# 15-213

*"The course that gives CMU its Zip!"*

## Code Optimization I: Machine Independent Optimizations Feb 11, 2003

Topics

- Machine-Independent Optimizations
  - Code motion
  - Strength Reduction/Induction Var Elim
  - 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

- 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

er

les

gest

The Bottom Line:

When in doubt, do nothing
i.e., The compiler must be conservative.

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

# Machine-Independent Optimizations

- Optimizations that should be done 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];
}
```



# 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
```



# Strength Reduction†

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x` → `x << 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 (induction var analysis)

```
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;
}
```

†As a result of Induction Variable Elimination



# Make Use of Registers

- Reading and writing registers much faster than reading/writing memory

## Limitation

- Limited number of registers
- Compiler cannot always 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

```
/* 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 multiplies: i*n, (i–1)*n, (i+1)*n

```
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 multiply: 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
```

†AKA: Common Subexpression Elimination (CSE)



# Measuring Performance: 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
  - 100 MHz
    - » $10^8$ cycles per second
    - » Clock period = 10ns
  - 2 GHz
    - » 2 X $10^9$ cycles per second
    - » Clock period = 0.5ns
- Fish machines: 550 MHz (1.8 ns clock period)



# Measuring Performance

## For many programs, cycles per element (CPE)

- Especially true of programs that work on lists/vectors
- Total time = fixed overhead + CPE * length-of-list

```
void vsum1(int n)
{
  int i;

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

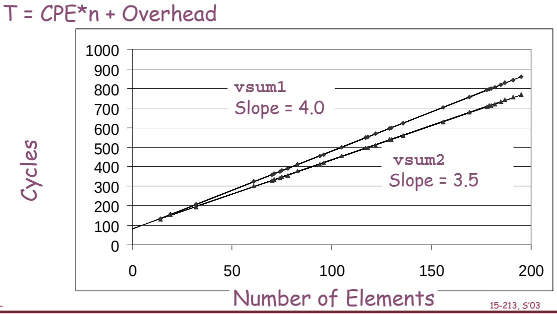
```
void vsum2(int n)
{
  int i;

  for (i = 0; i<n; i+=2)
    c[i] = a[i] + b[i];
    c[i+1] = a[i+1] + b[i+1];
}
```

- vsum2 only works on even n.
- vsum2 is an example of loop unrolling.

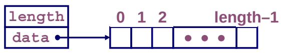


# Cycles Per Element

- Convenient way to express performance of a program that operates on vectors or lists
- Length = n
- T = CPE*n + Overhead



## Vector ADT




### 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

`int vec_length(v)(vec_ptr v)`
- Return length of vector

- 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 Perf: 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;                      1 iteration
  loop:
    get_vec_element(v, i, &val);
    *dest += val;
    i++;
    if (i < vec_length(v))
      goto loop
  done:
}
```

### 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



## Code Motion Example #2

### Procedure to Convert String to Lower Case

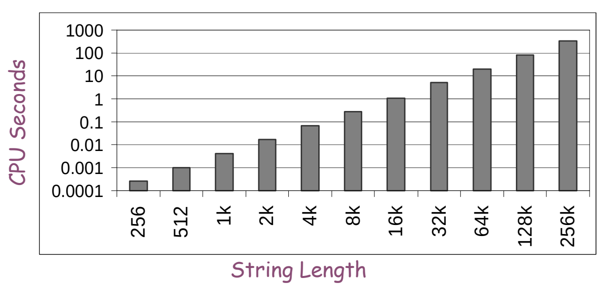
```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- Extracted from 213 lab submissions, Fall, 1998



## Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance of lower






## 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'`
- Overall performance is quadratic

# Improving Performance

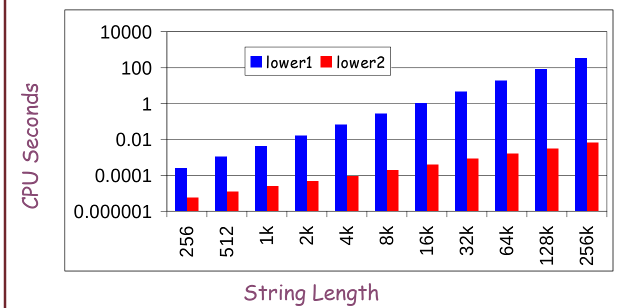
```
void lower(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion



# Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2






# Optimization Blocker: Procedure Calls

Why doesn't the compiler move `vec_len` or `strlen` out of the inner loop?

Why doesn't compiler look at code for `vec_len` or `strlen`?



# Optimization Blocker: Procedure Calls

Why doesn't the compiler move `vec_len` or `strlen` out of the inner loop?

- Procedure may have side effects
  - Can alter global state each time called
- Function may return diff value for same arguments
  - Depends on other parts of global state
  - Procedure `lower` could interact with `strlen`
- GCC has an extension for this:
  - `int square (int) __attribute__ ((const));`
  - Check out info.

Why doesn't compiler look at code for `vec_len` or `strlen`?

## Optimization Blocker: Procedure Calls

Why doesn't the compiler move `vec_len` or `strlen` out of the inner loop?

- Procedure may have side effects
- Function may return diff value for same arguments

Why doesn't compiler look at code for `vec_len` or `strlen`?

- Linker may overload with different version
  - Unless declared static
- Interprocedural opt isn't used extensively due to cost

Warning:

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



## What next?

```
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;
  }
}
```



## Reduction in Strength

Anything else?

```
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];
}
```

Aside: Rational for Classes

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

```
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.

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 one location

## Example

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

## Observations

- Can easily 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/Loop Invariant Code Motion

- Compilers good if for simple loop/array structures
- Bad in presence of procedure calls and memory aliasing

## Strength Reduction/Induction Var Elimination

- 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/CSE

- 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 words sorted from most to least frequent

## 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 freq 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

## Add information gathering to executable

- Computes (approximate) 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 collect number of times each function is 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

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  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

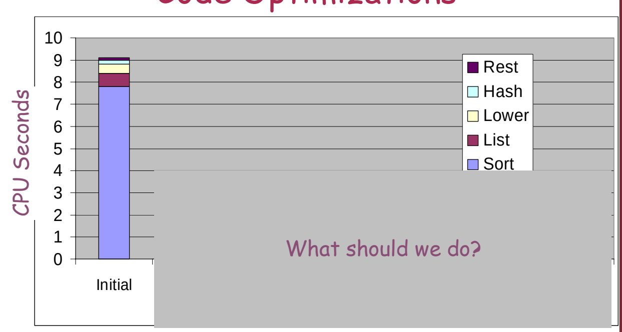
- Number of calls and cumulative time for each function

## Performance Limiter

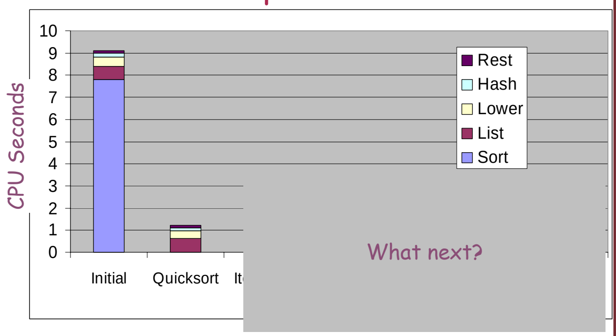
- Using inefficient sorting algorithm
- Single call uses 87% of CPU time



# Code Optimizations



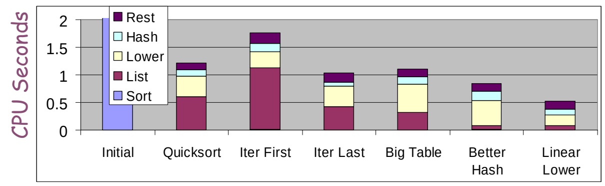
## Code Optimizations




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



## Further Optimizations




- Iter first: Use iterative func to insert elments into linked list
- Iter last: Iterative func, places new entry at end 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

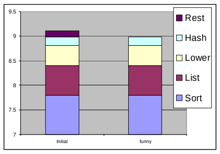


## How Much Effort Should we Expend?

Amdahl's Law:

Overall performance improvement is a combination
- How much we sped up a piece of the system
- How important that piece is!

Example, suppose Chose to optimize "rest" & you succeed! It goes to ZERO seconds!



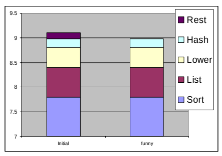
# How Much Effort Should we Expend?

Amdahl's Law:

Overall performance improvement is a combination

- How much we sped up a piece of the system
- How important that piece is!

Example, suppose Chose to optimize "rest" & you succeed! It goes to ZERO seconds!




Amdahl's Law

- Total time = $(1-\alpha)T + \alpha T$
- Component optimizing takes $\alpha T$ time.
- Improvement is factor of k, then:
- $T_{new} = T_{old}[(1-\alpha) + \alpha/k]$
- Speedup = $T_{old}/T_{new} = 1/ [(1-\alpha) + \alpha/k]$
- Maximum Achievable Speedup ($k = \infty$) = $1/(1-\alpha)$



# A Stack Based Optimization

```
_fib:
        pushl  %ebp
        movl   %esp,%ebp
        subl   $16,%esp
        pushl  %esi
        pushl  %ebx
        movl   8(%ebp),%ebx
        cmpl   $1,%ebx
        jle    L3
        addl   $-12,%esp
        leal   -1(%ebx),%eax
        pushl  %eax
        call   _fib
        movl   %eax,%esi
        addl   $-12,%esp
        leal   -2(%ebx),%eax
        pushl  %eax
        call   _fib
        addl   %esi,%eax
        jmp    L5
        .align 4
```

```
        .align 4
L3:
        movl   $1,%eax
L5:
        leal   -24(%ebp),%esp
        popl   %ebx
        popl   %esi
        movl   %ebp,%esp
        popl   %ebp
        ret
```

```
int fib(int n)
{
    if (n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
```