# CSc 553
# Principles of Compilation

## 08. Code Optimization

Saumya Debray

*The University of Arizona*

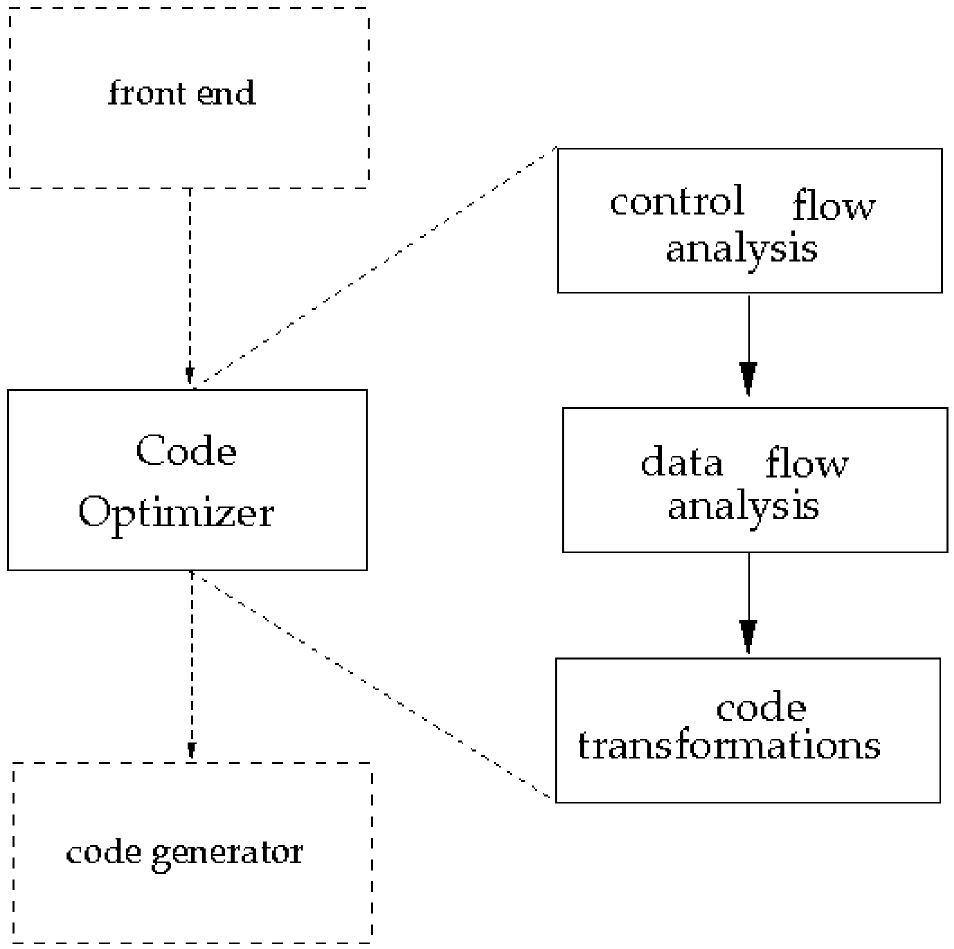
*Tucson, AZ 85721*

# Code Optimization

- Aim: to improve program performance.
  - “optimization” a misnomer: attaining “optimal” performance is impossible or impractical in general.

- Criteria:
  - must be “safe,” i.e., preserve program semantics;
  - on average, should improve performance measurably;
    - *occasionally, a few programs may suffer performance degradation.*
  - the transformation should be worth the effort.

# Code Optimizer Organization



# Code Optimization: Basic Requirements

- ***Fundamental Requirement***: safety.

  The "observable behavior" of the program (i.e., the output computed for any given input) must not change.

- Program analyses must be correspondingly safe.
  - most runtime properties of a program are statically undecidable.
  - static program analyses are (necessarily) imprecise.
  - any imprecision *must* be in the direction of safety.

# Some Important Optimizations

- Peephole optimization:
  - simple pattern-matching based transformations to improve common code sequences
- Loop transformations:
  - reduces the number/cost of instructions within loops. E.g.:
    - invariant code motion out of loops
    - induction variable elimination
    - loop unrolling
- Function-preserving transformations:
  - reduces unnecessary computations, but not aimed specifically at loops. E.g.:
    - common subexpression elimination
    - copy propagation
    - dead/unreachable code elimination

# 1. Peephole Optimization

Basic idea:

- examine the instruction sequence for simple short patterns that can be replaced by equivalent but more efficient patterns
- the patterns correspond to commonly occuring instruction sequences

# Peephole Optimization: Example

```
enter copy
tmp$0 := 0
i := tmp$0
tmp$1 := 0
i := tmp$1
label Lbl0
tmp$2 := a
tmp$3 := i * 1
tmp$2 := tmp$2 + tmp$3
tmp$4 := deref(tmp$2)
tmp$5 := 0
if tmp$4 > tmp$5 goto Lbl1
goto Lbl2
label Lbl1
tmp$8 := b
tmp$9 := i * 1
tmp$8 := tmp$8 + tmp$9
tmp$10 := a
tmp$11 := i * 1
tmp$10 := tmp$10 + tmp$11
...
```

Do you see any patterns we could optimize?

# Peephole Optimization: Example

```
enter copy
tmp$0 := 0
i := tmp$0
tmp$1 := 0
i := tmp$1
label Lbl0
tmp$2 := a
tmp$3 := i * 1
tmp$2 := tmp$2 + tmp$3
tmp$4 := deref(tmp$2)
tmp$5 := 0
if tmp$4 > tmp$5 goto Lbl1
goto Lbl2
label Lbl1
tmp$8 := b
tmp$9 := i * 1
tmp$8 := tmp$8 + tmp$9
tmp$10 := a
tmp$11 := i * 1
tmp$10 := tmp$10 + tmp$11
...
```

assignment

i := 0

# Peephole Optimization: Example

indexing
into a char
array

```
enter copy
tmp$0 := 0
i := tmp$0
tmp$1 := 0
i := tmp$1
label Lbl0
tmp$2 := a
tmp$3 := i * 1          tmp$3 := i
tmp$2 := tmp$2 + tmp$3
tmp$4 := deref(tmp$2)
tmp$5 := 0
if tmp$4 > tmp$5 goto Lbl1
goto Lbl2
label Lbl1
tmp$8 := b
tmp$9 := i * 1
tmp$8 := tmp$8 + tmp$9
tmp$10 := a
tmp$11 := i * 1
tmp$10 := tmp$10 + tmp$11
...
```

# Peephole Optimization: Example

```
enter copy
tmp$0 := 0
i := tmp$0
tmp$1 := 0
i := tmp$1
label Lbl0
tmp$2 := a
tmp$3 := i * 1
tmp$2 := tmp$2 + tmp$3
tmp$4 := deref(tmp$2)
tmp$5 := 0
if tmp$4 > tmp$5 goto Lbl1
goto Lbl2
label Lbl1
tmp$8 := b
tmp$9 := i * 1
tmp$8 := tmp$8 + tmp$9
tmp$10 := a
tmp$11 := i * 1
tmp$10 := tmp$10 + tmp$11
...
```

control flow improvement

```
if tmp$4 ≤ tmp$5 goto Lbl2
label Lbl1
```

# Peephole Optimization: common patterns

- *Null sequences*: delete useless operations
- *Combine operations*: replace several instructions with a single equivalent instruction
  - e.g.: a jump to a jump ⇒ a jump to the ultimate target
- *Algebraic simplification*: use algebraic laws to simplify or replace instructions
  - e.g.: i *1 ⇒ i
- …

# 2. Copy Propagation

- Copy instructions: of the form ‘x = y’
  - arise from normal code generation (e.g., assignment of an expression)
  - also as a result of other optimizations, e.g., global common subexpression elimination (CSE).

- Goal of Copy Propagation:
  - given a copy instruction ‘x = y’, try to replace subsequent uses of x by y
    - must guarantee that x and y have the same value at the point of replacement
  - if the copy instruction becomes dead as a result, it can then be optimized away (dead code elimination)

# Local Copy Propagation (intra-block)

Given a copy instruction 'x = y' in a basic block:

- iterate through the instructions in the rest of the block
- replace any use of x by y:

```
x = y              x = y
z = x + w    =>    z = y + w
```

- if either x or y is redefined, stop propagating
    - x and y are no longer guaranteed to have the same value

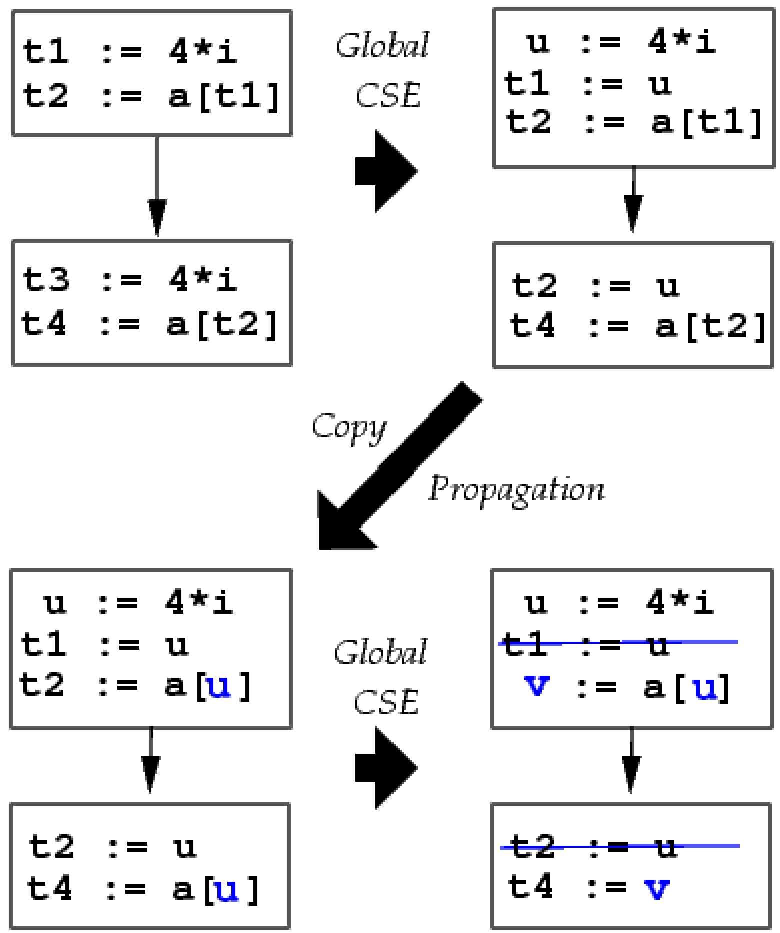
# Local Copy Propagation: Example

| Initial code sequence | OK |
| --- | --- |
| x = 10 | x = 10 |
| y = x | y = x |
| z = y + 1 | z = 10 + 1 |
| x = z * 2 | x = z * 2 |
| w = x - 1 | w = x - 1 |
| y = u + 1 | y = u + 1 |
| w = z * y | w = z * y |

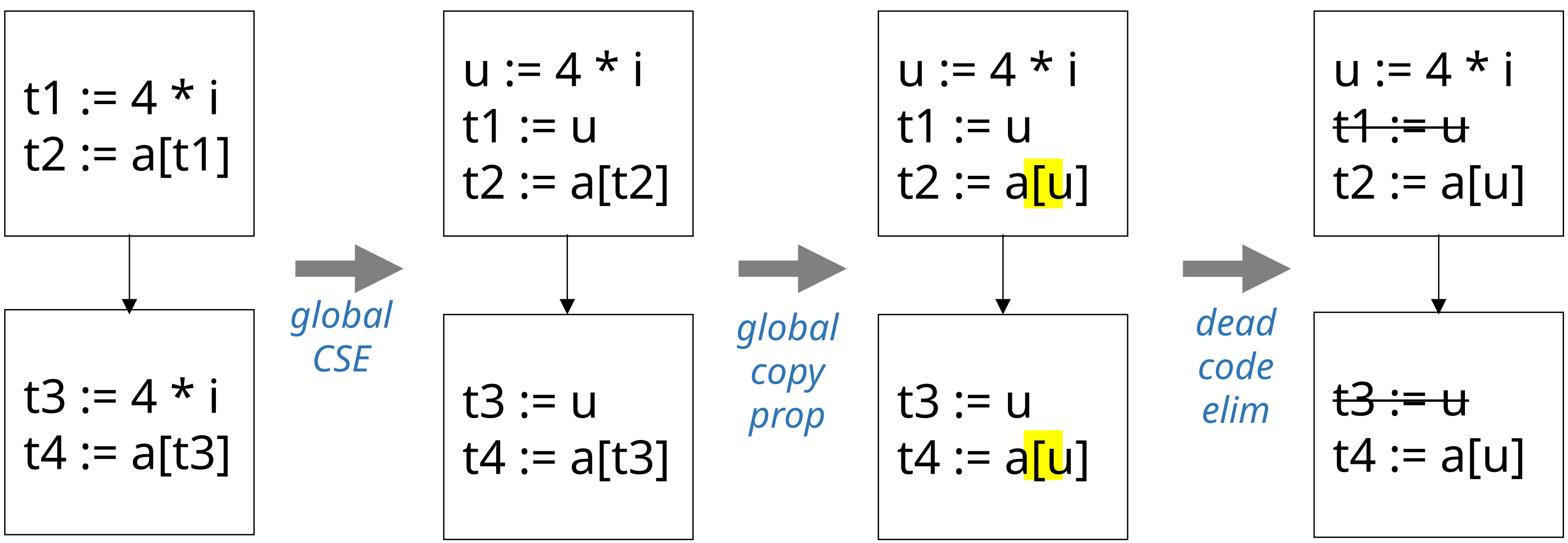
# Local Copy Propagation: Example

| Initial code sequence | OK | Not OK |
|---|---|---|
| x = 10 | x = 10 | x = 10 |
| y = x | y = x | y = x |
| z = y + 1 | z = 10 + 1 | z = y + 1 |
| x = z * 2 | x = z * 2 | x = z * 2 ← x redefined |
| w = x - 1 | w = x - 1 | w = 10 - 1 |
| y = u + 1 | y = u + 1 | y = u + 1 ← y redefined |
| w = z * y | w = z * y | w = z * 10 |

# Global Copy Propagation: Example

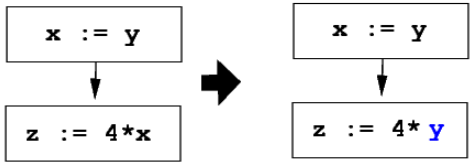


# Global Copy Propagation: Example

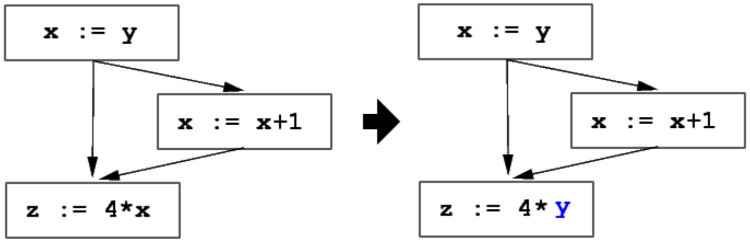


# When is Copy Propagation Legal?

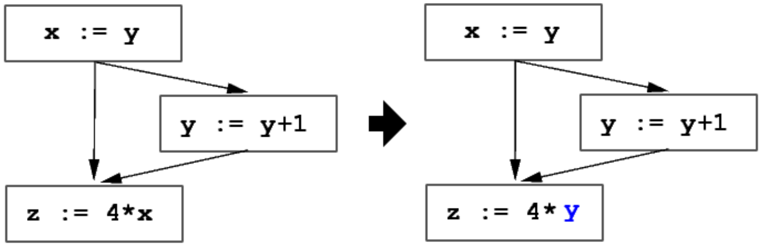
- OK:




- Not OK:




- Not OK:



# Legality Conditions for Copy Propagation

A copy instruction $s \equiv$ 'x = y' can be propagated to a use $u$ of x if:

1. $s$ is the only definition of x reaching $u$; and
2. there is no path from $s$ to $u$ that redefines y and which does not then go through $s$ again.

Condition 1 can be checked using *u-d chains*.

This is is generalization of reaching definitions that associates, with each use $u$ of a variable x, all the definitions of x that reach $u$.

# Effects of Copy Propagation

- When a copy instruction $s \equiv$ ‘x = y’ is propagated to uses of x, the no. of uses of s decreases.

- If the number of uses of $s$ goes to 0, then s becomes dead code, and can be eliminated.

# Optimization 3. Dead Code Elimination

*Definition*: An instruction is *dead* if the value it computes can be guaranteed to not be used.

I $\equiv$ 'x = $e$' is dead if

- x is dead at the point immediately after I, and
- the evaluation of $e$ has no side effects on any variable that is live at the point after I.

Dead code can arise due to other optimizations, e.g., constant propagation, copy propagation.

# Dead Code and its Elimination

Eliminating a dead instruction can cause other instructions to become dead:

(1) x = y + z

← v, w dead; x live

(2) w = 4 * x

← v, x dead; w live

(3) v = w + 1

← v, w, x dead; instr. (3) dead

Goal: Identify instructions that are (a) dead, or (b) will become dead once other dead instructions are eliminated.

# Dead Code Elimination: Algorithm 1

1. mark all instructions 'live';
2. **repeat:**

   **for** each instruction $I \equiv$ 'x = ...':

   **if** the value of x defined by $I$

   ($i$) is not visible outside the current function; and

   ($ii$) is not used by any instr. $J$ ($J \neq I$) marked 'live' **then**:

   mark $I$ 'dead';

   **until** no more instructions can be marked.

requires global analysis to identify all uses of $I$

# Dead Code Elimination: Algorithm 2

**repeat**:

1. Perform liveness analysis
2. **for** each basic block B:

    liveset = OUT[B]    /* live variables at B's exit */

    **for** each instruction *I* in reverse order from B's end:

        **let** *I* ≡ 'x = y ⊕ z'

        **if** dst(I) ∈ liveset:

            liveset = (liveset – {x}) ∪ {y, z}

        **else**

            mark *I* as 'dead'

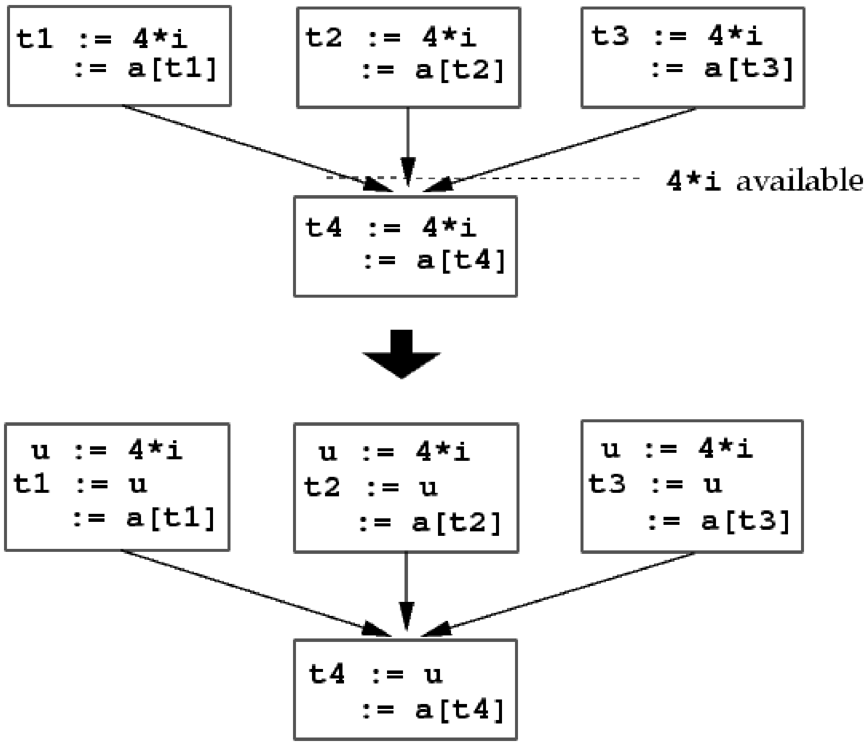
**until** no more instructions can be marked 'dead'

# 4. Common Subexpression Elimination

- *Goal*: to detect and eliminate repeated computations of the same expression.
- Can be done at two different levels:
  - Local CSE:
    - scope limited to a single basic block
  - Global CSE:
    - applies across basic block boundaries
    - uses *available expressions* analysis.

# Global Common Subexpression Elimination

Uses available expression information to identify common subexpressions



# Global CSE: Algorithm

- Compute available expressions for each block.
- Process each block B as follows:
  - for each instruction $I \equiv$ 'x = y ⊕ z' where 'y ⊕ z' is available immediately before $I$, do:
    1. find the evaluations of 'y ⊕ z' that reach I:

       *traverse B, and then the control flow edges, backwards, not going beyond any block that evaluates 'y ⊕ z'.*
    2. create a new variable u.
    3. replace each instruction 'w = y ⊕ z' found in (1) by the following:

       u = y ⊕ z

       w = u
    4. replace instruction $I$ by 'x = u'.

# Comments on Global CSE

- For “lightweight” expressions (e.g., ‘***p***’), CSE may be profitable only if the expression can be kept in a register.

  *But this means that the register is unavailable for other uses.*

- The algorithm given will miss the fact that ‘t1*4’ and ‘t3*4’ have the same value in

  t1 = x+y
  t2 = t1*4
  t3 = x+y
  t4 = t3*4

  This can be handled by iterative applications of global CSE + *copy propagation.*