Code Optimization

- Ideally, compiler should produce code as good (i.e., fast) as the best human programmer can
- Practically, this rarely happens – “optimize” is a misnomer
- Optimizing compiler is one that applies some transformations to produce better code (“meliorating” compiler?)
- Chapter 9 covered machine-dependent optimizations (register allocation, special instructions)
- Here we focus on optimizing intermediate code

10.1 Intro to Optimization

- Transformations must
  1. Preserve program semantics (meaning, input/output relation)
  2. Speed up programs by a measurable amount
  3. Be worth the effort (human effort + extra compile time)
- Biggest improvements are multi-stage effort
  1. **Source**: programmer profiles code and uses faster algorithms (Rice's Theorem – this can't be automated!)
  2. **Intermediate**: Compiler does transforms described here
  3. **Target**: Compiler does register allocation and peephole transforms

Code Optimization

- Identify most frequently executed statements
  - 90/10 (80/20) Rule: most time is spent in a small amount of code (loops)
  - Profiling at runtime can help reveal this
    - `java -Xprof MyProgram`
- Of course, compiler doesn't have run-time info!
- So compiler uses *data-flow analysis*: collect (static) information about how variables are used in a program
Optimization Example : Quicksort

```c
void quicksort(int[] a, int m, int n) {
    if (n <= m) return;
    int i = m-1, j = n, v = a[n];
    while (true) {
        do { i++; } while (a[i] < v);
        do { j--; } while (a[j] > v);
        if (i >= j) break;
        int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
    }
    int tmp = a[i]; a[i] = a[n]; a[n] = tmp;
    quicksort(a, m, j);
    quicksort(a, i+1, n);
}
```

Optimization Example : Quicksort

```c
void quicksort(int[] a, int m, int n) {
    if (n <= m) return;
    int i = m-1, j = n, v = a[n];
    while (true) {
        do { i++; } while (a[i] < v);
        do { j--; } while (a[j] > v);
        if (i >= j) break;
        int x = a[i]; a[i] = a[j]; a[j] = x;
    }
    int x = a[i]; a[i] = a[n]; a[n] = x;
    quicksort(a, m, j);
    quicksort(a, i+1, n);
}
```

• Three-address (intermediate) code provides opportunities for optimization that cannot be done in source language

• E.g. array dereference on a 32-bit (four-byte) machine:

  ```
  int x = a[i]; a[i] = a[j]; a[j] = x;
  ```

  ![Compiler](https://via.placeholder.com/150)

  ![Optimize](https://via.placeholder.com/150)

  ```
  t1 := 4 * i
  x := a[t1]
  t2 := 4 * j
  a[t1] := a[t2]
  a[t2] := x
  ```

• After this stage, only compiler has control over writing efficient target code (except for register declarator in C)

• If compiler can do transformations, programmer can concentrate on writing clear code – e.g., algorithms in literature typically contain redundancies for clarity

• Stages of optimization are:

  1. Control-flow analysis: determine locations of jumps
  2. Data-flow analysis: build flow graph
  3. Transformations: use flow graph to transform code
### 10.2 The Principle Sources of Optimization

- Transformations can be:
  - **Local**: look within basic block
  - **Global**: look across blocks
- Transformations should preserve function of program.
  - Function-preserving transformations include:
    - Common subexpression elimination
    - Copy propagation
    - Dead-code elimination
    - Constant-folding

---

### Common Subexpression Elimination

- Occurrence of expression $E$ is called common subexpression if
  - $E$ was previously computed, and
  - values of variables in $E$ have not changed since previous computation
- Use previous computed value:
Copy Propagation
- Statement of form \( x := \text{g} \) is called a copy statement
- Idea is to use \( \text{g} \) instead of \( x \) in subsequent statements
- Doesn't help by itself, but can combine with other transformations to help eliminate code:

Dead Code Elimination
- Variable that is no longer live (subsequently used) is called dead
- Copy propagation often turns copy statement into dead code:

Loop Optimizations
- Biggest speedups often come from moving code out of inner loop
- Works even if we have to put it in outer loop – why?
- Three techniques
  - Code motion
  - Induction-variable elimination
  - Reduction in strength
**Code Motion**

- Expression whose value doesn't change inside loop is called a **loop-invariant**
- Code motion moves loop-invariants outside loop

```plaintext
while (i <= limit-2)
    t = limit - 2;
while (i <= t)
```

**Induction Variables and Reduction in Strength**

- Variables that remain "in lock step" with each other inside a loop are called **induction variables**
- E.g., decreasing array byte-offset index by 4 as loop variable decreases by 1:

```plaintext
j := j - 1
```

- Addition is like multiplication, "reduced in strength" (less costly)
- Exploit induction variables and reduction in strength to make loop code more efficient

```
1 := n - 1
j := 4 * i
v := a[t4]
if t5 > v goto B
```