

# Tour of common optimizations

---

# Simple example

---

*point*

```
foo(z) {  
  x := 3 + 6;  
  y := x - 5 4  
  return z * y 4    z << z  
}
```

# Simple example

```
foo(z) {
```

```
  x := 3 + 6; 9
```

Constant folding (CF)

Const prop (CP)

```
  y := x - 5 4 (CF)
```

```
  return z * y 4 (CP)
```

$z \ll 2$



Strength reduction

Arith  
simpl

# Another example

---

`x := a + b;`

`...`

`y := a + b; x`

# Another example

---

**x := a + b;**

...

**y := ~~a + b~~; x**

} only if x, a, b not modified!

# Another example

---

```
if (...) {  
  a := read();  
  x' := a + b; x := x';  
  print(x);  
} else { x' := a + b }
```

```
...  
print(y)  
y := a + b; x'
```

# Another example

---

```
if (...) {  
  a := read(); t := a + b  
  x := a + b; t  
  print(x);  
} else { t := a + b }
```

...

```
y := a + b; t
```

Partial Redundancy  
Elimination PRE

# Another example

---

**x := y**

**...**

**z := z + x**



# Another example

---

**x := y**

...

**z := z + ~~x~~ y**

} x, y not modified  
copy prop

# Another example

---

$\left[ \begin{array}{l} \mathbf{x} := \mathbf{y} \\ \dots \\ \mathbf{z} := \mathbf{z} + \cancel{\mathbf{y}} \cancel{\mathbf{x}} \end{array} \right.$

What if we run CSE now?

$\mathbf{x} := \mathbf{E}$

$\dots \left[ \begin{array}{l} \dots \\ \mathbf{x} \end{array} \right] \dots$

# Another example

---

**x** := **y**

...

**z** := **z** + ~~**y**~~ **X**

What if we run CSE now?

# Another example

---

**x := y\*\*z**

**...**

**x := ...**

# Another example

~~$x := y * z$~~

...

$x := \dots$

} if  $x$  is not used  
dead assignment elim  
(unused assignment elim)

- Often used as a clean-up pass

$x := y$   
 $z := z + x$       Copy prop       $x := y$   
 $z := z + y$       DAE       ~~$x := y$~~   
 $z := z + y$

# Another example

---

```
if (false) {
```

```
    ...
```

```
}
```

# Another example

---

```
if (false) {  
    ...  
}
```

dead code elim  
( unreachable code elim)

Another common clean up opt

# Another example

---

- In Java:

```
a = new int [10];  
for (index = 0; index < 10; index ++) {  
    a[index] = 100;  
}
```



# Another example

---

- In “lowered” Java:

```
[a = new int [10];] a.length = 10
for (index = 0; index < 10; index ++ ) {
  if ((index < 0 || index >= a.length)) {
    throw OutOfBoundsException;
  }
  a[index] = 0;
}
```

*index [0..9]*

*False*

*10*

# Another example

- In “lowered” Java:

```
a = new int [10]; ①
for (index = 0; index < 10; index ++) {
  if (index < 0 || index >= a.length()) {
    throw OutOfBoundsException;
  }
  a[index] = 0;
}
```

Branch folding  
+ unreachable  
code elim

10 ← Kinda like CP  
if we assume  
stmt ① acts  
like a.length := 10

index ∈ [0..9] ← Range analysis

# Another example

---


```
p := &x;  
*p := 5  
y := x + 1;  
      5
```

# Another example

---

```
p := &x;  
x *p := 5  
y := x + 1; 6  
5
```

*pointer / alias analysis*

```
x := 5;  
*p := 3  
y := x + 1;  ???  
5
```

# Another example

---

```
for j := 1 to N
  for i := 1 to M
    a[i] := a[i] + b[j]
```

Handwritten annotations:  $t = b[j]$  above the inner loop, and underlines under  $a[i]$  and  $b[j]$  in the assignment statement. A red asterisk is next to  $b[j]$ .

$$t_1 = a + i * 4$$

$$t_2 = *t_1$$

~~$$t_3 = a + i * 4 * t_1$$~~

$$*t_1 = t_2 + t$$

# Another example

---

```
for j := 1 to N
  for i := 1 to M
    a[i] := a[i] + b[j] t
```

*t := b[j]*

*Loop invariant  
code motion*

# Another example

---

```
area(h,w) { return h * w }
```

```
h := ...;
```

```
w := 4;
```

```
a := area(h,w)
```

$h * w$

$h * 4$

$h \ll 2$

# Another example

---

```
area(h,w) { return h * w }
```

```
h := ...;
```

```
w := 4;
```

```
a := area(h,w)
```

~~$h * w$~~

~~$h * 4$~~

$h << 2$

Many "silly" opts became  
important after inlining



# Optimization themes

---

- Don't compute if you don't have to
  - unused assignment elimination
- Compute at compile-time if possible
  - constant folding, loop unrolling, inlining
- Compute it as few times as possible
  - CSE, PRE, PDE, loop invariant code motion
- Compute it as cheaply as possible
  - strength reduction
- Enable other optimizations
  - constant and copy prop, pointer analysis
- Compute it with as little code space as possible
  - unreachable code elimination