Chapter 8. Optimization (Part 1)

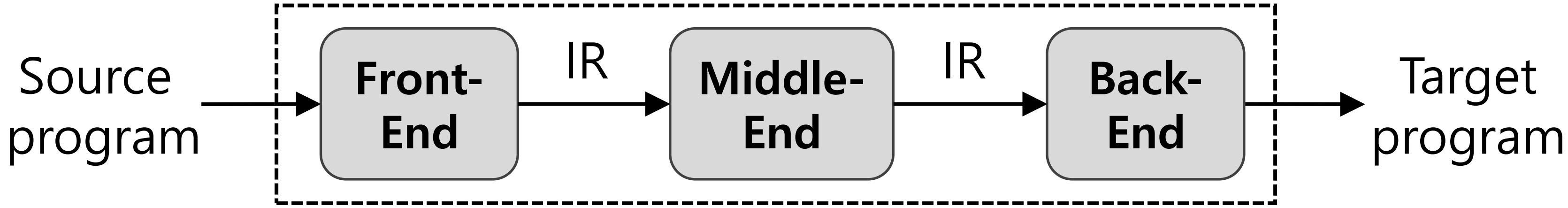
Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

Topics

- **General background**
  - Goal and principle of optimization
  - Scope of optimization
  - Basic block and CFG
- **Common types of optimization**
  - Constant folding, constant propagation, copy propagation, common subexpression elimination, dead code elimination, ...
- **Data-flow analyses to realize the optimization**
  - Reaching-definition analysis, constant propagation analysis, available expression analysis, liveness analysis, ...

Review: Compiler Structure

- **Middle-end takes in intermediate representation (IR) code and outputs more efficient version of IR code**
  - In modern compilers, this is often the most complex phase
  - Nowadays, IR optimization is an important topic in compiler
- **Machine-dependent optimization (which is applicable to specific architecture) may follow in the back-end phase**
  - Our course will not focus on this part



Optimization

- **Optimization converts code into more efficient form**
  - "More efficient" in terms of:
    - **Execution time** (the primary goal in most cases)
    - Code size
    - Memory usage
    - Energy consumption
  - Sometimes, these goals are corelated with each other
- **Optimization should not change the behavior of the input code (e.g., return value of a function)**
- **The name "optimization" is quite misleading: its result is far from *optimal code*; it is just *improved code***

Scope of Optimization (1)

- **Intra-procedural optimization**
  - Optimization that occurs **within a function boundary**
  - Analyze and reason about **one function** at a time
  - Ex) Function `f( ) { S }` can be rewritten into `f( ) { S'}`, regardless of what happens in other functions in the program
- **Inter-procedural optimization**
  - Optimization that occurs **across function boundaries**
  - Analyze and reason about **multiple functions** at a time
  - Ex) Considering the behavior of caller and callee functions of `f( )` as well during its optimization
  - Enables more optimization, but often **too expensive**
- **Most compilers do intra-procedural optimization only**

Scope of Optimization (2)

- **Recall that in our IR model (which resembles LLVM IR), registers and memory are separated**
- **In such cases, most compilers only focus on analyzing and optimizing the computation with registers**
  - Give up many optimizations that deal with variables in memory
  - Because reasoning about the memory state is hard
- **Q. Doesn't it limit the capacity of optimization too much?**
  - A. Good point, but we will also come back to this topic later

Memory

2000	100

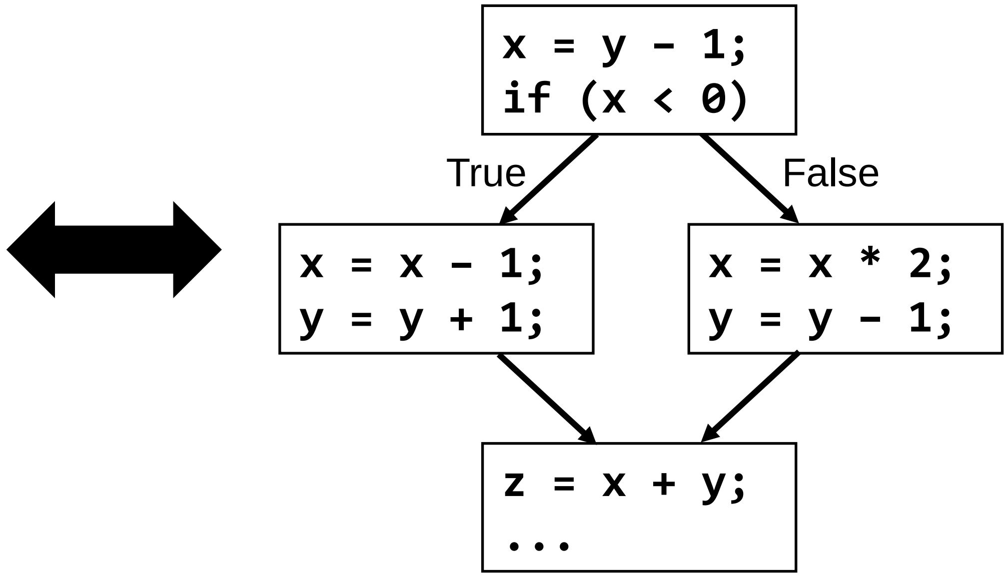
Registers

$t1	2000
$t2	100
...	...

Background: Control-Flow Graph

- **Control-flow graph (CFG): graph representation of code**
  - Applicable to both source-level code and IR-level code
  - Commonly used for data-flow analysis and optimization
  - Nodes are basic blocks (defined in the next page)
  - Edges represent control flows between the nodes

```
x = y – 1;
if (x < 0) {
  x = x - 1;
  y = y + 1;
} else {
  x = x * 2;
  y = y – 1;
}
z = x + y;
...
```



Basic Block

- **Basic block (BB) contains a sequence of statements (or instructions) that are always executed consecutively**
  - No jump-out from the middle of a basic block
  - No jump-in into the middle of a basic block
- **Ex) Basic block(s) of IR-level code**

```
$t1 = alloc 4
store 100, $t1
$t2 = load $t1
$t3 = $t2 + 5
```

```
$t2 = load $t1
$t3 = $t1 < 10
if $t3 goto L0
```

```
$t4 = $t3 + 3
store $t4, $t1
```

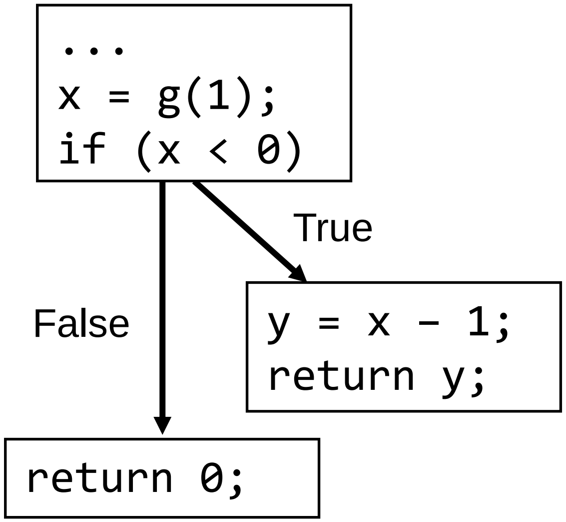
```
label L0
$t5 = load $t1
ret $t5
```

CFG and Function

- **Usually, we draw a CFG for each function**
  - The function entry becomes the initial node of the function CFG
  - Each return statement (instruction) becomes exit (terminal) node
- **What should we do with *call* statement (or instruction)?**
  - CFG often considers ***call*** as a non-jumping instruction, because it will eventually return back

Function `f()`

```
...
x = g(1);
if (x < 0) {
  y = x - 1;
  return y;
}
return 0;
```



Construction of CFG

- **Converting high-level control structures (`if-else`, `while`, `for`, ...) into CFG is straightforward**
  - If there are low-level jumps like C's **`goto`**, it can be more complex
- **Step 1. Find *leaders* from statement (instruction) list**
  - The first statement is a leader
  - A statement that is the target of a jump is a leader
  - A statement that immediately follows a jump or return is a leader
- **Step 2. Split statements into basic blocks**
  - Each leader forms a basic block: its basic block consists of the statements until the next leader or the end of statement list
- **Step 3. Draw edges between the basic blocks**

Common Types of Optimization

- **Popular optimizations adopted in modern compilers**
  - Constant folding
  - Constant propagation
  - Copy propagation
  - Common subexpression elimination
  - Dead code elimination
  - … and of course, many more
- **First, I will briefly explain what each optimization does**
  - Then, we will move on to how these optimizations can be actually implemented with data-flow analysis

Constant Folding

- **Let's start with the easy one**
  - Optimization that can be done by considering one instruction
- **Consider an instruction for addition: `$t1 = v1 + v2`**
  - Recall that in our IR, operand can be either constant or register
  - If both `v1` and `v2` are constants, the computation at the right-hand side can be simplified at compile time
  - Ex) `$t1 = 2 + 3 --> $t1 = 5`
- **The same principle can be also applied to other kinds of operations**
  - Ex) `$t2 = 5 > 3 --> $t2 = true`
  - Ex) `$t3 = true && false --> $t3 = false`

Constant Propagation

- If `$t1 = N` appears (where `N` is a constant), subsequent use of `$t1` may be replaced with the constant `N`
  - Ex)

```
$t1 = 5
$t2 = $t1
```

➡

```
$t1 = 5
$t2 = 5
```

- Constant propagation itself may not drastically improve the speed, but it gives chances to more optimizations
  - Ex)

```
$t1 = 5
$t2 = $t1 + 3
```

➡

```
$t1 = 5
$t2 = 5 + 3
```

Can fold constants

  - Ex)

```
$t1 = false
if $t1 goto L0
```

➡

```
$t1 = false
if false goto L0
```

Can be removed

Observation #1

- **Often, we must apply optimization passes repeatedly, until there is no more change in the code**
  - Or the compiler may choose to stop at certain threshold
- **Sometimes, we can reduce the number of repetition by making each optimization pass smarter**
  - For example, we may perform constant folding and constant propagation at once (we will see such optimization later)

```
$t1 = 4
$t2 = $t1 + 3
$t3 = $t2
```

Propagate

```
$t1 = 4
$t2 = 4 + 3
$t3 = $t2
```

Fold

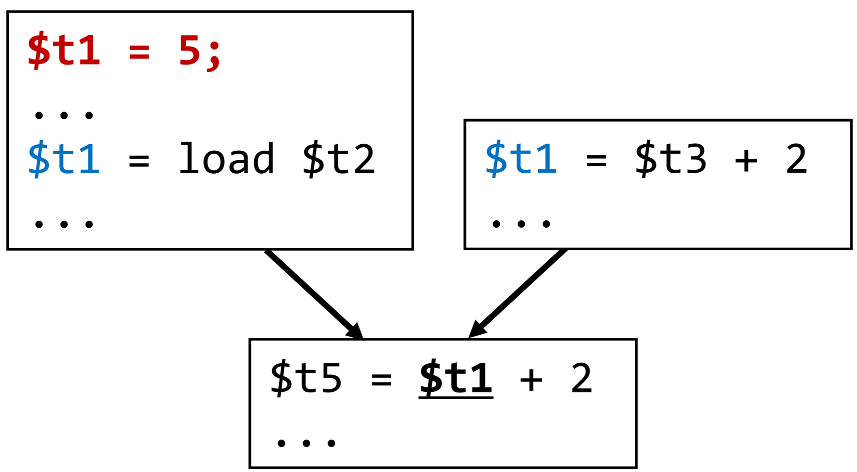
```
$t1 = 4
$t2 = 7
$t3 = $t2
```

Propagate

```
$t1 = 4
$t2 = 7
$t3 = 7
```

Observation #2

- **When performing constant propagation (and many other optimizations), we must carefully check if it is safe**
- **Ex) Consider the propagation with `$t1 = 5` below**
  - There are **other definitions of `$t1`** that can reach the use point of **`$t1`**, so we cannot replace it with constant **5**
- **This means we must analyze the data-flow of program**



Copy Propagation

- **If `$t1 = $t2` appears, subsequent use of `$t1` may be replaced with the register `$t2`**
  - As in constant propagation, data-flow analysis is required
- **Similarly, copy propagation is only meaningful for enabling other optimization**
  - Ex)

```
$t2 = $t1
$t3 = $t2 + 5
ret $t3
```

→

```
$t2 = $t1
$t3 = $t1 + 5
ret $t3
```

Can be removed as `$t2` is unused

- **Some textbooks consider constant propagation as a specific case of copy propagation**
  - Our lecture note will distinguish them when possible

Common Subexp. Elimination

- **Common subexpression elimination (CSE) can avoid recomputing certain expressions**
- **Assume two instructions that define registers (`$t2`, `$t3` in the example below)**
  - The two registers will have the same value if:
    (1) Their right-hand side expressions are same,
    (2) Registers used in this expression are not updated, and
    (3) The first register is not redefined
  - Then, we can replace the second computation with first register

```
$t2 = 4 * $t1
... (No update on $t1, $t2)
$t3 = 4 * $t1
```

➡

```
$t2 = 4 * $t1
...
$t3 = $t2
```

Dead Code* Elimination (DCE)

- **An instruction is said to be *dead* if it computes a value that is never used later**
  - Such instruction will not affect the execution result of a program, so it can be eliminated
- **Such dead-code may occur for various reasons**
  - Redundant logic in the original source code
  - As a result of other optimizations (e.g., constant/copy propagation)

```
$t2 = $t1
$t3 = $t2 + 5
ret $t3
```

Copy propagate

```
$t2 = $t1
$t3 = $t1 + 5
ret $t3
```

DCE

```
(removed)
$t3 = $t1 + 5
ret $t3
```

*In some contexts, "dead code" refers to *unreachable code*