

CS738: Advanced Compiler Optimizations

Overview of Optimizations

Amey Karkare

karkare@cse.iitk.ac.in

<http://www.cse.iitk.ac.in/~karkare/cs738>

Department of CSE, IIT Kanpur



Recap

- ▶ Optimizations

Recap

- ▶ Optimizations
 - ▶ To improve efficiency of generated executable (time, space, resources, ...)

Recap

- ▶ Optimizations
 - ▶ To improve efficiency of generated executable (time, space, resources, ...)
 - ▶ Maintain semantic equivalence

Recap

- ▶ Optimizations
 - ▶ To improve efficiency of generated executable (time, space, resources, ...)
 - ▶ Maintain semantic equivalence
- ▶ Two levels

Recap

- ▶ Optimizations
 - ▶ To improve efficiency of generated executable (time, space, resources, ...)
 - ▶ Maintain semantic equivalence
- ▶ Two levels
 - ▶ Machine Independent

Recap

- ▶ Optimizations
 - ▶ To improve efficiency of generated executable (time, space, resources, ...)
 - ▶ Maintain semantic equivalence
- ▶ Two levels
 - ▶ Machine Independent
 - ▶ Machine Dependent

Machine Independent Code Optimizations

Machine Independent Optimizations

- ▶ Scope of optimizations

Machine Independent Optimizations

- ▶ Scope of optimizations
 - ▶ Intraprocedural

Machine Independent Optimizations

- ▶ Scope of optimizations
 - ▶ Intraprocedural
 - ▶ Local

Machine Independent Optimizations

- ▶ Scope of optimizations
 - ▶ Intraprocedural
 - ▶ Local
 - ▶ Global

Machine Independent Optimizations

- ▶ Scope of optimizations
 - ▶ Intraprocedural
 - ▶ Local
 - ▶ Global
 - ▶ Interprocedural

Local Optimizations

- ▶ Restricted to a basic block

Local Optimizations

- ▶ Restricted to a basic block
- ▶ Simplifies the analysis

Local Optimizations

- ▶ Restricted to a basic block
- ▶ Simplifies the analysis
- ▶ Not all optimizations can be applied locally

Local Optimizations

- ▶ Restricted to a basic block
- ▶ Simplifies the analysis
- ▶ Not all optimizations can be applied locally
 - ▶ E.g. Loop optimizations

Local Optimizations

- ▶ Restricted to a basic block
- ▶ Simplifies the analysis
- ▶ Not all optimizations can be applied locally
 - ▶ E.g. Loop optimizations
- ▶ Gains are also limited

Local Optimizations

- ▶ Restricted to a basic block
- ▶ Simplifies the analysis
- ▶ Not all optimizations can be applied locally
 - ▶ E.g. Loop optimizations
- ▶ Gains are also limited
- ▶ Simplify global/interprocedural optimizations

Global Optimizations

- ▶ Typically restricted within a procedure/function

Global Optimizations

- ▶ Typically restricted within a procedure/function
 - ▶ Could be restricted to a smaller scope, e.g. a loop

Global Optimizations

- ▶ Typically restricted within a procedure/function
 - ▶ Could be restricted to a smaller scope, e.g. a loop
- ▶ Most compiler implement up to global optimizations

Global Optimizations

- ▶ Typically restricted within a procedure/function
 - ▶ Could be restricted to a smaller scope, e.g. a loop
- ▶ Most compiler implement up to global optimizations
- ▶ Well founded theory

Global Optimizations

- ▶ Typically restricted within a procedure/function
 - ▶ Could be restricted to a smaller scope, e.g. a loop
- ▶ Most compiler implement up to global optimizations
- ▶ Well founded theory
- ▶ Practical gains

Interprocedural Optimizations

- ▶ Spans multiple procedures, files

Interprocedural Optimizations

- ▶ Spans multiple procedures, files
 - ▶ In some cases multiple languages!

Interprocedural Optimizations

- ▶ Spans multiple procedures, files
 - ▶ In some cases multiple languages!
- ▶ Not as popular as global optimizations

Interprocedural Optimizations

- ▶ Spans multiple procedures, files
 - ▶ In some cases multiple languages!
- ▶ Not as popular as global optimizations
 - ▶ No single theory applicable to all scenarios

Interprocedural Optimizations

- ▶ Spans multiple procedures, files
 - ▶ In some cases multiple languages!
- ▶ Not as popular as global optimizations
 - ▶ No single theory applicable to all scenarios
 - ▶ Time consuming

A Catalog of Code Optimizations

Compile-time Evaluation

- ▶ Move run-time actions to compile-time

Compile-time Evaluation

- ▶ Move run-time actions to compile-time
- ▶ Constant Folding

$$\text{Volume} = \frac{4}{3} \times \pi \times r \times r \times r$$

Compile-time Evaluation

- ▶ Move run-time actions to compile-time
- ▶ Constant Folding

$$\text{Volume} = \frac{4}{3} \times \pi \times r \times r \times r$$

- ▶ Compute $\frac{4}{3} \times \pi$ at compile-time

Compile-time Evaluation

- ▶ Move run-time actions to compile-time
- ▶ Constant Folding

$$\text{Volume} = \frac{4}{3} \times \pi \times r \times r \times r$$

- ▶ Compute $\frac{4}{3} \times \pi$ at compile-time
- ▶ Applied frequently for linearizing indices of multidimensional arrays

Compile-time Evaluation

- ▶ Move run-time actions to compile-time
- ▶ Constant Folding

$$\text{Volume} = \frac{4}{3} \times \pi \times r \times r \times r$$

- ▶ Compute $\frac{4}{3} \times \pi$ at compile-time
 - ▶ Applied frequently for linearizing indices of multidimensional arrays
- ▶ **When should we NOT apply it?**

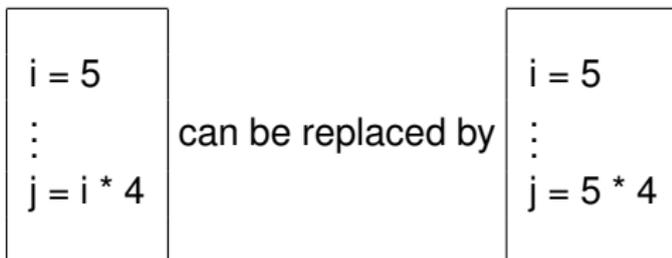
Compile-time Evaluation

- ▶ Constant Propagation
 - ▶ Replace a variable by its “constant” value

```
i = 5
⋮
j = i * 4
```

Compile-time Evaluation

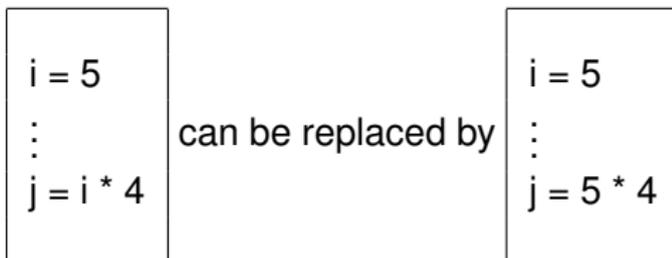
- ▶ Constant Propagation
 - ▶ Replace a variable by its “constant” value



Compile-time Evaluation

- ▶ Constant Propagation

- ▶ Replace a variable by its “constant” value

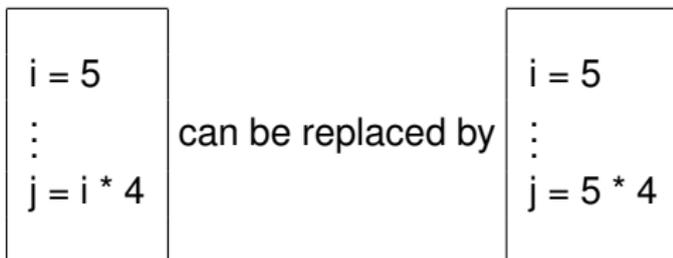


- ▶ May result in the application of constant folding

Compile-time Evaluation

- ▶ Constant Propagation

- ▶ Replace a variable by its “constant” value



- ▶ May result in the application of constant folding

- ▶ **When should we NOT apply it?**

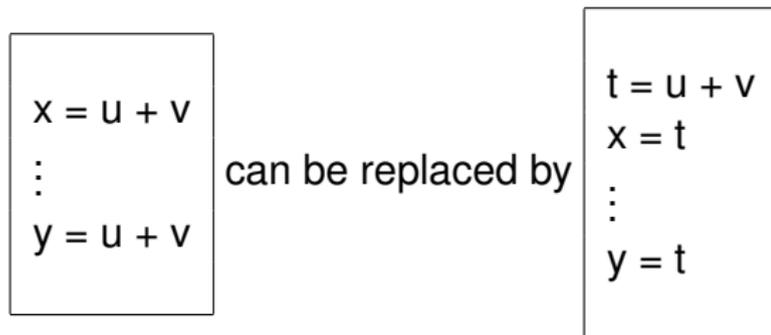
Common Subexpression Elimination

- ▶ Reuse a computation if already “available”

$$\begin{array}{l} x = u + v \\ \vdots \\ y = u + v \end{array}$$

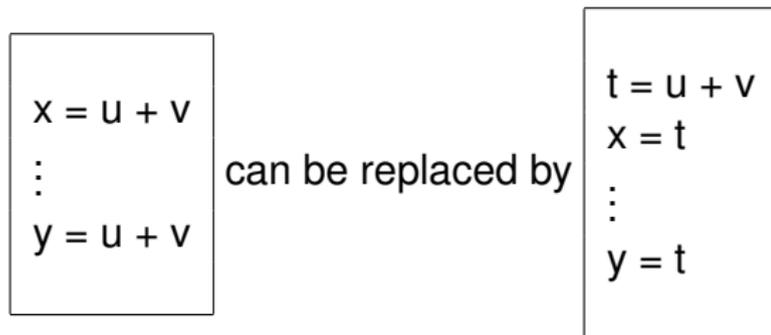
Common Subexpression Elimination

- ▶ Reuse a computation if already “available”



Common Subexpression Elimination

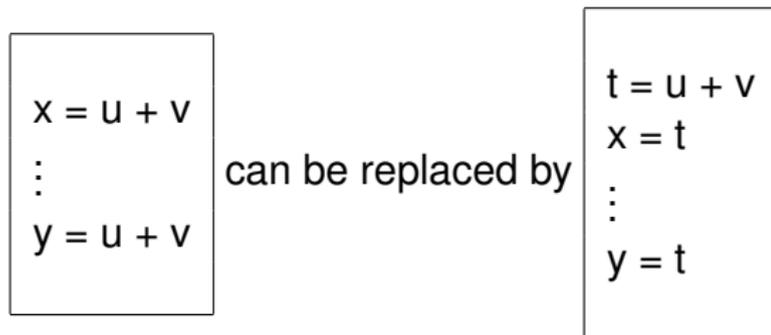
- ▶ Reuse a computation if already “available”



- ▶ How to check if an expression is already available?

Common Subexpression Elimination

- ▶ Reuse a computation if already “available”



- ▶ How to check if an expression is already available?
- ▶ **When should we NOT apply it?**

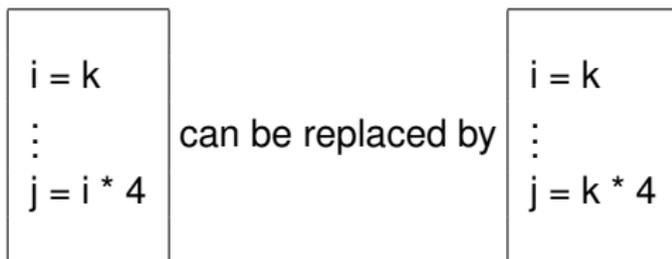
Copy Propagation

- ▶ Replace (use of) a variable by another variable
 - ▶ If they are guaranteed to have the “same value”

```
i = k
⋮
j = i * 4
```

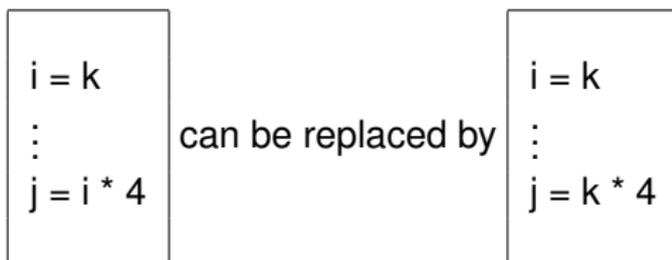
Copy Propagation

- ▶ Replace (use of) a variable by another variable
 - ▶ If they are guaranteed to have the “same value”



Copy Propagation

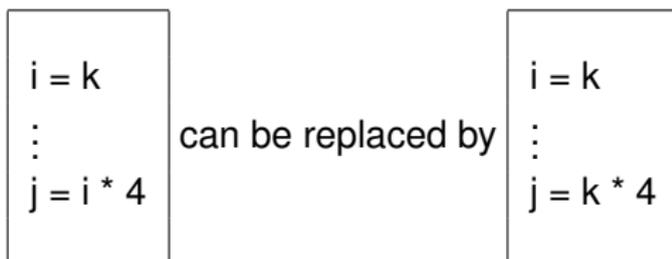
- ▶ Replace (use of) a variable by another variable
 - ▶ If they are guaranteed to have the “same value”



- ▶ May result in dead code, common subexpression

Copy Propagation

- ▶ Replace (use of) a variable by another variable
 - ▶ If they are guaranteed to have the “same value”



- ▶ May result in dead code, common subexpression
- ▶ **When should we NOT apply it?**

Code Movement

- ▶ Move the code around in a program

Code Movement

- ▶ Move the code around in a program
- ▶ Benefits

Code Movement

- ▶ Move the code around in a program
- ▶ Benefits
 - ▶ Code size reduction

Code Movement

- ▶ Move the code around in a program
- ▶ Benefits
 - ▶ Code size reduction
 - ▶ Reduction in the frequency of execution

Code Movement

- ▶ Move the code around in a program
- ▶ Benefits
 - ▶ Code size reduction
 - ▶ Reduction in the frequency of execution
- ▶ How to find out which code to move?

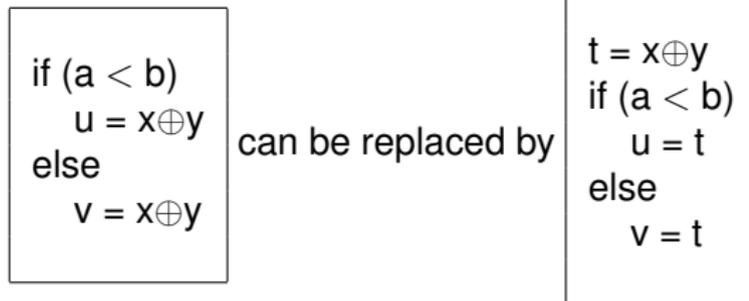
Code Movement

- ▶ Code size reduction
 - ▶ Suppose the operator \oplus results in the generation of a large number of machine instructions. Then,

```
if (a < b)
    u = x $\oplus$ y
else
    v = x $\oplus$ y
```

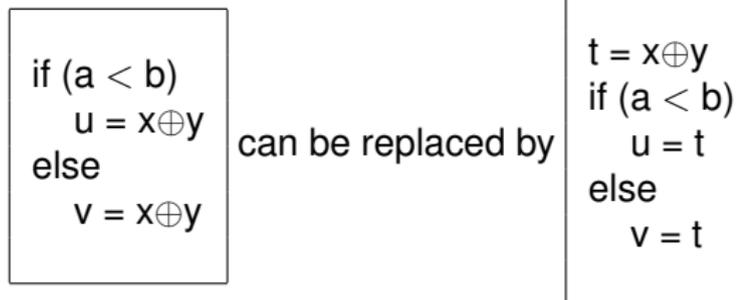
Code Movement

- ▶ Code size reduction
 - ▶ Suppose the operator \oplus results in the generation of a large number of machine instructions. Then,



Code Movement

- ▶ Code size reduction
 - ▶ Suppose the operator \oplus results in the generation of a large number of machine instructions. Then,



- ▶ **When should we NOT apply it?**

Code Movement

- ▶ Execution frequency reduction

```
if (a < b)
    u = ...
else
    v = x * y
w = x * y
```

Code Movement

- ▶ Execution frequency reduction

```
if (a < b)
    u = ...
else
    v = x * y
    w = x * y
```

can be replaced by

```
if (a < b)
    u = ...
    t = x * y
else
    t = x * y
    v = t
    w = t
```

Code Movement

- ▶ Execution frequency reduction

```
if (a < b)
    u = ...
else
    v = x * y
    w = x * y
```

can be replaced by

```
if (a < b)
    u = ...
    t = x * y
else
    t = x * y
    v = t
    w = t
```

- ▶ **When should we NOT apply it?**

Loop Invariant Code Movement

- ▶ Move loop invariant code out of the loop

```
for (...) {  
    ...  
    u = a + b  
    ...  
}
```

Loop Invariant Code Movement

- ▶ Move loop invariant code out of the loop

```
for (...) {  
  ...  
  u = a + b  
  ...  
}
```

can be replaced by

```
t = a + b  
for (...) {  
  ...  
  u = t  
  ...  
}
```

Loop Invariant Code Movement

- ▶ Move loop invariant code out of the loop

```
for (...) {  
  ...  
  u = a + b  
  ...  
}
```

can be replaced by

```
t = a + b  
for (...) {  
  ...  
  u = t  
  ...  
}
```

- ▶ **When should we NOT apply it?**

Code Movement

Safety of code motion

Code Movement

Safety of code motion

Profitability of code motion

Other Optimizations

- ▶ Dead code elimination

Other Optimizations

- ▶ Dead code elimination
 - ▶ Remove unreachable and/or unused code.

Other Optimizations

- ▶ Dead code elimination
 - ▶ Remove unreachable and/or unused code.
 - ▶ **Can we always do it?**

Other Optimizations

- ▶ Dead code elimination
 - ▶ Remove unreachable and/or unused code.
 - ▶ Can we always do it?
 - ▶ Is there ever a need to introduce unused code?

Other Optimizations

- ▶ Dead code elimination
 - ▶ Remove unreachable and/or unused code.
 - ▶ Can we always do it?
 - ▶ Is there ever a need to introduce unused code?
- ▶ Strength Reduction

Other Optimizations

- ▶ Dead code elimination
 - ▶ Remove unreachable and/or unused code.
 - ▶ **Can we always do it?**
 - ▶ **Is there ever a need to introduce unused code?**
- ▶ Strength Reduction
 - ▶ Use of *low strength* operators in place of *high* strength ones.

Other Optimizations

- ▶ Dead code elimination
 - ▶ Remove unreachable and/or unused code.
 - ▶ Can we always do it?
 - ▶ Is there ever a need to introduce unused code?
- ▶ Strength Reduction
 - ▶ Use of *low strength* operators in place of *high* strength ones.
 - ▶ $i * i$ instead of $i * * 2$, $\text{pow}(i, 2)$

Other Optimizations

- ▶ Dead code elimination
 - ▶ Remove unreachable and/or unused code.
 - ▶ Can we always do it?
 - ▶ Is there ever a need to introduce unused code?
- ▶ Strength Reduction
 - ▶ Use of *low strength* operators in place of *high* strength ones.
 - ▶ $i * i$ instead of $i * * 2$, $\text{pow}(i, 2)$
 - ▶ $i << 1$ instead of $i * 2$

Other Optimizations

- ▶ Dead code elimination
 - ▶ Remove unreachable and/or unused code.
 - ▶ Can we always do it?
 - ▶ Is there ever a need to introduce unused code?
- ▶ Strength Reduction
 - ▶ Use of *low strength* operators in place of *high* strength ones.
 - ▶ $i * i$ instead of $i * * 2$, $\text{pow}(i, 2)$
 - ▶ $i < 1$ instead of $i * 2$
 - ▶ Typically performed for integers only – Why?

Agenda

- ▶ Static analysis and compile-time optimizations

Agenda

- ▶ Static analysis and compile-time optimizations
- ▶ For the next few lectures

Agenda

- ▶ Static analysis and compile-time optimizations
- ▶ For the next few lectures
- ▶ *Intraprocedural* Data Flow Analysis

Agenda

- ▶ Static analysis and compile-time optimizations
- ▶ For the next few lectures
- ▶ *Intraprocedural* Data Flow Analysis
 - ▶ Classical Examples

Agenda

- ▶ Static analysis and compile-time optimizations
- ▶ For the next few lectures
- ▶ *Intraprocedural* Data Flow Analysis
 - ▶ Classical Examples
 - ▶ Components

Assumptions

- ▶ Intraprocedural: Restricted to a single function

Assumptions

- ▶ Intraprocedural: Restricted to a single function
- ▶ Input in 3-address format

Assumptions

- ▶ Intraprocedural: Restricted to a single function
- ▶ Input in 3-address format
- ▶ Unless otherwise specified

3-address Code Format

- ▶ Assignments

3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

3-address Code Format

- ▶ Assignments

 - $x = y \text{ op } z$

 - $x = \text{op } y$

 - $x = y$

- ▶ Jump/control transfer

3-address Code Format

- ▶ Assignments

 - $x = y \text{ op } z$

 - $x = \text{op } y$

 - $x = y$

- ▶ Jump/control transfer

 - goto L

3-address Code Format

- ▶ Assignments

 - $x = y \text{ op } z$

 - $x = \text{op } y$

 - $x = y$

- ▶ Jump/control transfer

 - goto L

 - if x relop y goto L

3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

goto L

if x relop y goto L

- ▶ Statements can have label(s)

3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

goto L

if x relop y goto L

- ▶ Statements can have label(s)

L: ...

3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

goto L

if x relop y goto L

- ▶ Statements can have label(s)

L: ...

- ▶ Arrays, Pointers and Functions to be added later when needed