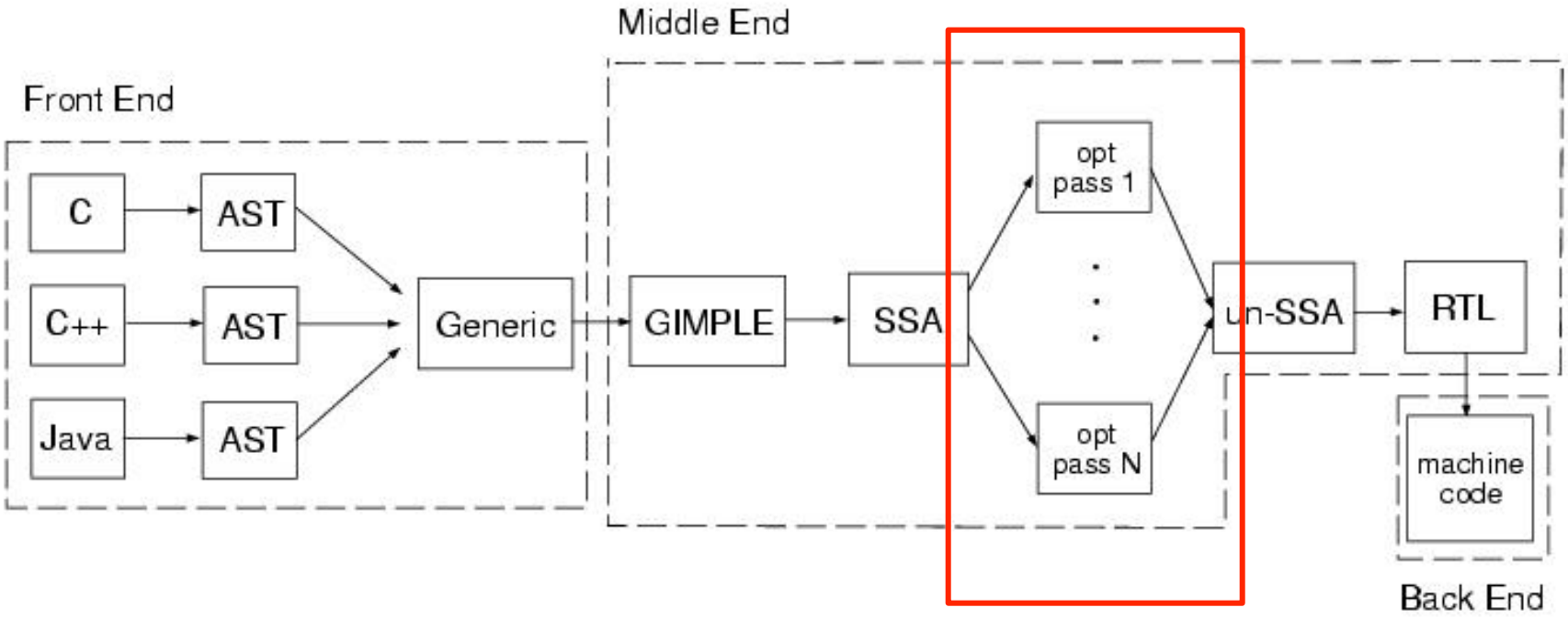
Example: algebraic rewrites

```
add %x, 0    ⟶    (none)
mul %x, 2    ⟶    shl %x, 1
```

Example: machine idioms

```
add %x, 1    ⟶    inc %x
```

Example: GCC architecture



GCC optimization passes

- Tree SSA passes
  - Remove useless statements
  - OpenMP lowering
  - OpenMP expansion
  - Lower control flow
  - Lower exception
  - Build the control flow graph
  - Enter static single assignment form
  - Warn for uninitialized variables
  - Dead code elimination
  - Dominator optimizations
  - Forward propagation of single-use variables
  - Copy Renaming
  - PHI node optimizations
  - May-alias optimization
  - Profiling
  - Static profile estimation

GCC optimization passes

- Lower complex arithmetic
- Scalar replacement of aggregates
- Dead store elimination
- Tail recursion elimination
- Forward store motion
- Partial redundancy elimination
- Full redundancy elimination
- Loop optimizations:

  Loop invariant motion.
  Canonical induction variable creation.
  Induction variable optimizations.
  Loop unswitching.
  Vectorization.
  SLP Vectorization.
  Autoparallelization.

- Tree level if-conversion for vectorizer
- Conditional constant propagation
- Conditional copy propagation
- Value range propagation
- Folding built-in functions
- Split critical edges
- Control dependence dead code elimination
- Tail call elimination
- Warn for function return without value
- Leave static single assignment form
- Merge PHI nodes that feed into one another
- Return value optimization
- Return slot optimization
- Optimize calls to __builtin_object_size
- Loop invariant motion
- Loop nest optimizations
- Removal of empty loops
- Unrolling of small loops
- Predictive commoning
- Array prefetching
- Reassociation
- Optimization of stdarg functions

GCC optimization pass manager

- The pass manager is located in passes.c, tree-optimize.c and tree-pass.h. It processes passes as described in passes.def.

- Its job is to run all of the individual passes in the correct order, and take care of standard bookkeeping that applies to every pass.

Issues with pass organization

- Pass ordering
- Software maintainability
  - Dependencies among passes
  - Adding / removing passes
- Code "optimality"?
  - This is equivalent to inventing the best algorithm for carrying out a certain task
  - Problem is undecidable in general [Touati et al]

Alternatives

- CompCert
  - Addresses soundness issue

- Superoptimization
  - Addresses the ordering issue

Superoptimization

- Given code fragment **C**:
  - exhaustively search through all semantically-equivalent rewrites of **C** such that it is optimal
- Issues
  - Exhaustive: search space blows up?
  - Semantically-equivalent: how to check?
  - Optimality: how to evaluate?

Superoptimizer, the original version
[Massalin, ASPLOS 87]

- Finds shortest program that computes the same function as **C**

- **C** is a sequence of assembly instructions
  - Straight-line code, no loops, jumps etc

Algorithm

```
L = generateCandidates(C);
best = C;
for (l in L) {
  if (l == C)
    best = l;
}
return best;
```

Generating candidates

- Choose subset of target machine's instruction set
- Enumerate instructions of length 1, 2, … , length of original code
  - Is this a good strategy?
- Optimization:
  - Remove obvious "non-candidates"
  - `MOV A, B; MOV B, A`

Proving program equivalence

- Strategy 1: Encode inputs as boolean vectors
- Encode instruction semantics as functions on boolean vectors

- Example:
  - AND R_A, R_B :
    $\langle a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 \rangle$ & $\langle b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 \rangle$
    $= \langle a_0 \& b_0\ a_1 \& b_1, \ldots, a_7 \& b_7 \rangle$

- Represent functions as minterms

Minterms

- Product of all variables in function s.t. it is equal to 1 on exactly one row in the truth table
- Example:

A	B	C	A | B | C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Minterms:
¬A¬BC
¬ABC
A¬B¬C
ABC

Minterms

- Can compare implementations semantically by matching minterms

- Issue:
  - Number of minterms might blow up
  - Model 8-bit register addition as 8 bit-wise functions
    - How many minterms will there be?

Proving program equivalence

- Strategy 2: random testing
- Small-scope hypothesis:
  - a high proportion of errors can be found by testing a program for all test inputs within some small scope

- Choose random test inputs
- Run boolean verification if no errors found

Denali [Joshi et al, PLDI 02]

- Same setup as Massalin
  - Optimize straight-line code
  - Search for semantic-equivalent sequence of instructions that can be executed using the fewest number of cycles
- Better ways to generate candidates
  - Specialized data structure for storing functional equivalences
- Use SAT solver for find sequence of instructions

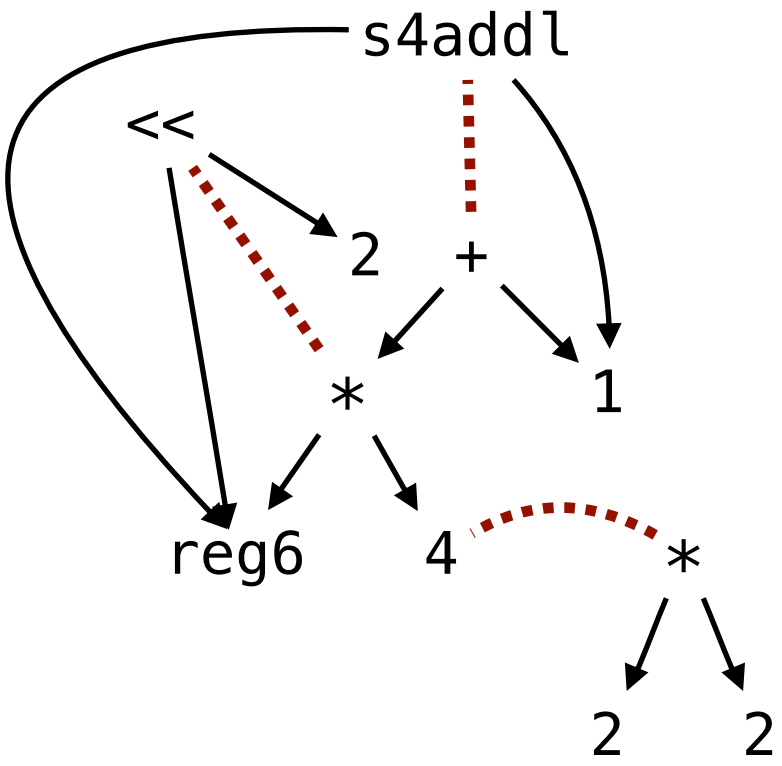
Generating candidates

- Algebraic axioms
  - ∀ x, y . add64(x, y) = add64(y, x)
  - Basically functional equivalences

- Architectural axioms
  - ∀ x, y . add(x, y) = add64(add64(x, y), carry(x, y))
  - Basically architecture-specific peephole optimizations

Generating candidates

- E-graph encodes all equivalent expressions



Generating candidates

- Generation process is non-deterministic
  - "[Denali generates all semantically equivalent expressions] if the matching phase is allowed to run long enough, and if the heuristics that are designed to keep the matcher from running forever don't mistakenly stop it from running long enough"

Search for the optimal sequence

```
assert(rewrite axioms);
assert(instruction constraints);

best = c;
for (k = (# cycles need to execute C – 1) → 0) {
    r = assert(¬ ∃  k cycle instruction that
                    implements C);
  if (r == UNSAT) { break; }
  else { best = r; }
}
```

Details

- Profile program for memory timings
  - Might not be accurate but results are still sound
- Candidate generation might not be complete due to non-determinism
- Bottomline: output is “near-optimal”

STOKE [Schkufza et al, ASPLOS 13]

- Same setting as before
  - Sequence of straight-line machine instructions
  - Goal: find faster versions as compared to original

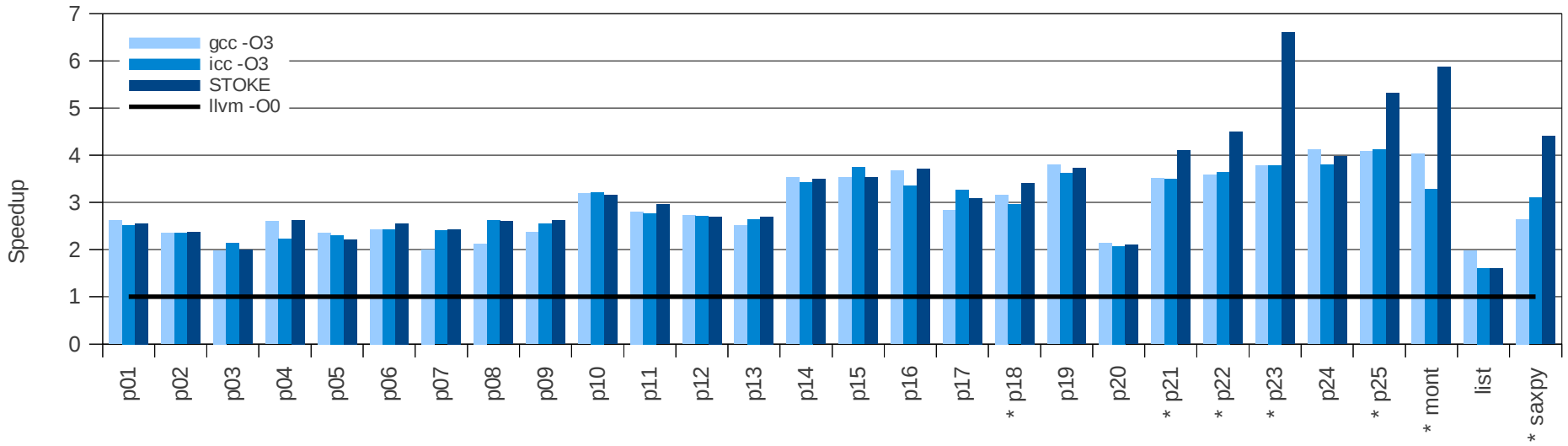
- Instead of writing equivalence axioms and generating all candidates, use random search
  - However, use "controlled" random search

Search algorithm

- Key: generate random seed candidates
  - Not care about correctness
- Mutation operator randomly adds / removes / or swaps instructions
- Runs test cases to prune away incorrect candidates
  - Formal verification afterwards
- Once a candidate is proven to be correct, optimize as before

Some results

"Hacker's delight" benchmarks



Discussion

- Extending to other architectures
- Crafting axioms
- Using domain knowledge

Summary

- Pass scheduling problem is hard
- Superoptimization: cast optimization as search
  - How to generate candidates
  - How to verify correctness
- Handling non straight-line code?
- Next (and last) lecture: synthesis-based compilation