There’s more to performance than asymptotic complexity

Constant factors matter too!

- Factor of 10 improvement is possible 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

Don’t (usually) improve asymptotic efficiency
- The programmer must select a good algorithm
- big-O savings are more important than constant factors
  - but constant factors also matter

Compilers have difficulty overcoming “optimization blockers”
- potential memory aliasing
- potential procedure side-effects
Limitations of Optimizing Compilers

Fundamental Constraint:
- Must not cause any change in program behavior under any possible condition
- Even pathological conditions.

Most analysis is performed only within procedures
- Whole-program analysis is too expensive

Most analysis is based only on static information
- The compiler doesn’t anticipate run-time inputs
- The programmer knows more about constraints on the data than the compiler.

When in doubt, the compiler must be conservative
Compiler-Generated Code Motion

- Most compilers do a good job with array code and simple loop structures

Code Generated by GCC

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

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

```
imull %ebx,%eax  # i*n
movl 8(%ebp),%edi  # a
leal (%edi,%eax,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 $4,%edx  # p++ (scaled by 4)
    incl %ecx  # j++
    jl .L40  # loop if j<n
```
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[ 16 \times x \rightarrow x \ll 4 \]
  - The utility of this is machine dependent
  - On Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products
Reduction in Strength

Write C code to show what the compiler generated.

```c
int foo(int a[], int b[], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            a[n * i + j] = b[j];
    }
}
```
### Make Use of Registers

- Reading and writing registers is 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

**Another limitation in the case of Intel processors**

- Almost no registers
- You have to make use of cache
Machine-Independent Opts. (Cont.)

Share Common Subexpressions

- Reuse 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;
```

How can we change this code so it doesn’t do 3 multiplications?

```assembly
leal -1(%edx),%ecx       # i-1
imull %ebx,%ecx          # (i-1)*n
leal 1(%edx),%eax        # i+1
imull %ebx,%eax          # (i+1)*n
imull %ebx,%edx          # i*n
```
Example

Data type: vector, illustrated above

Procedures

```c
vec_ptr new_vec(int len)
    ● Create vector of specified length
int get_vec_element(vec_ptr v, int index, int *dest)
    ● Retrieve vector element, store at *dest
    ● Return 0 if out of bounds, 1 if successful
int *get_vec_start(vec_ptr v)
    ● Return pointer to start of vector data
```

Structured programming

● Hide the implementation of the array
● Always do bounds checking
Optimization Example

void combine1(vec_ptr v, int *dest) {
    int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}

Procedure

- Compute sum of all elements of integer vector
- Store result at destination location
- Vector data structure and operations defined via abstract data type

Pentium II/III Performance: Clock Cycles / Element

- 42.06 (Compiled -g) 31.25 (Compiled -O2)
Understanding the “for” Loop

void combine1-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    if (i >= vec_length(v)) 1 iteration
        goto done;
    loop:
    get_vec_element(v, i, &val);
    *dest += val;
    i++;
    if (i < vec_length(v))
        goto loop
    done:

Inefficiency

- Procedure vec_length is called every iteration
- Even though result is always the same
Exercise

Write a function \textit{combine2} that does the same thing as \textit{combine1}, without calling \texttt{vec\_length} on each iteration.
Move `vec_length` Call Out of Loop

Optimization

- **Move call to `vec_length` out of inner loop**
  - Value does not change from one iteration to next
  - Code motion
- **CPE:** 20.66 (Compiled -O2)
  - `vec_length` requires only constant time, but significant overhead
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

![Graph](image)
Exercise

- Why is the time proportional to the square of the string length?
- How can you optimize the function to make it linear?
- Write the optimized code.
- Why can’t the compiler do that optimization?
Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance

![Graph showing CPU seconds vs string length for lower1 and lower2 performance.

<table>
<thead>
<tr>
<th>String Length</th>
<th>CPU Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.000001</td>
</tr>
<tr>
<td>512</td>
<td>0.00001</td>
</tr>
<tr>
<td>1024</td>
<td>0.0001</td>
</tr>
<tr>
<td>2048</td>
<td>0.001</td>
</tr>
<tr>
<td>4096</td>
<td>0.01</td>
</tr>
<tr>
<td>8192</td>
<td>0.1</td>
</tr>
<tr>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>32768</td>
<td>10</td>
</tr>
<tr>
<td>65536</td>
<td>100</td>
</tr>
<tr>
<td>131072</td>
<td>1000</td>
</tr>
<tr>
<td>262144</td>
<td>10000</td>
</tr>
</tbody>
</table>

(lower1, lower2)
Optimization Blocker: Procedure Calls

Compiler treats procedure call as a black box
- Weak optimizations in and around them

Why?
Reduction in Strength

void combine3(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        *dest += data[i];
    }
}

Optimization

■ Avoid procedure call to retrieve each vector element
  ● Get pointer to start of array before loop
  ● Not as clean in terms of data abstraction
    » Makes assumption about what a vector looks like internally

■ CPE: 6.00 (Compiled -O2)
  ● Procedure calls are expensive!
  ● Bounds checking is expensive
Eliminate Unneeded Memory Refs

```
void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}
```

**Optimization**

- How many memory references does this avoid per element?
- How does it avoid them?
- CPE: 2.00 (Compiled -O2)
  - Memory references are expensive!
Detecting Unneeded Memory Refs.

**Combine3**

```
.L18:
    movl (%ecx,%edx,4),%eax
    addl %eax,(%edi)
    incl %edx
    cmpl %esi,%edx
    jl .L18
```

**Combine4**

```
.L24:
    addl (%eax,%edx,4),%ecx
    incl %edx
    cmpl %esi,%edx
    jl .L24
```

**Performance**

- **Combine3**
  - 5 instructions in 6 (or more) clock cycles
  - `addl` must read memory and write to cache
    » With 200 mhz CPU, a cache miss can entail up to 30 cycles

- **Combine4**
  - 4 instructions in 2 clock cycles
Optimization Blocker: Memory Aliasing

Aliasing
- Two different memory references specify single location

Example
- \( v: [3, 2, 17] \)
- \( \text{combine3}(v, \text{get_vec_start}(v)+2) \)
- What’s the problem?

Observations
- Easy for this to happen in C, with address arithmetic
- Get in habit of introducing local variables
  - Accumulating within loops
  - Your way of telling compiler it can optimize to its heart’s content
Machine-Independent Opt. Summary

Code Motion
- *Compilers are good at this for simple loop/array structures*
  - Local variables, no possible side effects
- *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 are machine-dependent*

Keep data in registers rather than memory
- *compilers are not good at this, 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
Code Profiling Example

Task
- Count word frequencies in text document
- Produce sorted list of words from most frequent to least

Steps
- Convert strings to lowercase
- Apply hash function
- Read words and insert into hash table
  - Mostly list operations
  - Maintain counter for each unique word
- Sort results

Data Set
- Collected works of Shakespeare
- 946,596 total words, 26,596 unique
- Initial implementation: 9.2 seconds

<table>
<thead>
<tr>
<th>Shakespeare’s most frequent words</th>
</tr>
</thead>
<tbody>
<tr>
<td>29,801  the</td>
</tr>
<tr>
<td>27,529  and</td>
</tr>
<tr>
<td>21,029  I</td>
</tr>
<tr>
<td>20,957  to</td>
</tr>
<tr>
<td>18,514  of</td>
</tr>
<tr>
<td>15,370  a</td>
</tr>
<tr>
<td>14010   you</td>
</tr>
<tr>
<td>12,936  my</td>
</tr>
<tr>
<td>11,722  in</td>
</tr>
<tr>
<td>11,519  that</td>
</tr>
</tbody>
</table>
Code Profiling

Augment Executable Program with Timing Functions

- Computes (approximate) amount of time spent in each function
- Time computation method
  - Periodically (~ every 10ms) interrupt program
  - Determine what function is currently executing
  - Increment its timer by interval (e.g., 10ms)
- Also maintains counter for each function indicating number of times called

Using

```
gcc -O2 -pg prog.c -o prog
./prog
  • Executes in normal fashion, but also generates file gmon.out

gprof prog
  • Generates profile information based on gmon.out
```
Profiling Results

Call Statistics

- Number of calls and total time for each function

sort_words, called just once, uses 87% of CPU time

Where do you think we should focus our optimization efforts?
What Profiling is Good For

Amdahl’s Law

- The performance enhancement possible with a given improvement limited by the amount that the improved feature is used.

Suppose a module requires a fraction $\alpha$ of the total time, and we improve its performance by a factor of $k$.

- $T_{new} = (1 - \alpha)T_{old} + (\alpha T_{old})/k$
- $= T_{old}[(1 - \alpha) + \alpha/k]$
- Speedup $= [(1 - \alpha) + \alpha/k]^{-1}$

- As $\alpha \rightarrow 0$, Speedup $\rightarrow 1$, regardless of $k$
- As $\alpha \rightarrow 1$, Speedup $\rightarrow k$
Profiling Observations

Benefits

- Helps identify performance bottlenecks
- Especially useful with a complex system with many components

Limitations

- Only shows performance for data tested
  - Quadratic inefficiency could remain lurking in code
- Timing mechanism fairly crude
  - Only works for programs that run for > 3 seconds
Is it really a good idea to move code around to save some CPU cycles?

How often is it worthwhile to sacrifice maintainability for a linear performance improvement?

- Almost never.
- If you’re writing specialized library code, for example.

Why is it good to understand these concepts?

- Using local variables and avoiding possible side effects is a good habit in general
  - If the compiler can do good optimizations, it’s a sign that the code is well-structured
  - Optimizable code is not necessarily un-maintainable
- Every once in a while you run into a bottleneck or a performance anomaly that you need to understand.