IR Optimization

Ronghui Gu
Spring 2020
Columbia University

* Course website: https://www.cs.columbia.edu/rgu/courses/4115/spring2019
IR Optimization

int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

IR Optimization

Code Generation

0101110101...
IR Optimization

Goal

• Runtime
• Memory usage
• Power Consumption

Sources?
Optimizations from IR Generation

C code:

```c
int x;
int y;
bool b1;
bool b2;
bool b3;
b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

Three-Address:

```c
_to = x + x;
_t1 = y;
b1 = _to < _t1;
_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;
_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```
Optimizations from IR Generation

C code:

```c
int x;
int y;
bool b1;
bool b2;
bool b3;
b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

Three-Address:

```c
_to = x + x;
_t1 = y;
b1 = _to < _t1;
_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;
_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```
Optimizations from IR Generation

C code:

```c
int x;
int y;
bool b1;
bool b2;
bool b3;
b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

Three-Address:

```c
_to = x + x;
_t1 = y;
b1 = _to < _t1;

b2 = _to == _t1;

b3 = _to < _t1;
```
C code:

```c
while (x < y + z) {
    x = x - y;
}
```

Three-Address:

```assembly
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    bz _L1 _t1;
    x = x - y;
    jmp _L0;
_L1:
```
C code:

```c
while (x < y + z) {
    x = x - y;
}
```

Three-Address:

```assembly
_L0:
   _to = y + z;
   _t1 = x < _to;
   bz _L1 _t1;
   x = x - y;
   jmp _L0;
_L1:
```
C code:

```c
while (x < y + z) {
    x = x - y;
}
```

Three-Address:

```
_t0 = y + z;
_L0:
_t1 = x < _t0;
bz _L1 _t1;
x = x - y;
jmp _L0;
_L1:
```
IR Optimization Discussion

Optimal? Undecidable!

Soundness: semantics-preserving

IR optimization v.s. code optimization:

\[ x \times 0.5 \Rightarrow x \gg 1 \]

Local optimization v.s. global optimization
```c
int main() {
    int y;
    int z;
    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```

```
to = 137;
y = to;
bz Lo x;
```

```
t1 = y;
z = t1;
```

```
t2 = y;
x = t2;
```

```
END:
```
int main() {
    int y;
    int z;
    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```c
int main() {
    int y;
    int z;
    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```
int main() {
    int y;
    int z;
    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
int main() {
    int y;
    int z;
    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
int main() {
    int y;
    int z;
    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
Local Optimization
Common Subexpression Elimination

\[ v_1 = a \ op \ b \]
\[
. \ . \ .
\]
\[ v_2 = a \ op \ b \]

If values of \( v_1, a, \) and \( b \) have not changed, rewrite the code:

\[ v_1 = a \ op \ b \]
\[
. \ . \ .
\]
\[ v_2 = v_1 \]
Common Subexpression Elimination

C code:
```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:
```c
_to = 4;
a = _to;
t1 = a + b;
c = t1;
t2 = a + b;
param _t2
call f;
```
Common Subexpression Elimination

C code:
```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:
```c
_to = 4;
a = _to;
_t1 = a + b;
c = _t1;
_t2 = a + b;
param _t2
call f;
```
C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```c
_to = 4;
a = _to;
_t1 = a + b;
c = _t1;
_t2 = _t1;
param _t2
call f;
```
Common Subexpression Elimination

C code:
```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:
```c
_to = 4;
a = _t0;
_t1 = a + b;
c = _t1;
_t2 = c;
param _t2
call f;
```
Copy Propagation

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```c
_to = 4;
a = _to;
t1 = a + b;
c = t1;
t2 = c;
param _t2
call f;
```
Copy Propagation

If we have

\[ v_1 = v_2 \]

then as long as \( v_1 \) and \( v_2 \) have not changed, we can rewrite

\[ a = \ldots \ v_1 \ldots \]

as

\[ a = \ldots \ v_2 \ldots \]
Copy Propagation

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```c
_to = 4;
a = 4;
_t1 = a + b;
c = _t1;
_t2 = c;
param _t2
call f;
```
Copy Propagation

C code:
```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:
```plaintext
_t0 = 4;
a = 4;
t1 = a + b;
c = t1;
t2 = c;
param _t2
call f;
```
Copy Propagation

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```plaintext
_to = 4;
a = 4;
_t1 = 4 + b;
c = _t1;
_t2 = c;
param _t2
call f;
```
Copy Propagation

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```c
_to = 4;
a = 4;
_t1 = 4 + b;
c = _t1;
_t2 = c;
param _t2
call f;
```
Copy Propagation

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```c
_to = 4;
a = 4;
t1 = 4 + b;
c = 4 + b;
t2 = c;
param _t2
call f;
```
Copy Propagation

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```plaintext
_to = 4;
a = 4;
_t1 = 4 + b;
c = 4 + b;
_t2 = c;
param _t2
call f;
```
Copy Propagation

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```c
_to = 4;
a = 4;
_t1 = 4 + b;
c = 4 + b;
_t2 = 4 + b;
param _t2
call f;
```
An assignment to a variable \( v \) is called \textit{dead} if its value is \textit{never} read anywhere.
Dead Code Elimination

C code:
```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:
```c
_to = 4;
a = 4;
t1 = 4 + b;
c = 4 + b;
t2 = 4 + b;
param _t2
call f;
```
Dead Code Elimination

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```plaintext
_to = 4;
a = 4;
t1 = 4 + b;
c = 4 + b;
t2 = 4 + b;
param t2
call f;
```
Dead Code Elimination

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```c
a = 4;
_t1 = 4 + b;
c = 4 + b;
_t2 = 4 + b;
param _t2
call f;
```
Dead Code Elimination

C code:
```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:
```
_t1 = 4 + b;
c = 4 + b;
_t2 = 4 + b;
param _t2
call f;
```
Dead Code Elimination

C code:
```
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:
```
c = 4 + b;
_t2 = 4 + b;
param _t2
call f;
```
Dead Code Elimination

C code:

```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:

```plaintext
_t2 = 4 + b;
param _t2
call f;
```
For Comparison

C code:
```c
int a;
int b;
int c;
a = 4;
c = a + b;
f(a + b);
```

Three-address code:
```c
_to = 4;
a = _to;
_t1 = a + b;
c = _t1;
_t2 = a + b;
param _t2
call f;
```

Optimized code:
```c
_t2 = 4 + b;
param _t2
call f;
```
Other Types of Local Optimization

Arithmetic simplication:

• e.g., rewrite \( x = 4 \times a \) as \( x = a \ll 2 \)

Constant folding:

• e.g., rewrite \( x = 4 \times 5 \) as \( x = 20 \)
Global Optimization
Global Constant Propagation

START:

a = 6;

b = a;
c = b;

END: d = a
Global Optimization
Global Constant Propagation

Replace each variable that is known to be a constant value with the constant.
Global Constant Propagation

START:

\[ a \textit{ equal osf } 6 \textit{ osf}; \]
\[ x \textit{ equal osf } y \textit{ osf}; \]

b = a;

\[ c \textit{ equal osf } b \textit{ osf}; \]

END: d = x + a
Global Constant Propagation

START:

a = 6;

x = y;

b = 6;

c = b;

END: d = x + 6
Global Dead Code Elimination

START:

a = 6;
x = y;

END: d = x + 6
Global Dead Code Elimination

START:

a = 6;
x = y;

b = 6;
c = b;

END: d = x + 6
Global Dead Code Elimination

START:

\[ x = y; \]

END: \[ d = x + 6 \]