Great Reality #4
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
- 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

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

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

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

```assembly
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++
j1 .L40 # loop if j<n
```

### Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - $16*x \rightarrow x \ll 4$
    - 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];
```

```c
int ni = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[ni + 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

- 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;
```

```c
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

```
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
```

---

Vector ADT

```
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
```

- Similar to array implementations in Pascal, ML, Java
- E.g., 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;
    }
}
```

---

Time Scales

**Absolute Time**

- Typically use nanoseconds
- \(10^{-9}\) seconds

**Clock Cycles**

- Most computers controlled by high frequency clock signal
- Typical Range
  - \(100\) MHz
    - \(10^8\) cycles per second
    - Clock period = 10ns
  - \(2\) GHz
    - \(2 \times 10^9\) cycles per second
    - Clock period = 0.5ns
- Fish machines: 550 MHz (1.8 ns clock period)
Cycles Per Element

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- \[ T = \text{CPE} \cdot n + \text{Overhead} \]

![Graph showing cycles per element](image)

Optimization Example

```c
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 Loop

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

Move vec_length Call Out of Loop

```c
void combine2(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

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

Inefficiency

- Procedure vec_length called every iteration
- Even though result always the same
Code Motion Example #2

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

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
- strlen linear in length of string
- Must scan string until finds '\0'
- Overall performance is quadratic

Improving Performance

```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');
}
```

- 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

![Graph showing CPU time vs. string length]

Optimization Blocker: Procedure Calls

Why couldn’t the compiler move `vec_len` or `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 `vec_len` or `strlen`?
- Linker may overload with different version
  - Unless declared static
- Interprocedural optimization is not used extensively due to cost

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

Reduction in Strength

```c
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
  - Within loop just do pointer reference
  - Not as clean in terms of data abstraction
- CPE: 6.00 (Compiled -O2)
  - Procedure calls are expensive!
  - Bounds checking is expensive

Eliminate Unneeded Memory Refs

```c
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
- Don’t need to store in destination until end
- Local variable `sum` held in register
- Avoids 1 memory read, 1 memory write per cycle
- CPE: 2.00 (Compiled -O2)
  - Memory references are expensive!
Detecting Unneeded Memory Refs.

**Performance**
- **Combine3**
  - 5 instructions in 6 clock cycles
  - `addl %eax, (%edi)` must read and write memory
- **Combine4**
  - 4 instructions in 2 clock cycles

```
.L18:
  movl (%ecx, %edx, 4), %eax
  addl %eax, (%edi)
  incl %edx
  cmpl %esi, %edx
  j1 .L18

.L24:
  addl (%eax, %edx, 4), %ecx
  incl %edx
  cmpl %esi, %edx
  j1 .L24
```

Optimization Blocker: Memory Aliasing

**Aliasing**
- Two different memory references specify single location

**Example**
- `v: [3, 2, 17]`
- `combine3(v, get_vec_start(v)+2) --> ?`
- `combine4(v, get_vec_start(v)+2) --> ?`

**Observations**
- 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

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*
**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

---

**Shakespeare’s most frequent words**

<table>
<thead>
<tr>
<th>Word</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>29,801</td>
</tr>
<tr>
<td>and</td>
<td>27,529</td>
</tr>
<tr>
<td>i</td>
<td>21,029</td>
</tr>
<tr>
<td>to</td>
<td>20,957</td>
</tr>
<tr>
<td>of</td>
<td>18,514</td>
</tr>
<tr>
<td>a</td>
<td>15,370</td>
</tr>
<tr>
<td>you</td>
<td>14,010</td>
</tr>
<tr>
<td>my</td>
<td>12,936</td>
</tr>
<tr>
<td>in</td>
<td>11,722</td>
</tr>
<tr>
<td>that</td>
<td>11,519</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 -o prog
  ./prog`
  - Executes in normal fashion, but also generates file `gmon.out`
- `gprof prog`
  - Generates profile information based on `gmon.out`

---

**Profiling Results**

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self (seconds)</th>
<th>self (seconds)</th>
<th>total (seconds)</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>86.60</td>
<td>8.21</td>
<td>8.21</td>
<td>8210.00</td>
<td>1</td>
<td>8210.00</td>
<td></td>
<td>sort_words</td>
</tr>
<tr>
<td>5.80</td>
<td>8.76</td>
<td>0.55</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td>lower1</td>
</tr>
<tr>
<td>4.75</td>
<td>9.21</td>
<td>0.45</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td>find_ele_rec</td>
</tr>
<tr>
<td>1.27</td>
<td>9.33</td>
<td>0.12</td>
<td>946596</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td>h_add</td>
</tr>
</tbody>
</table>

**Call Statistics**
- Number of calls and cumulative time for each function

**Performance Limiter**
- Using inefficient sorting algorithm
- Single call uses 87% of CPU time

---

**Code Optimizations**

**First step:** Use more efficient sorting function
- Library function `qsort`
Further Optimizations

- Iter first: Use iterative function to insert elements into linked list
  - Causes code to slow down
- Iter last: Iterative function, places new entry at end of list
  - Tend to place most common words at front of list
- Big table: Increase number of hash buckets
- Better hash: Use more sophisticated hash function
- Linear lower: Move `strlen` out of loop

Profiling Observations

Benefits
- Helps identify performance bottlenecks
- Especially useful when have complex system with many components

Limitations
- Only shows performance for data tested
- E.g., linear lower did not show big gain, since words are short
  - Quadratic inefficiency could remain lurking in code
- Timing mechanism fairly crude
  - Only works for programs that run for > 3 seconds