Compiler Design

Lecture-22

Introduction to Optimization

Topics Covered

- Introduction to Optimization
- Classifications of Optimization techniques
- Compile-Time Evaluation

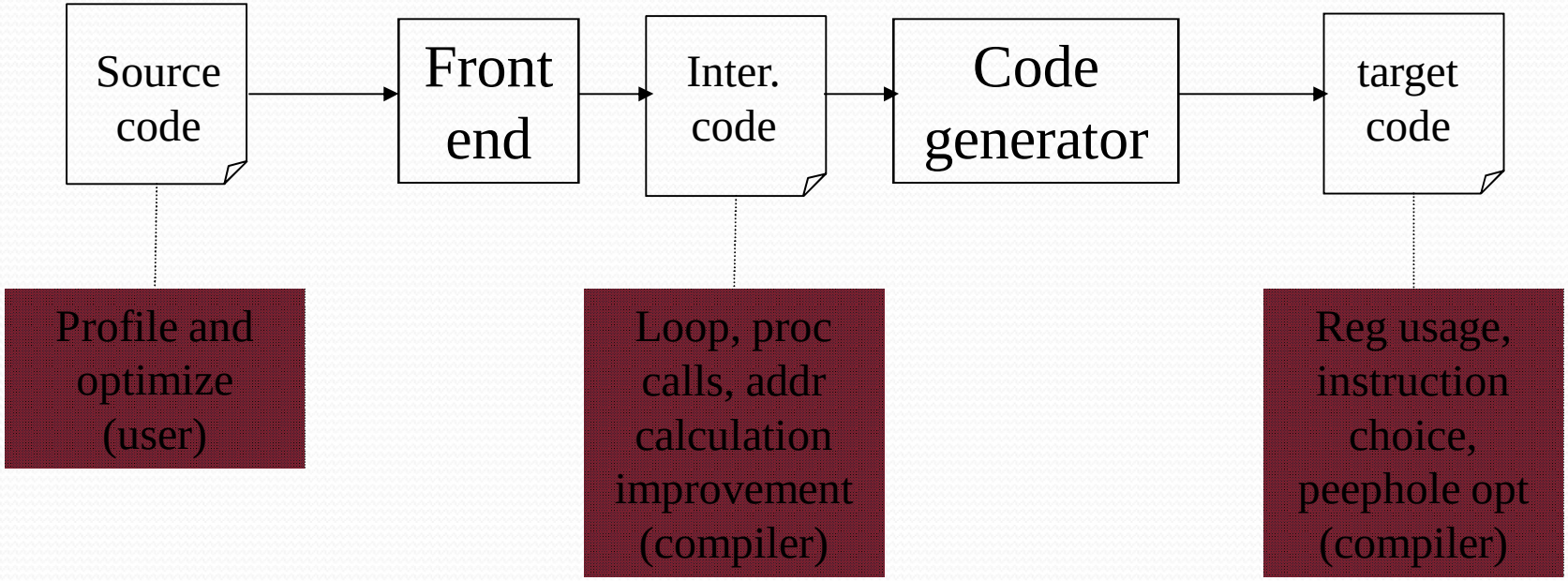
Introduction

- Criterion of code optimization
  - Must preserve the semantic equivalence of the programs
  - The algorithm should not be modified
  - Transformation, on average should speed up the execution of the program
  - Worth the effort: Intellectual and compilation effort spend on insignificant improvement.

    Transformations are simple enough to have a good effect

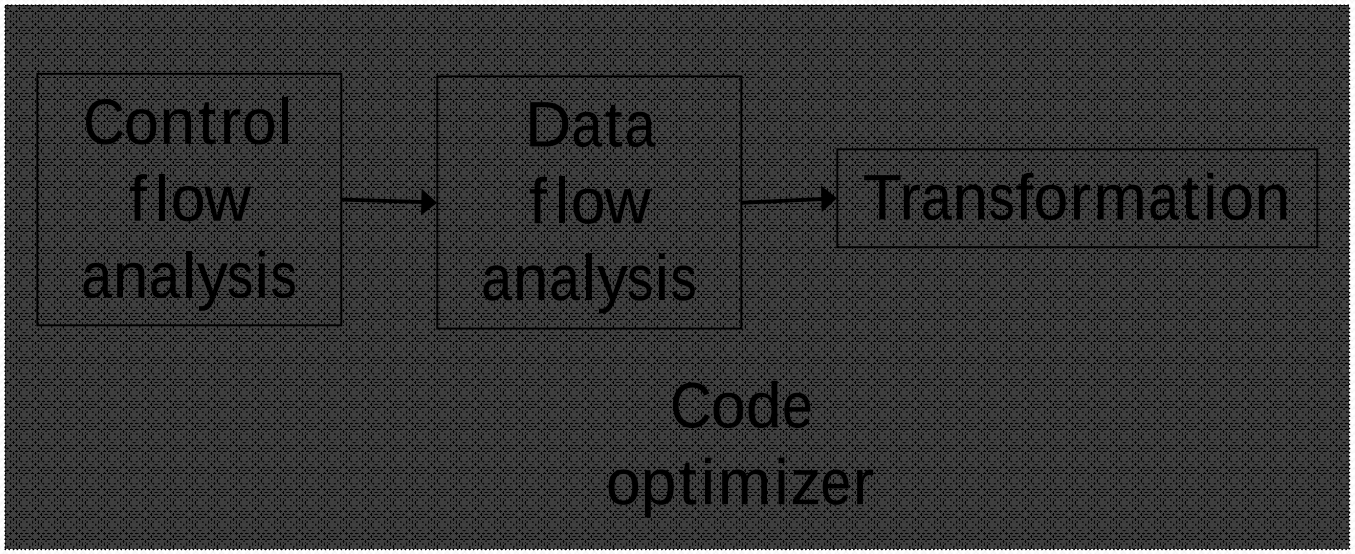
Introduction

- Optimization can be done in almost all phases of compilation.



Introduction

- Organization of an optimizing compiler



Classifications of Optimization techniques

- Peephole optimization
- Local optimizations
- Global Optimizations
  - Inter-procedural
  - Intra-procedural
- Loop optimization

Factors influencing Optimization

- The target machine: machine dependent factors can be parameterized to compiler for fine tuning
- Architecture of Target CPU:
  - Number of CPU registers
  - RISC vs CISC
  - Pipeline Architecture
  - Number of functional units
- Machine Architecture
  - Cache Size and type
  - Cache/Memory transfer rate

Themes behind Optimization Techniques

- Avoid redundancy: something already computed need not be computed again
- Smaller code: less work for CPU, cache, and memory!
- Less jumps: jumps interfere with code pre-fetch
- Code locality: codes executed close together in time is generated close together in memory – increase locality of reference
- Extract more information about code: More info – better code generation

Redundancy elimination

- Redundancy elimination = determining that two computations are equivalent and eliminating one.
- There are several types of redundancy elimination:
  - Value numbering
    - Associates symbolic values to computations and identifies expressions that have the same value
  - Common subexpression elimination
    - Identifies expressions that have operands with the same name
  - Constant/Copy propagation
    - Identifies variables that have constant/copy values and uses the constants/copies in place of the variables.
  - Partial redundancy elimination
    - Inserts computations in paths to convert partial redundancy to full redundancy.

Optimizing Transformations

- Compile time evaluation
- Common sub-expression elimination
- Code motion
- Strength Reduction
- Dead code elimination
- Copy propagation
- Loop optimization
  - Induction variables and strength reduction

Compile-Time Evaluation

- Expressions whose values can be pre-computed at the compilation time
- Two ways:
  - Constant folding
  - Constant propagation

Compile-Time Evaluation

- **Constant folding**: Evaluation of an expression with constant operands to replace the expression with single value
- Example:

```
area := (22.0/7.0) * r ** 2
```

```
area := 3.14286 * r ** 2
```

Compile-Time Evaluation

- **Constant Propagation**: Replace a variable with constant which has been assigned to it earlier.
- Example:

```
pi := 3.14286
area = pi * r ** 2
area = 3.14286 * r ** 2
```

Constant Propagation

- What does it mean?
  - Given an assignment x = c, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.
    - Similar to copy propagation
    - Extra feature: It can analyze constant-value conditionals to determine whether a branch should be executed or not.
- When is it performed?
  - Early in the optimization process.
- What is the result?
  - Smaller code
  - Fewer registers

Common Sub-expression Evaluation

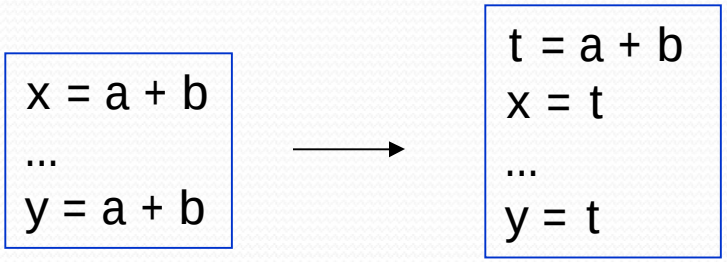
- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
  - The *definition* of the variables involved should not change

Example: ⟶

```
a := b * c              temp := b * c
…                       a := temp
…                       …
x := b * c + 5          x := temp + 5
```

Common Subexpression Elimination

- **Local** common subexpression elimination
  - Performed within basic blocks
  - Algorithm sketch:
    - Traverse BB from top to bottom
    - Maintain table of expressions evaluated so far
      - if any operand of the expression is redefined, remove it from the table
    - Modify applicable instructions as you go
      - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.



Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to L
```

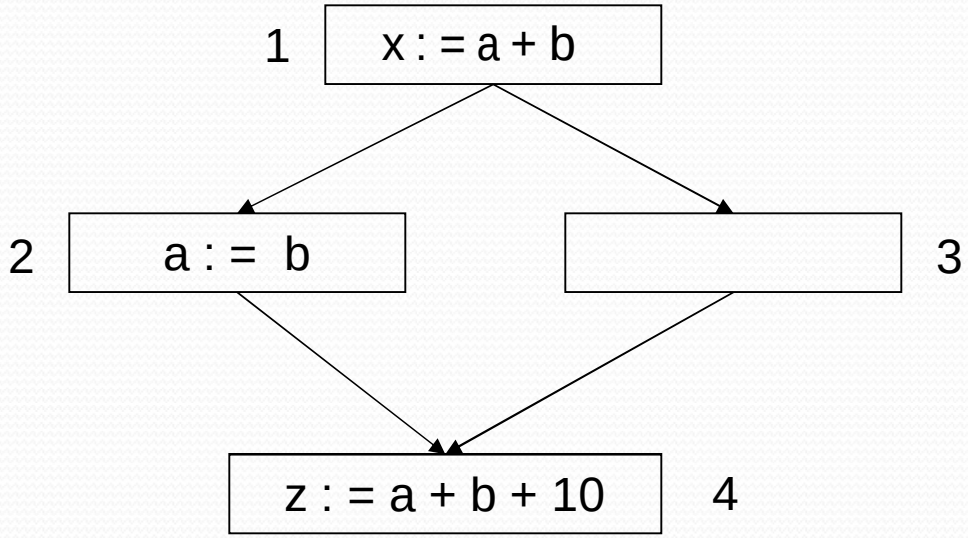
```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /* commutative */
a = j + a
k = t2
j = t3
a = -b
if t2 go to L
```

the table contains quintuples:
(pos, opd1, opr, opd2, tmp)

Common Subexpression Elimination

- **Global** common subexpression elimination
  - Performed on flow graph
  - Requires available expression information
    - In addition to finding what expressions are available at the endpoints of basic blocks, we need to know where each of those expressions was most recently evaluated (which block and which position within that block).

Common Sub-expression Evaluation



"a + b" is not a common sub-expression in 1 and 4

None of the variable involved should be modified in any path