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, generality, readability

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraint
  - Must not cause any change in program behavior under any possible condition
  - Often prevents making optimizations that 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
  - (gcc does quite a bit of interprocedural analysis—but not across files)

Most analysis is based only on static information
- Compiler has difficulty anticipating run-time inputs

When in doubt, the compiler must be conservative
Generally Useful 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
    - Gcc often does this for you (so check assembly)

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

**Compiler-Generated Code Motion (-O1)**

```asm
set_row:
    testq %rcx, %rcx # Test njle .L1 # If 0, goto done
    imulq %rcx, %rdx # ni = n*i
    leaq (%rdi,%rdx,8), %rdx # rowp = A + ni*8
    movl $0, %eax # j = 0
.L3: # loop:
    movsd (%rsi,%rax,8), %xmm0    # t = b[j]
    movsd %xmm0, (%rdx,%rax,8)   # M[A+ni*8 + j*8] = t
    addq $1, %rax # j++
    cmpq %rcx, %rax # j:njne .L3 # if !=, goto loop
.L1: # done:
    rep ; ret
```

**Strength Reduction**

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  
  ```c
  long j;long ni = n*i;double *rowp = a+n*i;for (j = 0; j < n; j++)
      rowp[j] = b[j];
  ```

**Share Common Subexpressions**

- Reuse portions of expressions
- Gcc will do this with -O1 and up

```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;
```

```asm
leaq    1(%rsi), %rax  # i+1
leaq   -1(%rsi), %r8  # i-1
imulq  %rcx, %rsi     # i*n
imulq  %rcx, %rax     # (i+1)*n
imulq  %rcx, %r8      # (i-1)*n
addq   %rdx, %rsi     # i*n+j
addq   %rdx, %rax     # (i+1)*n+j
addq   %rdx, %r8      # (i-1)*n+j
```
Optimization Blocker #1: Procedure Calls

Procedure to Convert String to Lower Case

```c
#include <ctype.h>
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (isupper(s[i]))
            s[i] = tolower(s[i]);
}
```

- Extracted from many student programs

Lower-Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance

![Graph showing quadratic performance](image)

Convert Loop To Goto Form

```c
void lower(char *s)
{
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
    loop:
        if (isupper(s[i]))
            s[i] = tolower(s[i]);
        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;
}
```

Strlen performance

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

Overall performance, string of length N

- N calls to strlen, each takes O(N) time
- Overall O(N^2) performance
Improving Performance

void lower(char *s)
{
    size_t i;
    size_t 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

Optimization Blocker: Procedure Calls

Why couldn’t compiler move strlen out of inner loop?
- Procedure may have side effects
- Might alter 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 inline functions
  - GCC does this with -O1
  - But only within single file
- Do your own code motion

Memory Matters

- Code updates b[i] on every iteration
- Why couldn’t compiler optimize this away?
Memory Aliasing

Code updates $b[i]$ on every iteration
Must consider possibility that these updates will affect program behavior

```
/* Sum rows is of n X n matrix a
and store in vector b */
void sum_rows(double *, double *, 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];
    }
}
```

double A[9] =
{ 0, 1, 2,
 4, 8, 16,
32, 64, 128};
double* B = A + 3;
sum_rows1(A, B, 3);

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

```
# sum_rows2 inner loop.
.L10:
addsd (%rdi), %xmm0 # FP load + add
addq $8, %rdi
icmpq %rax, %rdi
je .L10
```

```
/* Sum rows is of n X n matrix a
and store in vector b */
void sum_rows2(double *, double *, 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;
    }
}
```

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

Exploiting Instruction-Level Parallelism

Need general understanding of modern processor design
- Hardware can execute multiple instructions in parallel
Performance limited by data dependencies
Simple transformations can yield dramatic performance improvement
- Compilers often cannot make these transformations
- Lack of associativity and distributivity in floating-point arithmetic
Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct {
    size_t len;
    data_t *data;
} vec;

// retrieve vector element and store at val
int get_vec_element(vec *v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Benchmark Computation

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

Cycles Per Element (CPE)

Convenient way to express performance of program that operates on vectors or lists

\[ T = \text{CPE} \times n + \text{Overhead} \]

CPE is slope of line

Benchmark Performance

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>22.68</td>
<td>20.02</td>
<td>19.98</td>
</tr>
<tr>
<td>Combine1 -O1</td>
<td>10.12</td>
<td>10.12</td>
<td>10.17</td>
</tr>
</tbody>
</table>
Basic Optimizations

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

```c
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

Effect of Basic Optimizations

- Eliminates sources of overhead in loop

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine1 --O1</td>
<td>10.12</td>
<td>10.12</td>
</tr>
<tr>
<td>Combine4</td>
<td>1.27</td>
<td>3.01</td>
</tr>
</tbody>
</table>