

Schedule...

- **Today: Lab 2 due**
- **No Lab 3**
- **Use bonus free time during the next two weeks to study for Exam 1**

Last Time

- **Linux / x86 memory layout**
- **C operators and type declarations**
- **Buffer overflow exploits**

Code Optimization I: Machine Independent Optimizations

- **Topics**
 - **Machine-Independent Optimizations**
 - » Code motion
 - » Reduction in strength
 - » Common subexpression sharing
 - **Tuning**
 - » Identifying performance bottlenecks

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**
 - code selection and ordering
 - eliminating minor inefficiencies
 - function inlining, constant propagation, dead code elimination
 - register allocation
 - pipeline scheduling
- **Almost never improve asymptotic efficiency**
 - up to programmer to select best overall algorithm
 - big-O savings are (often) more important than constant factors
 - » particularly for very large data sets
 - » 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 change behavior of a “correct program” → correctness
 - Often prevents it from making optimizations which would only affect behavior under pathological conditions
 - » compiler has no way of knowing that pathological conditions won't occur
- **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
 - » plus it's often difficult to determine optimal strategy
- **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

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

What else can be pulled out of the inner loop?

Compiler-Generated Code Motion

- Most compilers do a good job with array code + 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];
```

```
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*x --> x << 4
 - » Utility of this optimization is machine dependent
 - » Depends on cost of multiply or divide instruction
 - shift or add is usually a single cycle operation
 - » On Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products
 - » turn them into a sequence of adds

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



```
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

- **Limitation**

- Compiler not always able to determine whether variable can be held in register
- Possibility of *Aliasing*
 - » let's say we want to compute $x = x + 2y$ using pointers

```
*xp += *yp  
*xp += *yp
```

 → 6 memory references

can a smart compiler replace this with

```
*xp += (*yp << 1)
```

 → 3 memory references ??

NO! Consider what happens when $xp == yp$

1st Rule: compiler must be correct

Machine-Independent Opts. (Cont.)

- **Share Common Subexpressions**

- Reuse portions of expressions
- Compilers often 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;
```

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

```
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
```

How can we improve on this?

```
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;
```

1 multiplication: $i*n$

Aside: Time Scales

- **Absolute Time**

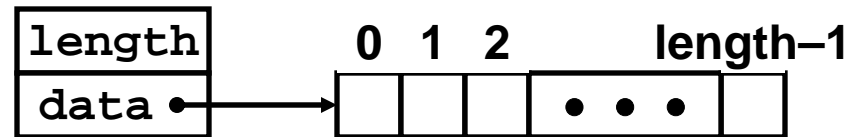
- Typically use nanoseconds
 - » 10^{-9} seconds
- Time scale of computer instructions

- **Clock Cycles**

- Most computers controlled by high frequency clock signal
- Typical Range
 - » 1 MHz
 - 10^6 cycles per second
 - Clock period = 1000 ns
 - Embedded processor
 - » 3 GHz
 - 3×10^9 cycles per second
 - Clock period = 0.33 ns
 - Desktop or server

Vector ADT

(abstract data type)



- **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 vec_length(vec_ptr v)`

- » Returns of the vector

`int *get_vec_start(vec_ptr v)`

- » Returns a pointer to the start of the 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;
    }
}
```

- **Procedure**

- Compute sum of all elements of vector
- Store result at destination location

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

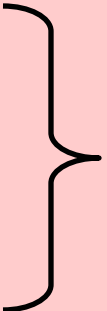
- **Pentium II/III Performance: Clock Cycles / Element**

- 42.06 (Compiled -g) 31.25 (Compiled -O2)

Understanding Loop

```
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:
}
```

1 iteration



- **Inefficiency**

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

Move `vec_length` Call Out of Loop

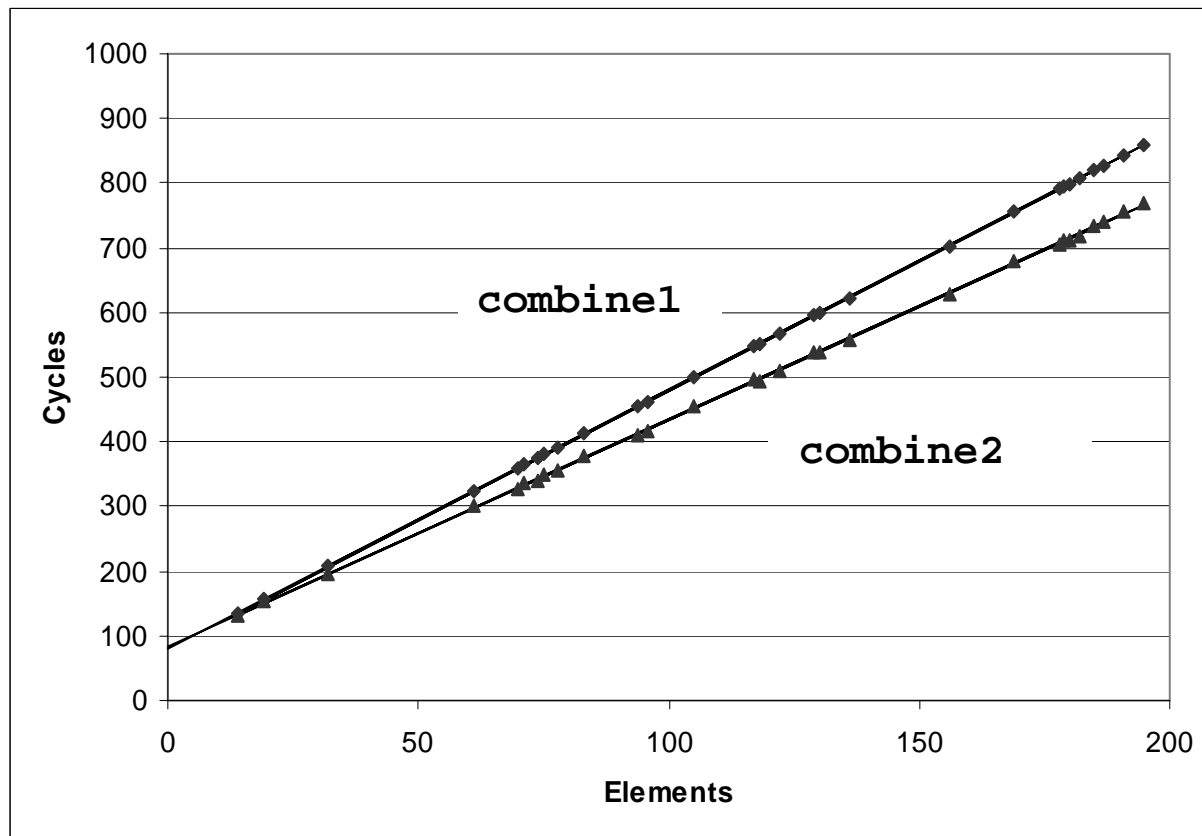
```
void combine2(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

- **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

Cycles Per Element

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- $T = CPE * n + \text{Overhead}$



Code Motion Example #2

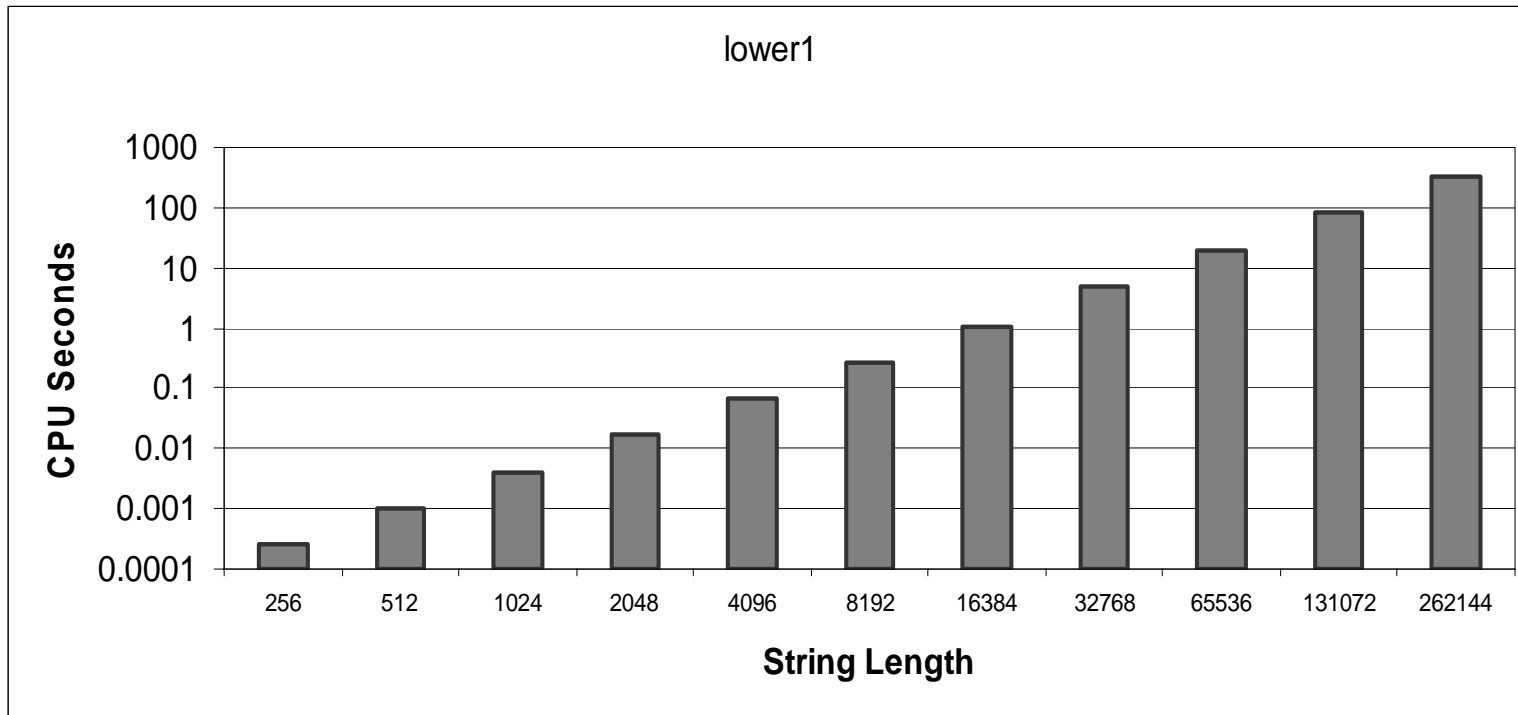
- Procedure to Convert String to Lower Case

```
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 an actual student's lab code

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance
 - » note semi-log scale
 - » sure the code was a little stupid but how can you end up w/ quadratic???



Convert Loop To Goto Form

```
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'`
- `lower` is linear in string length as well → quadratic overall

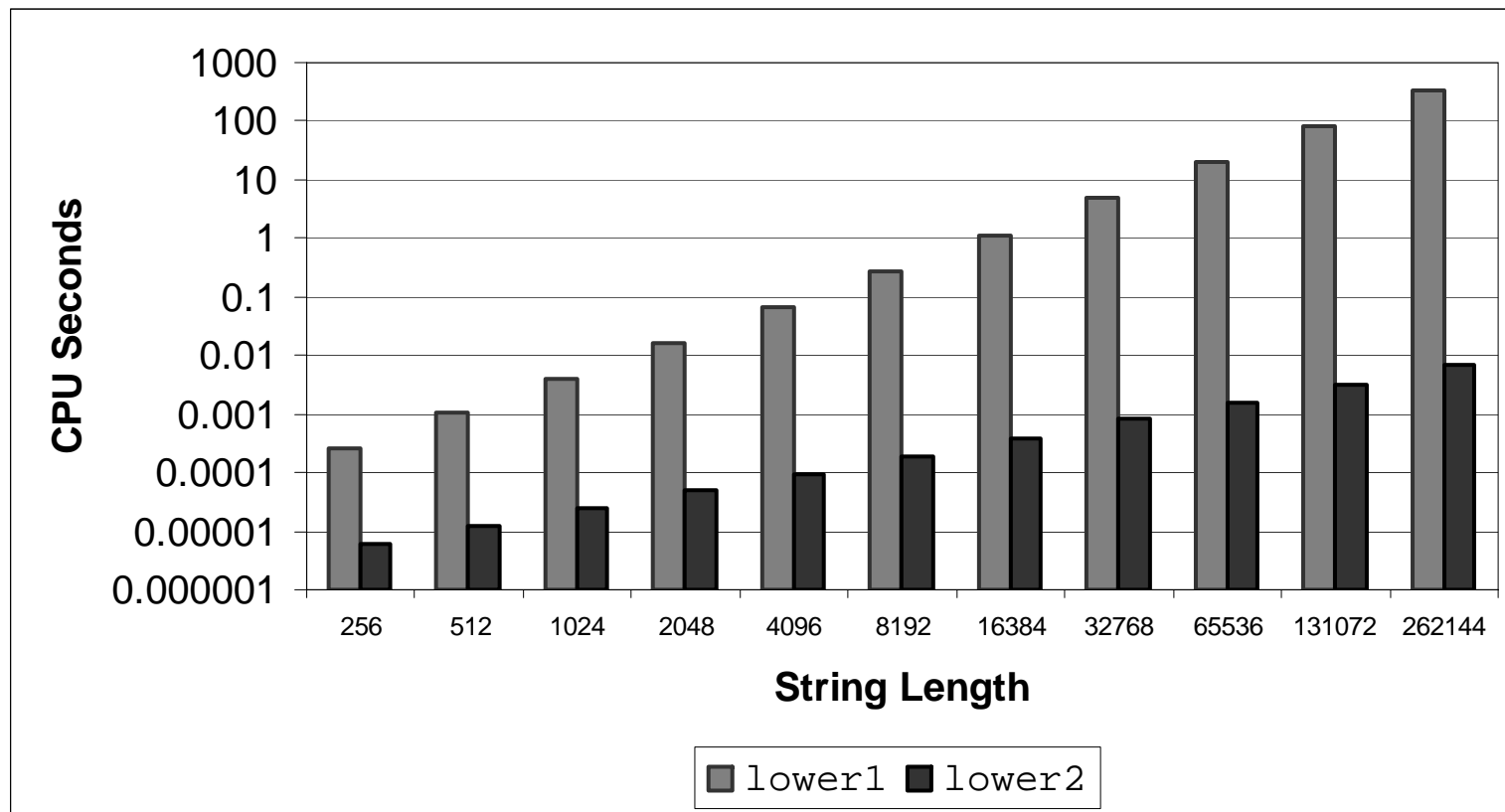
Improving Performance

```
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
 - » changes Big-O complexity from quadratic to linear
- **Form of code motion**
- **What else can be done?**
 - » 'A' – 'a' can be brought outside the inner loop
 - » 2 `s[i]`'s can be turned into one and lifted out of the if statement
 - » both do not change the Big-O complexity but do affect overhead

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance



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 often treats procedure call as a black box
 - » Results in weak optimizations in and around them

Reducing Cost of Procedure Calls

- **Idea: replace a call to a function with the body of the function – “inlining”**
 - **Minor advantage: Eliminates prologue and epilogue**
 - **Major advantage: Gives most of the benefits of interprocedural analysis without most of the costs**
 - **Disadvantages?**
 - **When does inlining break a program?**
 - **How would you decide which functions to inline?**

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**
 - » **Within loop just do pointer reference**
 - » **Not as clean in terms of data abstraction**
- **CPE: 6.00 (Compiled -O2) [as opposed to 20.66 with -O2 before]**
 - » **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**

- 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) [3x improvement in overhead]
 - » 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 clock cycles
- » addl must read and write memory!

- **Combine4**

- » 4 instructions in 2 clock cycles

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
 - since cost mode compares multiply vs. shift and add times
 - 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
 - » primarily due to cost vs. benefit issues

Important Tools

- **Observation**

- **Looking at assembly code**

- » Lets you see what optimizations compiler can make

- » Understand capabilities/limitations of particular compiler

- **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

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**

29,801	the
27,529	and
21,029	I
20,957	to
18,514	of
15,370	a
14010	you
12,936	my
11,722	in
11,519	that

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)**
 - **note sampling error opportunity**
 - **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

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
86.60	8.21	8.21	1	8210.00	8210.00	sort_words
5.80	8.76	0.55	946596	0.00	0.00	lower1
4.75	9.21	0.45	946596	0.00	0.00	find_ele_rec
1.27	9.33	0.12	946596	0.00	0.00	h_add

- **Call Statistics**

- Number of calls and cumulative time for each function

- **Performance Limiter**

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

» hence if you want to speed this code up you'll need to focus on `sort_words`

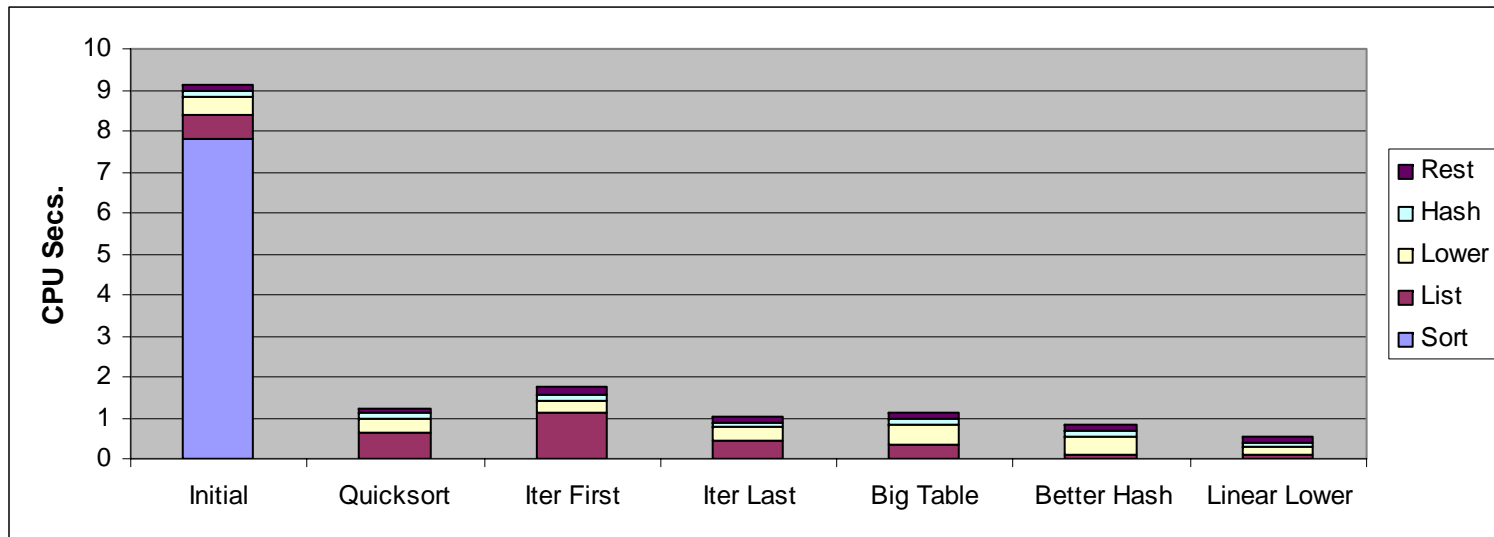
Amdahl's Law

- Just in case you forgot something from CS3810

$$\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

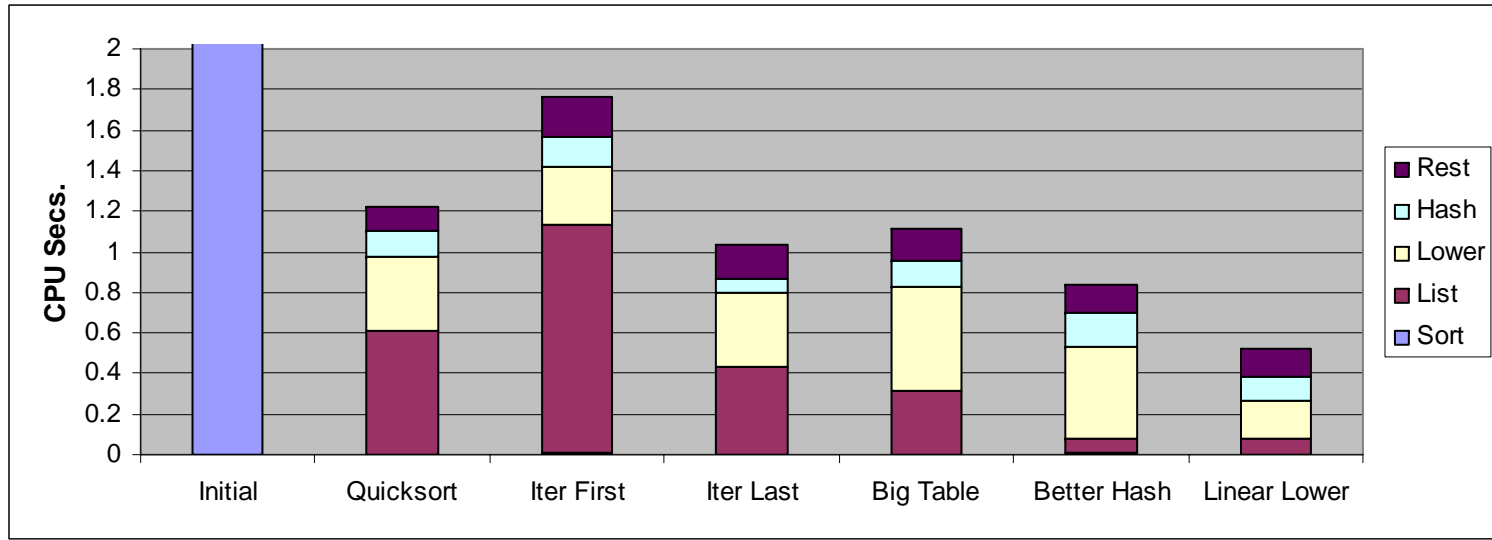
- Interesting to consider $\text{speedup}_{\text{enhanced}} = \text{infinite}$
 - $\rightarrow \text{speedup} = 1/(1 - \text{frac}_{\text{enh}})$
 - for `sort_words`
 - » $1/(1 - .866) = 746\%$

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 at least a few seconds
 - » statistical error due to sampling model
 - wake up every 10 ms and add one to whoever is running
 - → short things may not get seen
 - error goes to 0 for an infinitely long run