

# Global Optimizations

# Types of optimizations

Classification of optimization (based on their scope)

- Local (within basic blocks)
- Global Intra-procedural
- Global Inter-procedural

Classification based on their positioning:

- High level optimizations (use the program structure to optimize).
- Low level optimizations (work on medium/lower level IR)

# Optimization classification (contd)

Classification with respect to their dependence on the target machine.

## Machine independent

- applicable across broad range of machines
- Examples
  - move evaluation to a less frequently executed place
  - remove redundant (unreachable, useless) code.
- create opportunities.

## Machine dependent

- capitalize on machine-specific properties
- improve mapping from IR onto machine
- strength reduction.
- replace sequence of instructions with more powerful one (use “exotic” instructions)

## Desirable properties of an optimizing compiler

- code at least as good as an assembler programmer
- stable, robust performance (predictability)
- architectural strengths fully exploited
- architectural weaknesses fully hidden
- broad, efficient support for language features
- instantaneous compilation

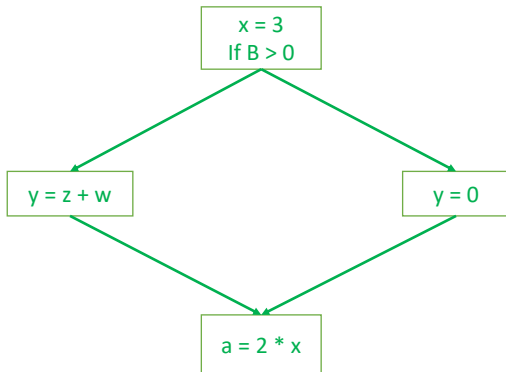
# From local to global optimization

- Recall the local basic-block optimizations
  - Constant propagation
  - Dead code elimination

$x = 3$		$x = 3$		$y = z * w$
$y = z * w$	$-->$	$y = z * w$	$-->$	$q = 3 + y$
$q = x + y$		$q = 3 + y$		

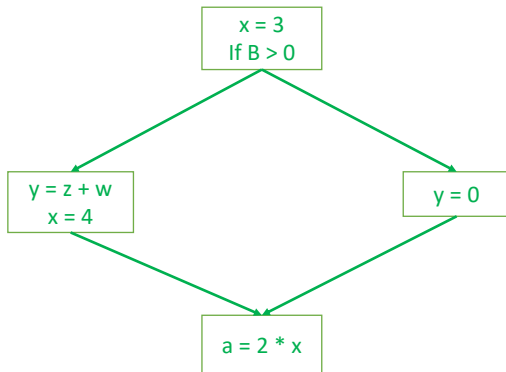
# From local to global optimization

Can these optimizations be directly extended to an entire control-flow graph?



# From local to global optimization

Can these optimizations be directly extended to an entire control-flow graph? There are situations where it is incorrect to globally propagate constants:



# From local to global optimization: Constant Propagation

## Correctness Criterion

To replace a use of  $x$  by constant  $k$ , on every path to the use of  $x$ , the last assignment to  $x$  is  $x = k$ .

- This correctness criterion is non-trivial to check.
- ‘Every path’ includes paths around loops and through branches of conditionals.
- This requires a ‘global’ analysis, i.e. an analysis of the entire control-flow graph.



# Global Analysis

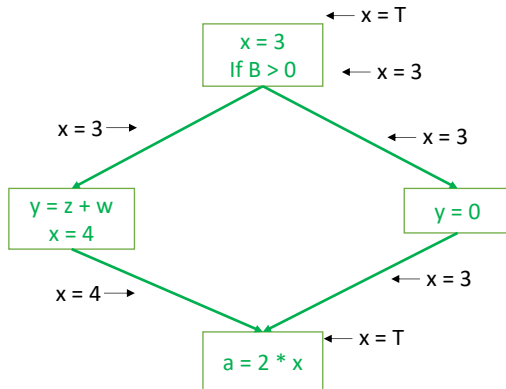
Global optimization tasks share several traits:

- The optimization depends on knowing some property  $X$  at a particular point in program execution.
- Proving  $X$  at any point requires knowledge of the entire function.
- It is OK to be conservative. If the optimization requires  $X$  to be true, then the analysis can output one of two things:
  - $X$  is definitely true
  - Don't know if  $X$  is true.
- Global dataflow analysis is a standard technique for performing global optimizations.

# Global Constant Propagation

- To perform global constant propagation at a program point, we need to know whether a variable will always have a constant value at the point.
- We associate one of the following values with variable  $x$  at every program point:
  - $\perp$ : This program point is not reachable.
  - $c$ :  $x$  has a constant value  $c$
  - $\top$ :  $x$  is not a constant

# Example



# Defining Global Constant Propagation Analysis

- We associate two functions  $in, out : Var \rightarrow \mathbb{Z} \cup \{\top, \perp\}$  with each basic block.
- Global Constant Propagation Analysis is a forward analysis:  $in$  of a basic block is defined in terms of  $out$  of predecessors.
- For a basic block  $b$ ,  $In_b$  is defined as follows:

$$in_b(x) = \begin{cases} \top & \exists p \in Pred(b). out_p(x) = \top \\ c & \forall p \in Pred(b). out_p(x) = c \vee out_p(x) = \perp \\ \top & \exists p_1, p_2 \in Pred(b). out_{p_1}(x) \neq out_{p_2}(x) \\ \perp & \forall p \in Pred(b). out_p(x) = \perp \end{cases}$$

The above definition is also called a meet operation.

We use the following notation for the above definition:

$$in_b = \bigvee_{p \in Pred(b)} out_p$$

# Defining Global Constant Propagation Analysis

$out_b$  is defined in terms of  $in_b$  and the statements in basic block  $b$   
For simplicity, assume that we have a separate basic block for each statement:

$$out_b(x) = \begin{cases} \perp & in_b(x) = \perp \\ c & b : x = c \text{ where } c \in \mathbb{Z} \\ \top & b : x = e \text{ where } e \text{ is an expression} \\ In_b(x) & b : y = \dots \end{cases}$$

The above definition is also called a transfer function.

We can express the definition as a function  $f_b$  such that  $f_b(in_b) = out_b$ .

# Defining Global Constant Propagation Analysis

$$out_b(x) = \begin{cases} \perp & in_b(x) = \perp \\ c & b : x = c \text{ where } c \in \mathbb{Z} \\ e[In_b] & b : x = e \text{ where } e \text{ is an expression} \\ In_b(x) & b : y = \dots \end{cases}$$

We can also do constant folding while evaluating expressions.

Given a function  $f : Var \rightarrow \mathbb{Z} \cup \{\top, \perp\}$ ,  $e[f]$  denotes the evaluation of expression  $e$  using function  $f$ .

While evaluating,  $\top + c = \top$  (similar for other arithmetic operators).

# Iterative method for computing *In*, *Out*

$N$  : Set of nodes of CFG;

$Start$  : Entry basic blocks of CFG (i.e. successors of `entry`);

**foreach**  $\underline{n \in Start}$  **do**

$in_n \leftarrow \lambda v. \top$ ;

**end**

**foreach**  $\underline{n \in N - Start}$  **do**

$in_n \leftarrow \lambda v. \perp$ ;

$out_n \leftarrow \lambda v. \perp$ ;

**end**

**repeat**

**foreach**  $\underline{n \in Nodes}$  **do**

$in'_n \leftarrow in_n$ ;

$out'_n \leftarrow out_n$ ;

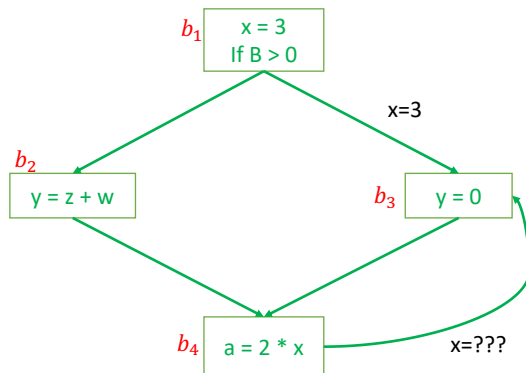
$in_n \leftarrow \bigvee_{p \in Pred(n)} out_p$ ;

$out_n \leftarrow f_n(in_n)$ ;

**end**

**until**  $\underline{\forall n, in'_n = in_n \wedge out'_n = out_n}$ ;

# Why do we need $\perp$ ?



- To compute  $in_{b_3}$ , we need  $out_{b_4}$ , but for that we need  $out_{b_3}$ !
- We will encounter similar problems whenever we have loops.
- Hence, we initialize  $in$  and  $out$  values with  $\perp$ , which intuitively means that 'so far as we know, control never reaches this point'



# Orderings

- We can simplify the presentation of the data-flow analysis by ordering the values.
  - $\forall c \in \mathbb{Z}. \top \leq c \leq \perp$
  - Formally,  $\leq$  is a partial order on the set  $\mathbb{Z} \cup \{\perp, \top\}$ .
- We can define the greatest lower bound of a set of values.
  - Formally,  $(\mathbb{Z} \cup \{\perp, \top\}, \leq)$  forms a meet semi-lattice, and hence the *glb* always exists for any set of values.

$$in_b(x) = \begin{cases} \top & \exists p \in Pred(b). out_p(x) = \top \\ \top & \exists p_1, p_2 \in Pred(b). out_{p_1}(x) \neq out_{p_2}(x) \\ \perp & \forall p \in Pred(b). out_p(x) = \perp \\ c & \forall p \in Pred(b). out_p(x) = c \vee out_p(x) = \perp \end{cases}$$

- Notice that  $in_b(x) = glb(\{out_p(x) \mid p \in Pred(b)\})$ .
  - The greatest lower bound is also called **meet**.

# Orderings

- Every data-flow analysis can be represented by defining its meet semi-lattice, with the corresponding meet operation being used in the iterative method.
  - Useful for proving the soundness of the analysis, for comparing precision of different analyses, and for proving termination of the iterative method.
  - Dataflow analysis/Abstract Interpretation covered in detail in advanced courses: CS5030, CS6013.
- Termination argument: We start with the highest value ( $\perp$ ) and we only move down.
  - $\perp$  can change to a constant value, which can change to  $\top$ .
  - Thus, each  $in(x)$  or  $out(x)$  can change at most twice at any basic block.
  - Maximum number of iterations =  $2 * 2 * \text{Number of variables} * \text{Number of basic blocks}$ .

# Liveness Analysis... revisited

- We can represent liveness analysis in the dataflow analysis framework.
- Let  $Var$  be the set of variables. Then, the meet semi-lattice would be  $(\mathbb{P}(Var), \supseteq)$ .
  - The  $glb$  operation is set union.
  - The analysis works in the backward direction. Hence,
$$out_b = \bigcup_{s \in succ(b)} in_s.$$
- The transfer function is  $f_b(S) = use_b \cup (S - def_b)$ .
  - $use_b$  are variables which are used before they are (possibly) defined in  $b$ . Can be determined using the next-use algorithm.
  - $def_b$  are variables which are defined in  $b$ .

# Types of program analysis

## Classification of analysis (based on their view)

```
if (cond) {  
    a = ...  
    b = ...  
} else {  
    a = ...  
    c = ...  
}  
  
// Which of the variables may be assigned? -- {a,b,c}  
// Which of the variables must be assigned? -- {a}
```

- May analysis – the analysis holds on at least one data flow path.
- Must analysis – the analysis must hold on all data flow paths.
  - What can we say about constant propagation analysis? May or Must?
  - What can we say about liveness analysis? May or Must?

# Classification of analysis (contd)

## Classification of analysis (based on precision)

- Flow sensitive / insensitive.
  - Insensitive - the analysis should hold at every program point; does not depend on the control flow.
  - Sensitive - Each program point has its own analysis.

```
if (c) {  
    a = 2;  
    b = a;  
    c = 3;  
    print (a, b, c); // constants?  
} else {  
    a = 3;  
    b = a;  
    c = 3;  
    print (a, b, c); // constants?  
}
```

# Classification of analysis (contd)

- Context sensitive and insensitive

```
a = foo(2);
```

```
b = foo (3);
```

```
c = bar (2);
```

```
d = bar(2);
```

```
print (a, b, c, d); // a, b, c, d constants?
```

```
int foo(int x) { return x }
```

```
int bar(int x) { return x * x }
```

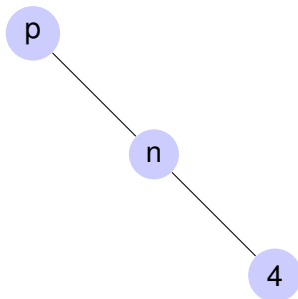
# Alias Analysis

**Alias analysis:** problem of identifying storage locations that can be accessed by more than one way.

Are variable `a` and `b` aliases?  $\Rightarrow$  `a` and `b` refer to the same location?  
Modifying the contents of `a`, modifies the contents of `b`.

Necessary for performing many optimizations such as constant/copy propagation, common sub-expression elimination, dead code elimination, etc.

```
foo() {  
    int *p;  
    int n;  
    p = &n;  
    n = 4;  
    print ("%d", *p);  
}
```



## Alias analysis (contd)

```
extern int *q;  
foo( ) {  
    int a, k;  
    k = a + 5;  
    f (a, &k);  
    *q = 13;  
    k = a + 5;    /* Assignment is redundant? */  
                  /* Expression is redundant? */  
    ...  
}
```

Only if a) the assignment to  $*q$  does not change  $k$  or  $a$ ,  
b) the function call  $f$ , does not change  $*k$ .



# Loop unrolling

## (Example) Matrix-matrix multiply

```
do i ← 1, n, 1
  do j ← 1, n, 1
    c(i,j) ← 0
    do k ← 1, n, 1
      c(i,j) ← c(i,j) + a(i,k) * b(k,j)
```

- All the array elements are floating point values.
- $2n^3$  flops,  $n^3$  loop increments and branches
- each iteration does 2 loads and 2 flops

# Example: loop unrolling

## Matrix-matrix multiply

(assume 4-word cache line)

```
do i ← 1, n, 1
  do j ← 1, n, 1
    c(i, j) ← 0
    do k ← 1, n, 4
      c(i, j) ← c(i, j) + a(i, k) * b(k, j)
      c(i, j) ← c(i, j) + a(i, k+1) * b(k+1, j)
      c(i, j) ← c(i, j) + a(i, k+2) * b(k+2, j)
      c(i, j) ← c(i, j) + a(i, k+3) * b(k+3, j)
```

- $2n^3$  flops,  $\frac{n^3}{4}$  loop increments and branches
- each iteration does 8 loads and 8 flops
- memory traffic is better
  - $c(i, j)$  is reused
  - $a(i, k)$  reference are from cache
  - $b(k, j)$  is problematic

(put it in a register)

# Example: loop unrolling

Matrix-matrix multiply

(to improve traffic on b)

```
do j ← 1, n, 1
  do i ← 1, n, 4
    c(i,j) ← 0
    do k ← 1, n, 4
      c(i,j) ← c(i,j) + a(i,k) * b(k,j)
        + a(i,k+1) * b(k+1,j) + a(i,k+2) * b(k+2,j)
        + a(i,k+3) * b(k+3,j)
      c(i+1,j) ← c(i+1,j) + a(i+1,k) * b(k,j)
        + a(i+1,k+1) * b(k+1,j)
        + a(i+1,k+2) * b(k+2,j)
        + a(i+1,k+3) * b(k+3,j)
      c(i+2,j) ← c(i+2,j) + a(i+2,k) * b(k,j)
        + a(i+2,k+1) * b(k+1,j)
        + a(i+2,k+2) * b(k+2,j)
        + a(i+2,k+3) * b(k+3,j)
      c(i+3,j) ← c(i+3,j) + a(i+3,k) * b(k,j)
        + a(i+3,k+1) * b(k+1,j)
        + a(i+3,k+2) * b(k+2,j)
        + a(i+3,k+3) * b(k+3,j)
```

# Example: loop unrolling

What happened?

- interchanged  $i$  and  $j$  loops
- unrolled  $i$  loop
- fused inner loops
- $2n^3$  flops,  $\frac{n^3}{16}$  loop increments and branches
- first assignment does 8 loads and 8 flops
- 2<sup>nd</sup> through 4<sup>th</sup> do 4 loads and 8 flops
- memory traffic is better
  - $c(i, j)$  is reused
  - $a(i, k)$  references are from cache
  - $b(k, j)$  is reused

(register)

(register)

# Loop optimizations: factoring loop-invariants

Loop invariants: expressions constant within loop body

Goal: move the loop invariant computation to outside the loop.

The loop independent code executes only once, instead of many times the loop might.

## Example: loop invariants

```
foreach i=1 .. 100 do
  foreach j=1 .. 100 do
    foreach k=1 .. 100 do
      | A[i, j, k] = i * j * k;
    end
  end
end
```

- 3 million index operations
- 2 million multiplications

## Example: loop invariants (cont.)

Factoring the inner loop:

```
foreach i=1 .. 100 do  
  | foreach j=1 .. 100 do  
  |   | t1 = &A[i][j];  
  |   | t2 = i * j ;  
  |   | foreach k=1 .. 100 do  
  |   |   | t1[k] = t2 * k;  
  |   | end  
  | end  
end
```

And the second loop:

```
foreach i=1 .. 100 do  
  | t3 = &A[i];  
  | foreach j=1 .. 100 do  
  |   | t1 = &t3[j];  
  |   | t2 = i * j ;  
  |   | foreach k=1 .. 100 do  
  |   |   | t1[k] = t2 * k;  
  |   | end  
  | end  
end
```

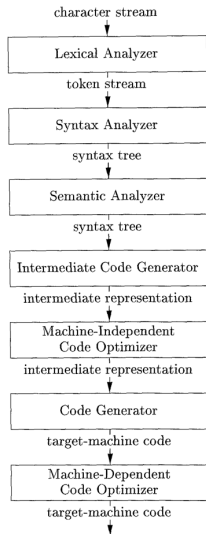
## Compilers are engineered objects

- minimize running time of compiled code
- minimize compile time
- use reasonable compile-time space
- find a reasonable trade-off

Thus, results are sometimes unexpected



# Back to first lecture



## Front end responsibilities:

- Recognize syntactically legal code; report errors.
- Recognize semantically legal code; report errors.
- Produce IR.

## Back end responsibilities:

- Optimizations, code generation.

## Our target

- five out of seven phases.
- glance over optimizations.