COMPILER OPTIMIZATION

Baishakhi Ray

Optimization

- Optimization is our last compiler phase
- Most complexity in modern compilers is in the optimizer
  - Also by far the largest phase
- Optimizations are often applied to intermediate representations of code

When should we perform optimizations?

- On AST
  - Pro: Machine independent
  - Con: Too high level
- On assembly language
  - Pro: Exposes optimization opportunities
  - Con: Machine dependent
  - Con: Must reimplement optimizations when retargetting
- On an intermediate language
  - Pro: Machine independent
  - Pro: Exposes optimization opportunities

Intermediate Languages

- Intermediate language = high-level assembly
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    - E.g., push translates to several assembly instructions
  - Most opcodes correspond directly to assembly opcodes

Three-Address Intermediate Code

- Each instruction is of the form

  x := y op z (binary operation)

  x := op y   (unary operation)

  - y and z are registers or constants
  - Common form of intermediate code

- The expression x + y * z is translated

  t1 := y * z

  t2 := x + t1

  - Each subexpression has a “name”

Optimization Overview

- Optimization seeks to improve a program's resource utilization
    - Execution time (most often)
    - Code size
    - Network messages sent, etc.
- Optimization should not alter what the program computes
    - The answer must still be the same

A Classification of Optimizations

- For languages like C there are three granularities of optimizations
  1. Local optimizations
     - Apply to a basic block in isolation
  2. Global optimizations
     - Apply to a control-flow graph (method body) in isolation
  3. Inter-procedural optimizations
     - Apply across method boundaries

- Most compilers do (1), many do (2), few do (3)

Cost of Optimizations

- In practice, a conscious decision is made not to implement the fanciest optimization known

- Why?
  - Some optimizations are hard to implement
  - Some optimizations are costly in compilation time
  - Some optimizations have low benefit
  - Many fancy optimizations are all three!

- Goal: Maximum benefit for minimum cost

Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Example: algebraic simplification

Algebraic Simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1

- Some statements can be simplified

  x := x * 0 ⇒ x := 0

  y := y ** 2 ⇒ y := y * y

  x := x * 8 ⇒ x := x << 3

  x := x * 15 ⇒ t := x << 4; x := t - x

(on some machines << is faster than *; but not on all!)

Constant Folding

- Operations on constants can be computed at compile time
  - If there is a statement x := y op z
  - And y and z are constants
  - Then y op z can be computed at compile time
- Example: x := 2 + 2 ⇒ x := 4
- Example: if 2 < 0 jump L can be deleted
- When might constant folding be dangerous?
  - Floating point errors in cross-architecture compilation

Flow of Control Optimizations

- Eliminate unreachable basic blocks:
  - Code that is unreachable from the initial block
    - E.g., basic blocks that are not the target of any jump or “fall through” from a conditional
- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects (increased spatial locality)

Single Assignment Form

- Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment

- Rewrite intermediate code in single assignment form

x := z + y		b := z + y
a := x	⇒	a := b
x := 2 * x		x := 2 * b

(b is a fresh register)

- More complicated in general, due to loops

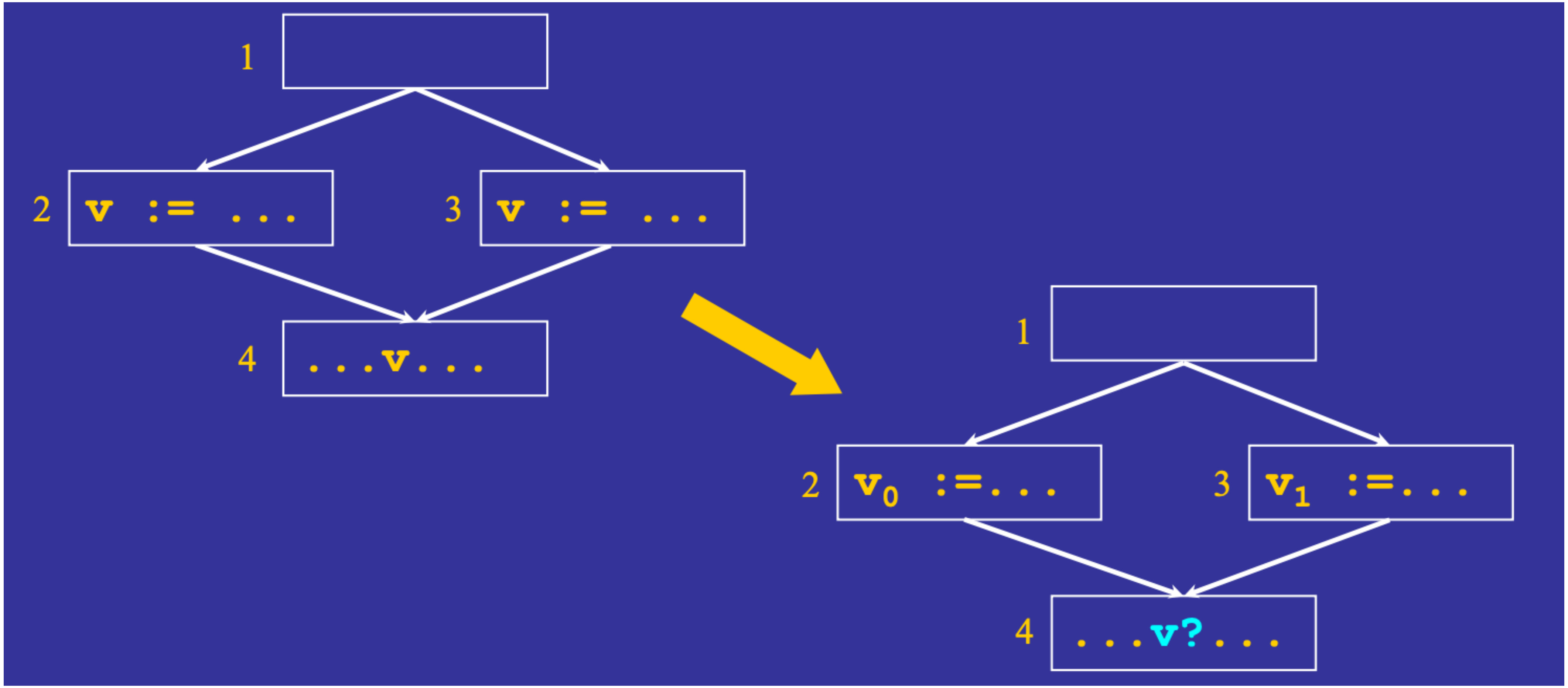
Static Single Assignment (SSA) Form

- Idea
  - Each variable has only one static definition
  - Makes it easier to reason about values instead of variables
  - The point of SSA form is to represent use-def information explicitly
- Transformation to SSA
  - Rename each definition
  - Rename all uses reached by that definition
- Example:

```
v  := …                    v0 := …
…  := … v …                …  := … v0 …
v  := …         ⟶          v1 := …
…  := … v …                …  := … v1 …
v  := …                    v2 := …
…  := … v …                …  := … v2 …
```

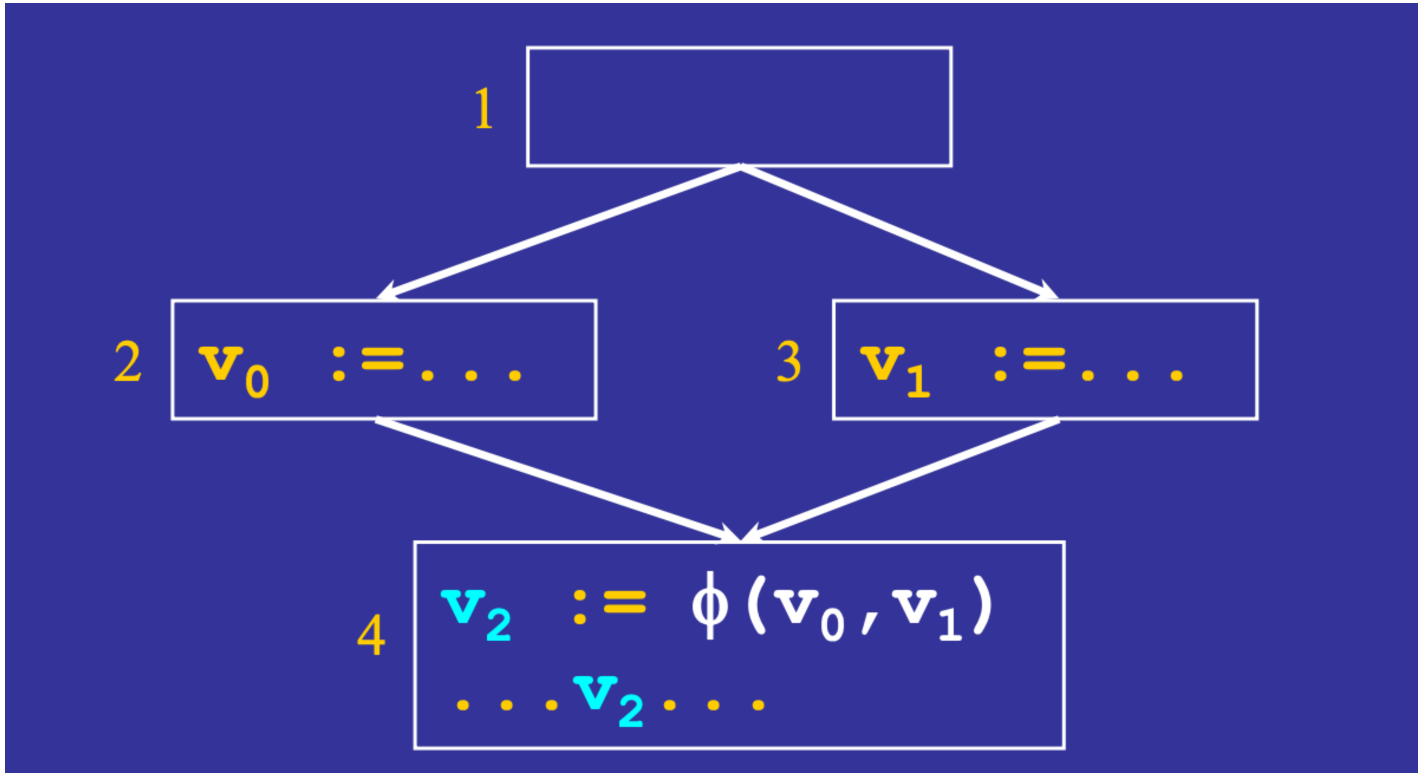
SSA and Control Flow

- Problem : A use may be reached by several definitions



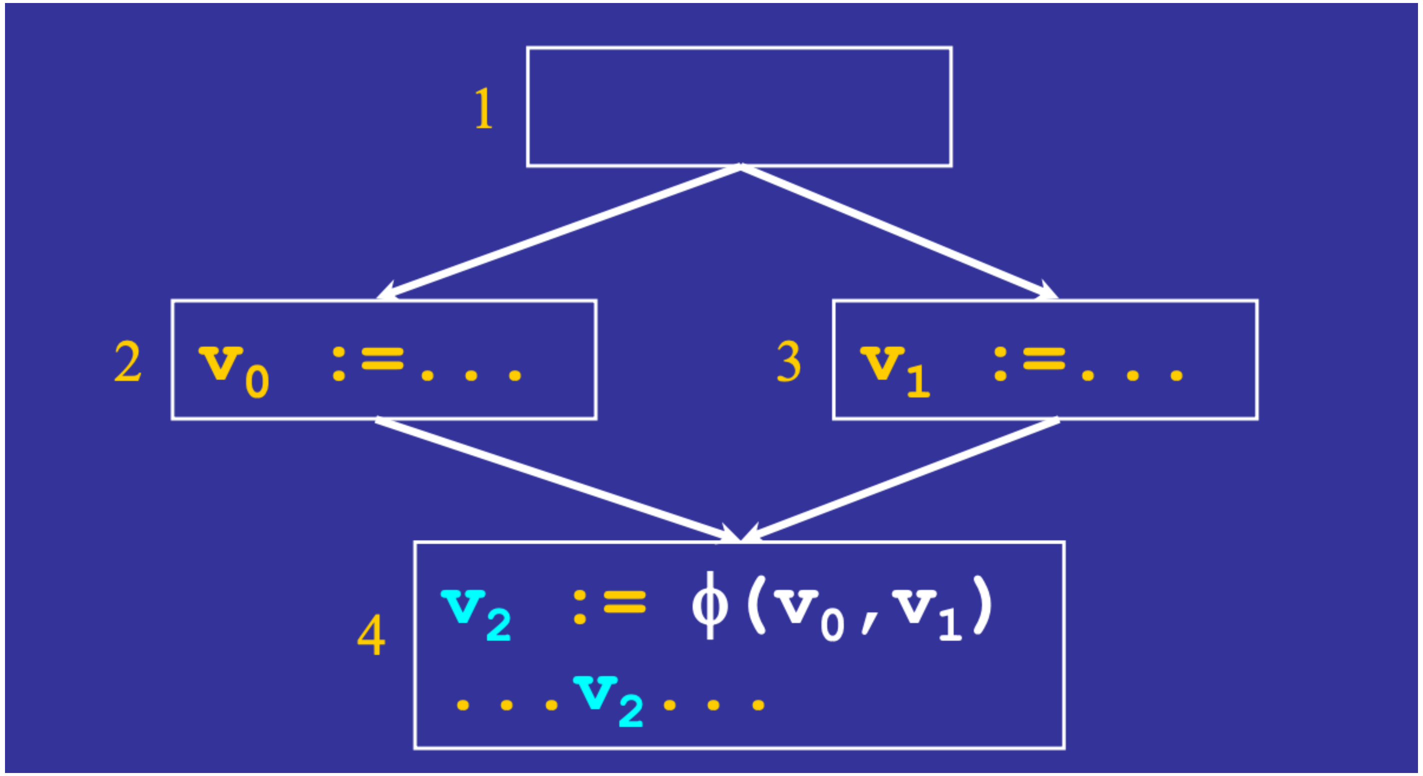
SSA and Control Flow (cont)

- Merging Definitions
  - ∅-functions merge multiple reaching definitions



SSA and Control Flow (cont)

- Merging Definitions
  - ∅-functions merge multiple reaching definitions



SSA vs. use-def chain

- SSA form is more constrained
- Advantages of SSA
  - More compact
  - Some analyses become simpler when each use has only one def
  - Value merging is explicit
  - Usually, easier to update and manipulate
- Furthermore
  - Eliminates false dependences (simplifying context)

SSA vs. use-def chain

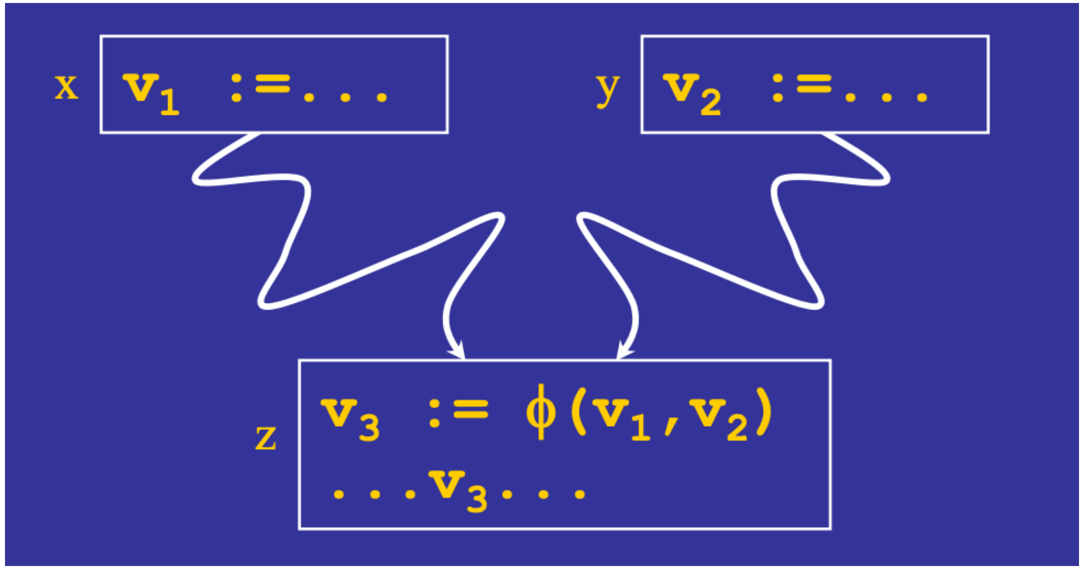
- Worst case du-chains?

```
switch (c1) {
      case 1: x = 1; break;
      case 2: x = 2; break;
      case 3: x = 3; break;
}
switch (c2) {
      case 1: y1 = x; break;
      case 2: y2 = x; break;
      case 3: y3 = x; break;
      case 4: y4 = x; break;
}
```

m defs and n uses leads to m x n du chains

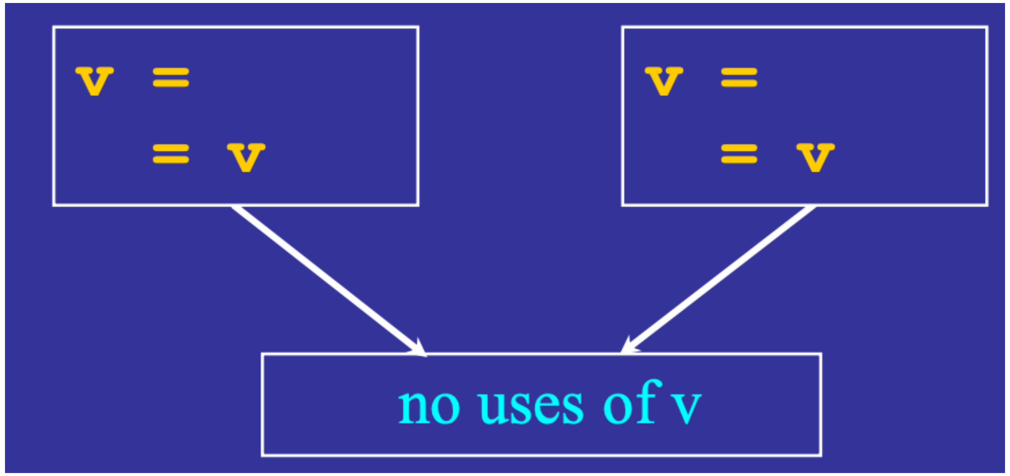
Transformation to SSA Form

- Two steps
  - Insert ∅-functions
  - Rename variables
- Basic Rule of Placing ∅-Functions?
  - If two distinct (non-null) paths x->z and y->z converge at node z, and nodes x and y contain definitions of variable v, then we insert a ∅-function for v at z



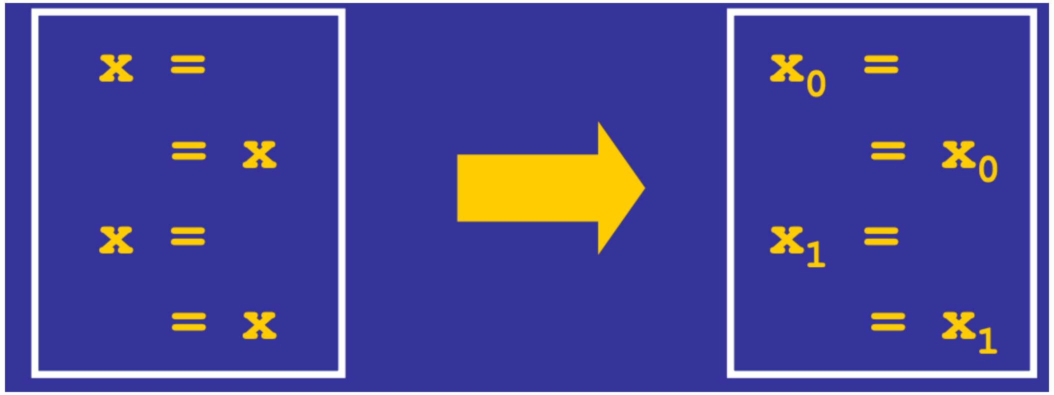
Approaches to Placing ∅-Functions

- Minimal
  - As few as possible subject to the basic rule
- Briggs-Minimal
  - Same as minimal, except v must be live across some edge of the CFG
    - Briggs Minimal will not place a ∅ function in this case because v is not live across any CFG edge.
    - Exploits the short lifetimes of many temporary variables



SSA: Variable Renaming

- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript
- Easy for straight forward code



- Harder when there’s control flow
  - For each use of x, find the definition of x that dominates it

Common Subexpression Elimination

- If
  - Basic block is in single assignment form
  - A definition x := is the first use of x in a block
- Then
  - When two assignments have the same rhs, they compute the same value
- Example:

x := y + z		x := y + z
…	⇒	…
w := y + z		w := x

(the values of x, y, and z do not change in the … code)

Copy Propagation

- If w := x appears in a block, replace subsequent uses of w with uses of x
  - Assumes single assignment form
- Example:

b := z + y		b := z + y
a := b	⇒	a := b
x := 2 * a		x := 2 * b

- Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination

Copy Propagation and Constant Folding

- Example:

a := 5		a := 5
x := 2 * a	⇒	x := 10
y := x + 6		y := 16
t := x * y		t := x << 4

Copy Propagation and Dead Code Elimination

- If
  - w := rhs appears in a basic block
  - w does not appear anywhere else in the program
- Then the statement w := rhs is dead and can be eliminated
  - Dead = does not contribute to the program's result
  - Example: (a is not used anywhere else)

x := z + y		b := z + y		b := z + y
a := x	⇒	a := b	⇒	x := 2 * b
x := 2 * a		x := 2 * b		

Applying Local Optimizations

- Each local optimization does little by itself
- Typically optimizations interact
  - Performing one optimization enables another
- Optimizing compilers repeat optimizations until no improvement is possible
  - The optimizer can also be stopped at any point to limit compilation time

An Example

- Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

- Algebraic optimization:

```
a := x ** 2        a := x * x
b := 3             b := 3
c := x             c := x
d := c * c         d := c * c
e := b * 2         e := b << 1
f := a + d         f := a + d
g := e * f         g := e * f
```

An Example

- Copy Propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example

- Dead code elimination:

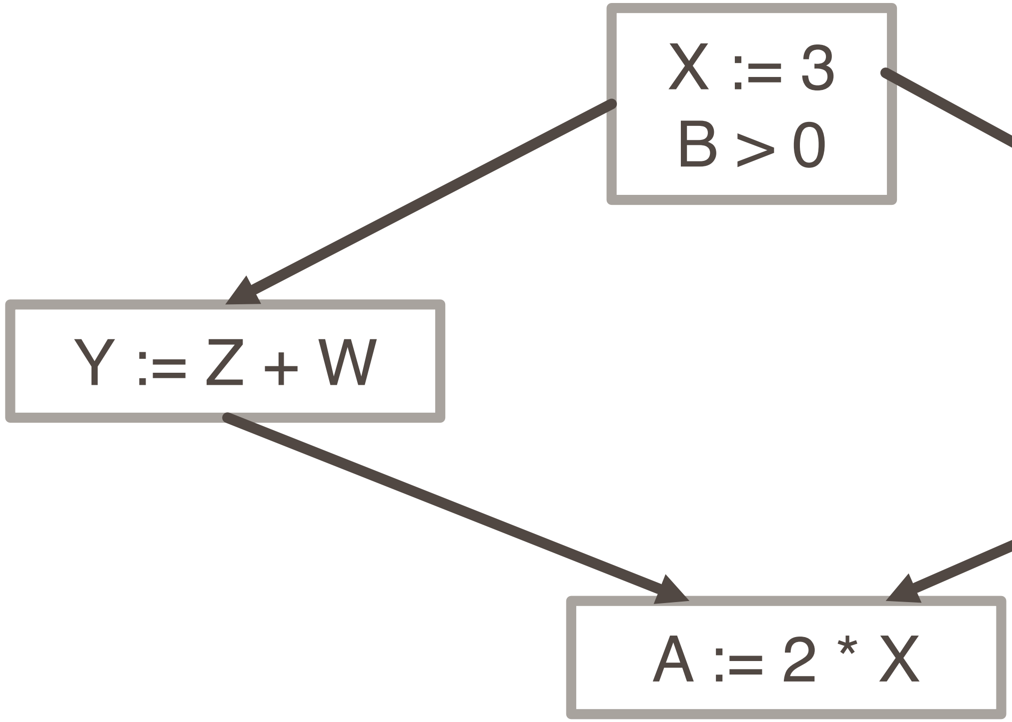
a := x * x	a := x * x
b := 3	
c := x	
d := a	
e := 6	
f := a + a	f := a + a
g := 6 * f	g := 6 * f

Global Optimization:

- Constant Propagation
- Dead-code elimination
- Liveness analysis
- Common subexpression elimination
- Loop optimization

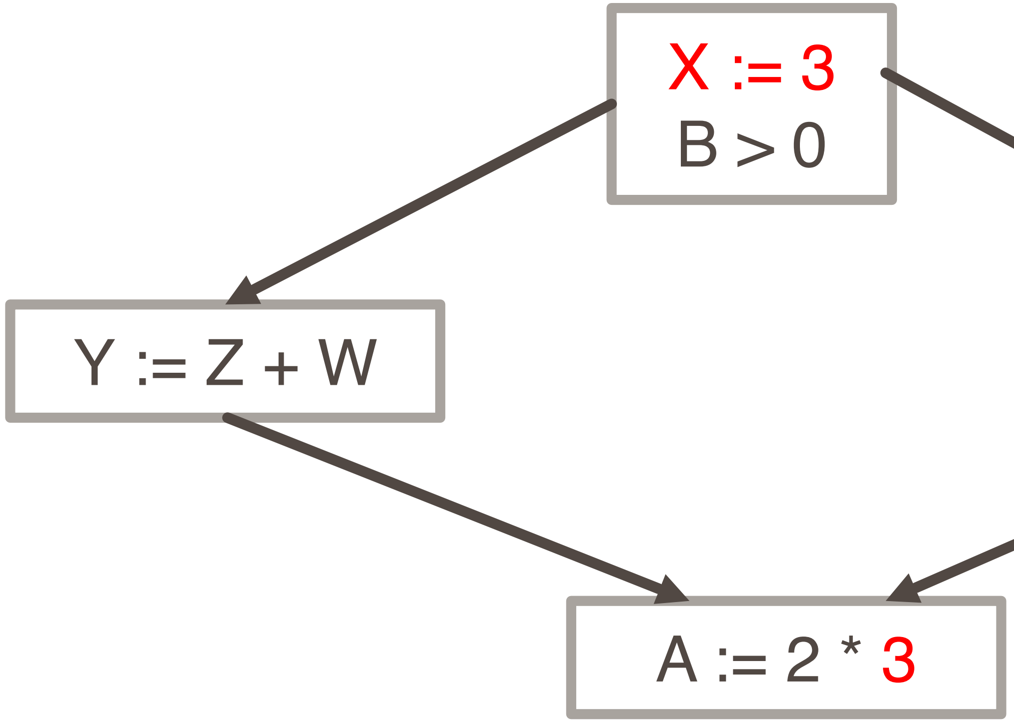
Global Optimization

- These optimizations can be extended to an entire control-flow graph



Global Optimization

- These optimizations can be extended to an entire control-flow graph



Common subexpression elimination

- Example:

$$\begin{array}{lcl} a := b + c & & a := b + c \\ c := b + c & \Rightarrow & c := a \\ d := b + c & & d := b + c \end{array}$$

- Example in array index calculations
  - c[i+1] := a[i+1] + b[i+1]
  - During address computation, i+1 should be reused
  - Not visible in high level code, but in intermediate code

Code Elimination

- Unreachable code elimination
  - Construct the control flow graph
  - Unreachable code block will not have an incoming edge
  - After constant propagation/folding, unreachable branches can be eliminated
- Dead code elimination
  - Ineffective statements

x := y + 1		(immediately redefined, eliminate!)
y := 5	$\Rightarrow$	y := 5
x := 2 * z		x := 2 * z

  - A variable is dead if it is never used after last definition
    - Eliminate assignments to dead variables
  - Need to do data flow analysis to find dead variables

Function Optimization

- Function inlining
  - Replace a function call with the body of the function
  - Save a lot of copying of the parameters, return address, etc.
- Function cloning
  - Create specialized code for a function for different calling parameters

Loop Optimization

- Loop optimization
  - Consumes 90% of the execution time

    $\Rightarrow$ a larger payoff to optimize the code within a loop
- Techniques
  - Loop invariant detection and code motion
  - Induction variable elimination
  - Strength reduction in loops
  - Loop unrolling
  - Loop peeling
  - Loop fusion

Loop Optimization

- Loop invariant detection
  - If the result of a statement or expression does not change within a loop, and it has no external side-effect
  - Computation can be moved to outside of the loop
  - Example

    ```
    for (i=0; i<n; i++) a[i] := a[i] + x/y;
    ```

    - Three address code

      ```
      for (i=0; i<n; i++) { c := x/y; a[i] := a[i] + c; }
      ```

    $\Rightarrow$ c := x/y;

    ```
    for (i=0; i<n; i++) a[i] := a[i] + c;
    ```

Loop Optimization

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Loop Optimization

- Strength reduction in loops
  - Replace costly operation with simpler one
  - Shift, add instead of multiply or divide

```
16*x   -->    x << 4
```

    - Depends on cost of multiply or divide instruction
  - Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Loop Optimization

- Strength reduction in loops
  - Replace costly operation with simpler one
  - Shift, add instead of multiply or divide

    ```
    16*x   -->    x << 4
    ```

    - Depends on cost of multiply or divide instruction
  - Recognize sequence of products

```
s := 0;
for (i=0; i<n; i++)
{
  v := 4 * i;
  s := s + v;
}
```

→

```
s := 0;
for (i=0; i<n; i++)
{
   v := v + 4;
   s := s + v;
}
```

Loop Optimization

- Induction variable elimination
  - If there are multiple induction variables in a loop, can eliminate the ones which are used only in the test condition
  - Example

    s := 0; for (i=0; i<n; i++) { s := 4 * i; … }   -- i is not referenced in loop

    ⇒ s := 0; e := 4*n; while (s < e) { s := s + 4; }

```
s := 0;
for (i=0; i<n; i++)
{ s := 4 * i; … }
-- i is not referenced in
loop
```

```
s := 0;
e := 4*n;
while (s < e) {
 s := s + 4;
}
```

Code Optimization Techniques

- Loop unrolling
  - Execute loop body multiple times at each iteration
  - Get rid of the conditional branches, if possible
  - Allow optimization to cross multiple iterations of the loop
    - Especially for parallel instruction execution
  - Space time tradeoff
    - Increase in code size, reduce some instructions
- Loop peeling
  - Similar to unrolling
  - But unroll the first and/or last few iterations

Loop Optimization

- Loop fusion
  - Example

```
for i=1 to N do
        A[i] = B[i] + 1
endfor
for i=1 to N do
        C[i] = A[i] / 2
endfor
for i=1 to N do
        D[i] = 1 / C[i+1]
endfor
```

Before Loop Fusion

```
for i=1 to N do
        A[i] = B[i] + 1
        C[i] = A[i] / 2
        D[i] = 1 /
  C[i+1]
endfor
```

Loop Optimization

- Loop fusion
  - Example

```
for i=1 to N do
        A[i] = B[i] + 1
endfor
for i=1 to N do
        C[i] = A[i] / 2
endfor
for i=1 to N do
        D[i] = 1 / C[i+1]
endfor
```

Before Loop Fusion

```
for i=1 to N do
        A[i] = B[i] + 1
        C[i] = A[i] / 2
        D[i] = 1 /C[i+1]
endfor
```

Is this correct?
Actually, cannot fuse the third loop

Limitations of Compiler Optimization

- Operate Under Fundamental Constraint
  - Must not cause any change in program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative