Topics

- Machine-Independent Optimizations
  - Basic optimizations
  - Optimization blockers

Harsh Reality

There's more to performance than asymptotic complexity

Constant factors matter too!
- Easily see 10:1 performance range depending on how code is written
- Must optimize at multiple levels:
  - algorithm, data representations, procedures, and loops

Must understand system to optimize performance
- How programs are compiled and executed
- How to measure program performance and identify bottlenecks
- How to improve performance without destroying code modularity and generality

Optimizing Compilers

Provide efficient mapping of program to machine
- register allocation
- code selection and ordering (scheduling)
- dead code elimination
- 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 that you or the compiler 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

void set_row(double *a, double *b, long l, long n)
{
  long j;
  for (j = 0; j < n; j++)
    a[n*j+i] = b[j];
}

Compiler-Generated Code Motion

void set_row(double *a, double *b, long l, long n)
{
  long j;
  for (j = 0; j < n; j++)
    a[n*j+i] = b[j];
}

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - $16 \times x \rightarrow x \ll 4$
    - Utility machine dependent
    - Depends on cost of multiply or divide instruction
    - On Pentium IV, integer multiply requires 10 CPU cycles
- Recognize sequence of products

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

Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

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

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

Optimization Blocker #1: Procedure Calls

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');
}
```

- Extracted from 213 lab submissions, Fall, 1998

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

![Graph showing performance of lower case conversion]

Convert Loop To Goto Form

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

- `strlen` executed every iteration

Calling `strlen`

```c
/* My version of strlen */
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- Only way to determine length of string is to scan its entire length, looking for null character.

Overall performance, string of length N

- N calls to `strlen`
- Require times N, N-1, N-2, ..., 1
- Overall O(N^2) performance
Improving Performance

```
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');
}
```

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

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of `lower2`

```
void lower2(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 compiler move `strlen` out of 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`

**Warning:**

- Compiler treats procedure call as a black box
- Weak optimizations near them

**Remedies:**

- Use of `inline` functions
- Do your own code motion

```
int lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0')
    {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

Memory Matters

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

```
/* Sum rows is of n X n matrix a
 and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++)
    {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L53:
addsd (%rcx), %xmm0    # FP add
addq $8, %rcx
deqq %rax
movsd %xmm0, (%rsi,%rax,8,8)  # FP store
jne .L53
```
**Memory Aliasing**

- Code updates \( b[i] \) on every iteration
- Must consider possibility that these updates will affect program behavior

```c
/* Sum rows is of n X n matrix a and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (;j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

- Value of \( B \):
  - \( i = 0 \): \([3, 8, 16]\)
  - \( i = 1 \): \([3, 22, 16]\)
  - \( i = 2 \): \([3, 22, 224]\)

**Removing Aliasing**

- No need to store intermediate results

```c
/* Sum rows is of n X n matrix a and store in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (;j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

**Unaliased Version**

- Aliasing still creates interference

```c
/* Sum rows is of n X n matrix a and store in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (;j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

- Value of \( B \):
  - \( init \): \([4, 8, 16]\)
  - \( i = 0 \): \([4, 8, 16]\)
  - \( i = 1 \): \([3, 22, 16]\)
  - \( i = 2 \): \([3, 27, 224]\)

**Optimization Blocker: Memory Aliasing**

- Aliasing
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

```c
double A[9] =
{ 0, 1, 2,
  4, 8, 16,
  32, 64, 128};
sum_rows1(A, B, 3);
```
Machine-Independent Opt. Summary

**Code Motion**
- Compilers are good at this for simple loop/array structures
- Don’t do well in the 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 (local variables) rather than memory
  - Compilers are not good at this, since concerned with aliasing
  - Compilers do know how to allocate registers (no need for register declaration)

**Share Common Subexpressions**
- Compilers have limited algebraic reasoning capabilities