Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering
  - eliminating minor inefficiencies
- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraint
  - Must not cause any change in program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Machine-Independent Optimizations

- Optimizations you should do regardless of processor / compiler
  - Code Motion
    - Reduce frequency with which computation performed
      - If it will always produce same result
      - Especially moving code out of loop

Compiler-Generated Code Motion

- Most compilers do a good job with array code + simple loop structures
  - Code Generated by GCC

```c
for (i = 0; i < n; i++) { int ni = n*i; int *p = a+ni; for (j = 0; j < n; j++) *p++ = b[j]; }
```

```c
for (i = 0; i < n; i++) { int ni = n*i; int *p = a+ni; for (j = 0; j < n; j++) *p++ = b[j]; }
```

```c
imull %ebx,%eax # i*n movl 8(%ebp),%edi # a leal (%edi,%ecx,4),%edx # p = a+i*n (scaled by 4)
# Inner Loop .L40:
movl 12(%ebp),%edi # b movl (%edi,%ecx,4),%eax # b+j (scaled by 4) movl %eax,%edx # p = b[j] addl %edx, %edx # p++ (scaled by 4) inc %ecx # j++ jne .L40                 # loop if j<n
```
**Reduction in Strength**

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

```c
16\times \rightarrow x << 4
```

- Utility machine dependent
- Depends on cost of multiply or divide instruction
- On Pentium II or III, integer multiply only requires 4 CPU cycles

- Recognize sequence of products

```c
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
a[n+i + j] = b[j];

int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[n+i + j] = b[j];
    ni += n;
}
```

**Make Use of Registers**

- Reading and writing registers much faster than reading/writing memory

- **Limitation**
  - Compiler not always able to determine whether variable can be held in register
  - Possibility of **Aliasing**
  - See example later

**Machine-Independent Opts. (Cont.)**

- Share Common Subexpressions
  - Reading portions of expressions
  - Compilers often not very sophisticated in exploiting arithmetic properties

```c
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;

int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

- 3 multiplications: i*n, (i-1)*n, (i+1)*n 1 multiplication: i*n

**Time Scales**

- **Absolute Time**
  - Typically use nanoseconds
    - $10^{-9}$ seconds
  - Time scale of computer instructions
- **Clock Cycles**
  - Most computers controlled by high frequency clock signal
  - Typical Range
    - 100 MHz
      - $10^9$ cycles per second
      - Clock period = 10ns
    - 2 GHz
      - $2 \times 10^9$ cycles per second
      - Clock period = 0.5ns
Code Motion Example

- Procedure to Convert String to Lower Case

```c
void lower(char *s) {
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```c
void lower(char *s) {
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Optimization Blocker: Procedure Calls

- Why couldn’t the compiler move `strlen` out of the inner loop?
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`
  - Why doesn’t compiler look at code for `strlen`?
    - Linker may overload with different version
      - Unless declared static
    - Interprocedural optimization is not used extensively due to cost

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
- Example
  - `v: [3, 2, 17]`
  - `combine(v, get_vec_start(v)+2)` --> ?

```c
void combine(vec_ptr v, int *dest){
    int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++)
        *dest += get_vec_element(v, i, &val);
}
```
Machine-Independent Opt. Summary

- Code Motion
  - **Compilers are good at this for simple loop/array structures**
  - **Don't do well in presence of procedure calls and memory aliasing**
- Reduction in Strength
  - Shift, add instead of multiply or divide
    - **Compilers are (generally) good at this**
    - **Exact trade-offs machine-dependent**
  - Keep data in registers rather than memory
    - **Compilers are not good at this, since concerned with aliasing**
- Share Common Subexpressions
  - **Compilers have limited algebraic reasoning capabilities**

Important Tools

- Measurement
  - Accurately compute time taken by code
    - **Most modern machines have built in cycle counters**
    - Using them to get reliable measurements is tricky
  - Profile procedure calling frequencies
    - Unix tool gprof
- Observation
  - Generating assembly code
    - Lets you see what optimizations compiler can make
    - Understand capabilities/limitations of particular compiler