IR Optimization

Ronghui Gu
Spring 2019
Columbia University

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

```plaintext
_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:

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

Three-Address:

_L0:
  _to = y + z;
  _t1 = x < _to;
  bz _L1 _t1;
  x = x - y;
  jmp _L0;
_L1:
Optimizations from Lazy Coders

C code:

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

Three-Address:

```assembly
_t0 = y + z;
_L0:
    _t1 = x < _t0;
    bz _L1 _t1;
    x = x - y;
    jmp _L0;
_L1:
```
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
int main() {
    int y;
    int z;
    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

START:

t0 = 137;
y = t0;
bez Lo x;

t1 = y;
z = t1;

END:

t2 = y;
x = t2;
int main() {
    int y;
    int z;
    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

START:
    to = 137;
    y = to;
    bz Lo x;

    t1 = y;
    z = t1;

    t2 = y;
    x = t2;

END:
```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

\[ v_1 = a \text{ op } b \]

\[ \ldots \]

\[ v_2 = a \text{ op } b \]

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

\[ v_1 = a \text{ op } b \]

\[ \ldots \]

\[ v_2 = v_1 \]
Common Subexpression Elimination

C code:

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

Three-address code:

```c
_t0 = 4;
a = _t0;
_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
_t0 = 4;
a = _t0;
_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 = t1;
param _t2
call f;
```
Common Subexpression Elimination

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

Three-address code:
```
_to = 4;
a = _to;
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
_t/to = 4;
a = _to;
t1 = a + b;
c = _t1;
t2 = c;
param _t2
call f;
```
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:
```plaintext
_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
_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:

```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 = _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 = 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
_t0 = 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:

```
_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 dead if its value is never read anywhere.
Dead Code Elimination

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

Three-address code:
```
_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:
```c
_t1 = 4 + b;
c = 4 + b;
_t2 = 4 + 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
_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
_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 implication:

- 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:

\[ a \text{ = 6;} \]

\[ b = a; \]

\[ c = b; \]

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:

\[
a = 6; \\
x = y;
\]

END:

\[
d = x + a
\]
Global Constant Propagation

START:

\[ a = 6; \]
\[ x = y; \]

b = 6;  \hspace{1cm} c = b;

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:

a = 6;
x = y;

b = 6;
c = b;

END: d = x + 6
Global Dead Code Elimination

START:

x = y;

END: d = x + 6