Performance: More than Asymptotic Complexity

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

Must understand the 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.
- It’s up the programmer to select best overall algorithm.
- Big-O savings are often more important than constant factors.
- But constant factors also matter.

Limitations of Optimizing Compilers

Optimizing compilers have difficulty overcoming “optimization blockers”:
- potential memory aliasing
- potential procedure side-effects.

Compilers operate under a fundamental constraint:
- They must not cause any change in program behavior under any possible condition.
- This often prevents making optimizations when they would only affect behavior under pathological conditions.
Limitations of Optimizing Compilers

- Behavior obvious to the programmer may be hidden by languages and coding styles.
  - e.g., data ranges may be more limited than the variable type suggests.
- Most analysis is performed only within procedures; whole-program analysis is too expensive in most cases.
- Most analysis is based only on static information.
- When in doubt, the compiler must be conservative.

Machine-Independent Optimizations

Some optimizations you should do regardless of the processor / compiler.

**Code Motion:**

- Reduce frequency with which computation is performed, if it will always produce the same result.
- Move code out of loops if possible.

Compiler-Generated Code Motion

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

**Code generated by gcc:**

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

Compiler generates the equivalent of:

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

Reduction in Strength

- Replace costly operations with simpler ones.
- Shift, add instead of multiply or divide: $16 \times x$ becomes $x \ll 4$.
- The utility of this is machine dependent; depends on the cost of multiply and divide instructions.
- On Pentium II or III, integer multiply only requires 4 CPU cycles.

**Optimize as follows:**

```
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 is much faster than reading / writing memory.

Limitations:
- Compiler is not always able to determine whether a variable can be held in a register.
- There’s the possibility of aliasing.

```
int adder(int *p, int *q) {
    *p = 2;
    *q = 3;
    return (*p + *q);
}
```

What value is returned? Couldn’t we just return 5 and save two memory references?

Not so fast! What if p and q point to the same location (i.e., contain the same address)?

Aliasing means that a location may have multiple names. Often, the compiler must assume that aliasing is possible.

Simple Program

```
int adder(int *p, int *q) {
    *p = 2;
    *q = 3;
    return (*p + *q);
}
```

What value is returned? Couldn’t we just return 5 and save two memory references?

Machine-Independent Optimizations (Continued)

Share Common Subexpressions:
- Reuse portions of expressions.
- Compilers often are not very sophisticated in exploiting arithmetic properties.

```
/* 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;
```

Uses 1 multiplication:

```
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;
```

Uses 3 multiplications:

```
leal -1(%edx),%ecx
imull %ebx,%ecx
leal 1(%edx),%eax
imull %ebx,%eax
imull %ebx,%edx
```
**Time Scales**

**Absolute time:** Typically uses nanoseconds ($10^{-9}$ seconds).

**Clock cycles:**
- Most computers are controlled by a high frequency clock signal.
- **Typical range:**
  - Low end: 100 MHz: $10^8$ cycles per second; clock period = 10 ns.
  - High end: 2 GHz: $2 \times 10^9$ cycles per second; clock period = 0.5 ns.

**Example of Performance Measurement**

**Loop unrolling:** Perform more in each iteration of the loop.
(Assume even number of elements.)

Original loop:
```c
void vsum1(int n)
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

Loop unrolled:
```c
void vsum2(int n)
{
    int i;
    for (i = 0; i < n; i+=2)
    {
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] + b[i+1];
    }
}
```

Why would this make any difference in performance?

**Cycles Per Element**

CPE is a convenient way to express performance of a program that operates on vectors or lists.

If the vector length = $n$, then

$$T = \text{CPE} \times n + \text{Overhead}$$

---

**Code Motion Example**

Procedure to convert a 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');
}
```

Time quadruples when string length doubles (quadratic performance: $O(n^2)$). Why would that be?
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:
}
```

So what is the issue?  
- `strlen` is executed every iteration.  
- `strlen` is linear in length of the string; must scan string until it finds `\0`. Why is that?  
- Overall performance is quadratic. What do you do?

Improving Performance

Can move the call to `strlen` outside of loop, since the result does not change from one iteration to another. This is a 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');
}
```

Now, the run time doubles when the string length doubles (linear performance: $O(n)$).

Can you see other obvious optimizations in this code?

Optimization Blocker: Procedure Calls

Why couldn’t the compiler move `strlen` out of the inner loop?  
- Procedures may have side effects. E.g., might alter global state each time called.  
- Function may not return the same value for given arguments; might depend on other parts of the global state.  
- Procedure `lower` could interact with `strlen`.

Why doesn’t the compiler just look at the code for `strlen`?  
- The linker might overload with a different version (unless it’s declared static.  
- Inter-procedural optimization is rare because of the cost.

Warning:  
- The compiler treats a procedure call as a black box.  
- It applies weak optimizations in and around procedures.
Create a vector abstract data type similar to array implementations in Pascal, ML, Java. E.g., always do bounds checking.

**Procedures:**

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

**Optimization**

- Avoid procedure call to retrieve each vector element.
- Get pointer to start of array before loop.
- Within the 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**

void combine3( 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++ )
        *dest += data[i];
    *dest = sum;
}
Detecting Unneeded Memory Refs

Combine2

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

Combine3

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

Performance:
- Combine2: 5 instructions in 6 clock cycles; addl must read and write memory.
- Combine3: 4 instructions in 2 clock cycles.

Aliasing: two different memory references specify a single location.

Example:
- let v: [3, 2, 17]
  - combine2( v, get_vec_start(v)+2 ) → ?
  - combine3( v, get_vec_start(v)+2 ) → ?

Observations:
- This can easily occur in C, since you’re allowed to do address arithmetic.
- You have direct access to storage structures.
- Get into the habit of introducing local variables and accumulating within loops.
- This is your way of telling the compiler not to check for potential aliasing.

Previous Best Combining Code

void combine3( 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;
}

Task:
- Compute sum of all elements in vector.
- Vector is represented by C-style abstract data type.
- Achieved cycles per element (CPE) of 2.00.

Previous Best Combining Code

void abstract_combine3( vec_ptr v, data_t *dest )
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t t = IDENT;
    for( i = 0; i < length; i++ )
        t = t OP data[i];
    *dest = t;
}

Data Types: Use different declarations for data_t (int, float, double, etc.)

Operations: Use different definitions of OP and IDENT (+/0, */1, etc.)
### Machine Independent Optimization Results

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>abstract -g</td>
<td>42.06</td>
<td>41.86</td>
</tr>
<tr>
<td>abstract -O2</td>
<td>31.25</td>
<td>33.25</td>
</tr>
<tr>
<td>move vec_length</td>
<td>20.66</td>
<td>21.25</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>9.00</td>
</tr>
<tr>
<td>accum in temp</td>
<td>2.00</td>
<td>4.00</td>
</tr>
</tbody>
</table>

Optimizations: reduce function calls and memory references within loop.

Performance anomaly:
- Computing FP product of all elements exceptionally slow.
- Very large speedup when accumulate in temporary.
  - Caused by quirk in IA32 floating point.
  - Memory uses 64-bit format; register uses 80-bit format.
  - Benchmark data caused overflow in 64 bits, but not in 80 bits.

```c
void combine3p( vec_ptr v, int *dest )
{
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int *dend = data + length;
    int sum = 0;
    while (data < dend) {
        sum += *data;
        data++;
    }
    *dest = sum;
}
```

### Pointer Code

#### Pointer vs. Array Code Inner Loops

**Array Code:**
```
.L24:
  addl (%eax,%edx,4), %ecx # Loop
  incl %edx # sum += data[i]
  cmpl %esi,%edx # i++
  jl .L24 # i : length
```

**Pointer Code:**
```
.L30:
  addl (%eax), %ecx # Loop
  addl $4,%eax # sum += *data[i]
  addl %edx,%eax # data++
  cmpl %edx,%eax # data : dend
  jl .L30 # if < goto Loop
```

**Performance:**
- Array code: 4 instructions in 2 clock cycles
- Pointer code: almost same 4 instructions in 3 clock cycles

### Machine-Independent Optimization Summary

#### Code Motion
- Compilers are good at this for simple loop/array structures
- They don’t do well in the presence of procedure calls and potential memory aliasing.

#### Reduction in Strength
- Shift, add instead of multiply, divide
  - Compilers (generally) good at this.
  - The exact trade off is machine-dependent.
- Keep data in registers rather than memory.
  - Compilers are not good at this, since they are concerned with potential 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 the compiler can make;
- allows you to understand the capabilities / limitations of a particular compiler.

---

**Code Profiling Example**

**Task**
- Count word frequencies in a 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 the results.

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

**Shakespeare’s most frequent words.**

- 29,801 the
- 27,529 and
- 21,029 I
- 20,957 to
- 18,514 of
- 15,370 a
- 14,010 you
- 12,936 my
- 11,722 in
- 11,519 that

---

**Code Profiling**

Augment executable program with timing functions.
- Computes the (approximate) amount of time spent in each function.
- Time Computation method:
  - Periodically (~ every 10ms) interrupt program.
  - Determine what function is currently executing.
  - Increment the timer by interval (e.g., 10ms).
- Also maintains counter for each function indicating the number of times it is called.

**Using:**
```
gcc -O2 -pg prog.c -o prog
./prog
```
This 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>% time</th>
<th>cumulative seconds</th>
<th>self seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>86.60</td>
<td>8.21</td>
<td>8.21</td>
<td>1</td>
<td>8210.00</td>
<td>8210.00</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>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>fine_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>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.

*The first obvious step in optimization is to use a more efficient sorting algorithm.* Replacing the initial slow sort with the library function `qsort` (QuickSort), brought the time down from 9 seconds to around 1 second!
Further Optimizations

Iter first: use iterative function to insert elements into the linked list; actually causes code to slow down.
Iter last: iterative function that places new entries at end of the list rather than front; tends to place common words near the front of the list.
Big table: increase the number of hash functions.
Better hash: use a more sophisticated hash function.
Linear lower: move strlen out of the loop.

By applying these optimizations successively and profiling the result, the overall runtime was reduced to around 0.5 seconds.

Profiling Observations

Benefits
- Helps identify performance bottlenecks.
- Especially useful for complex systems with many components.

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

Role of the Programmer

How should I write my programs, given that I have a good optimizing compiler?

Don’t: Smash code into oblivion.
  - Becomes hard to read, maintain, and assure correctness.

Do:
  - Select the best algorithm.
  - Write code that’s readable and maintainable.
    - Use procedures and recursion and eliminate built-in limits.
    - Even though these factors can slow down code.
  - Eliminate optimization blockers to allow the compiler to do its job.
  - Focus on inner loops.
    - Do detailed optimizations where code will be executed repeatedly.
    - You’ll get the most performance gain here.

Summary

- Optimization blocker: procedure calls
- Optimization blocker: memory aliasing
- Tools (profiling) for understanding performance