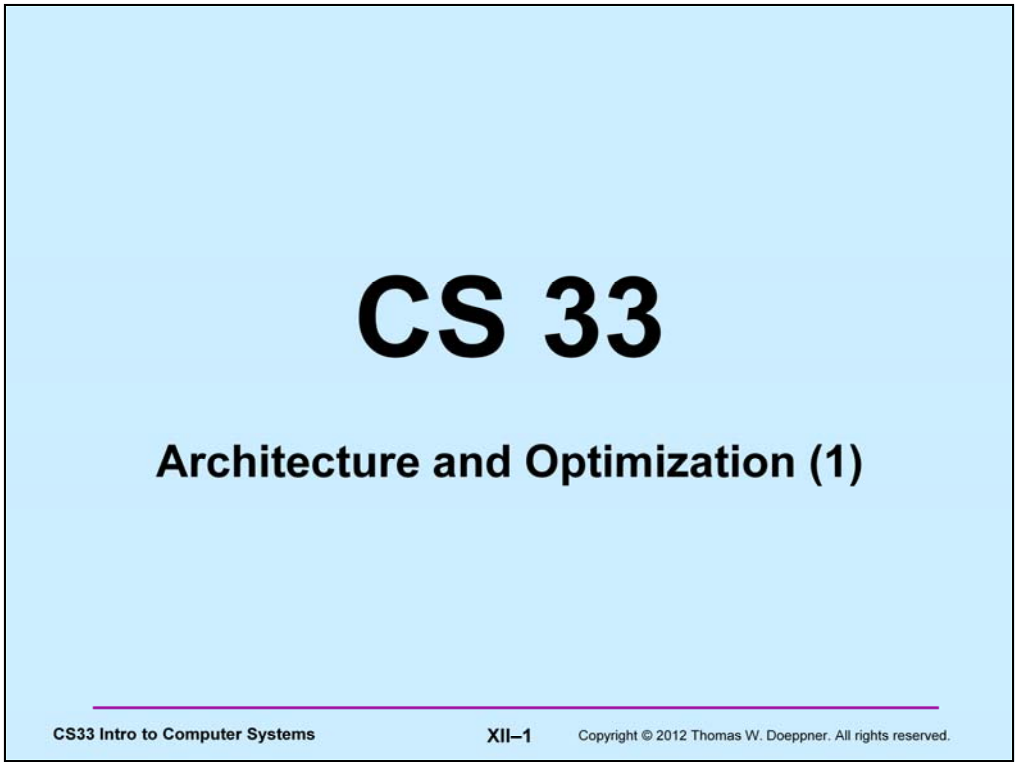


Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

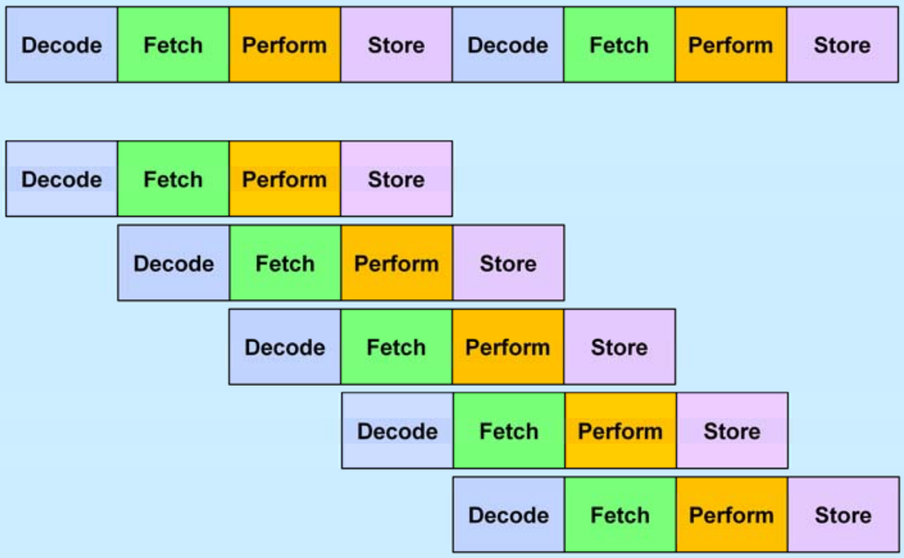
Simplistic View of Processor

```
while (true) {
  instruction = mem[eip];
  execute(instruction);
}
```

Some Details ...

```
void execute(instruction_t instruction) {
  decode(instruction, &opcode, &operands);
  fetch(operands, &in_operands);
  perform(opcode, in_operands, &out_operands);
  store(out_operands);
}
```

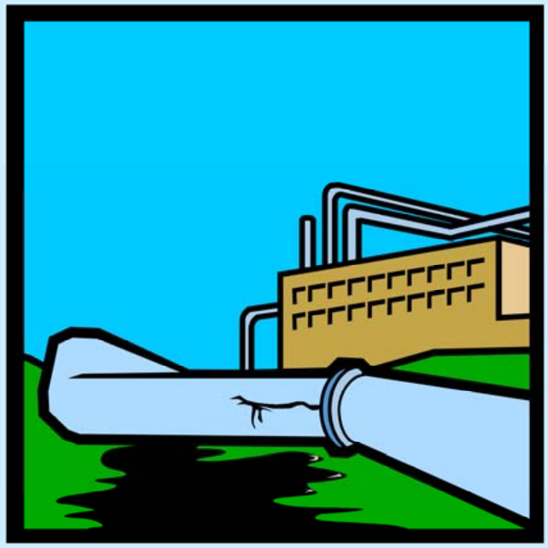
Pipelines


Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store
Decode
Fetch
Perform
Store

Analysis

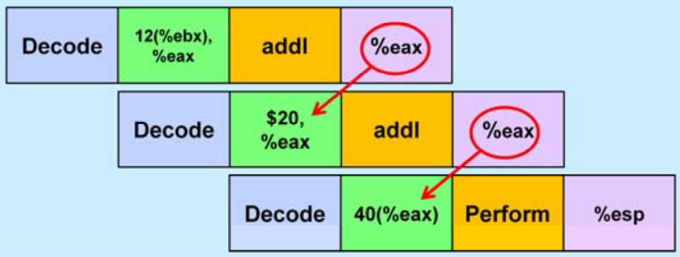
- **Not pipelined**
  - each instruction takes, say, 320 nanoseconds
    - » 320 ns latency
  - 3.125 billion instructions/second (GIPS)
- **Pipelined**
  - each instruction still takes 320 ns
    - » latency still 320 ns
  - an instruction completes every 80 ns
    - » 12.5 GIPS throughput

Hazards ...

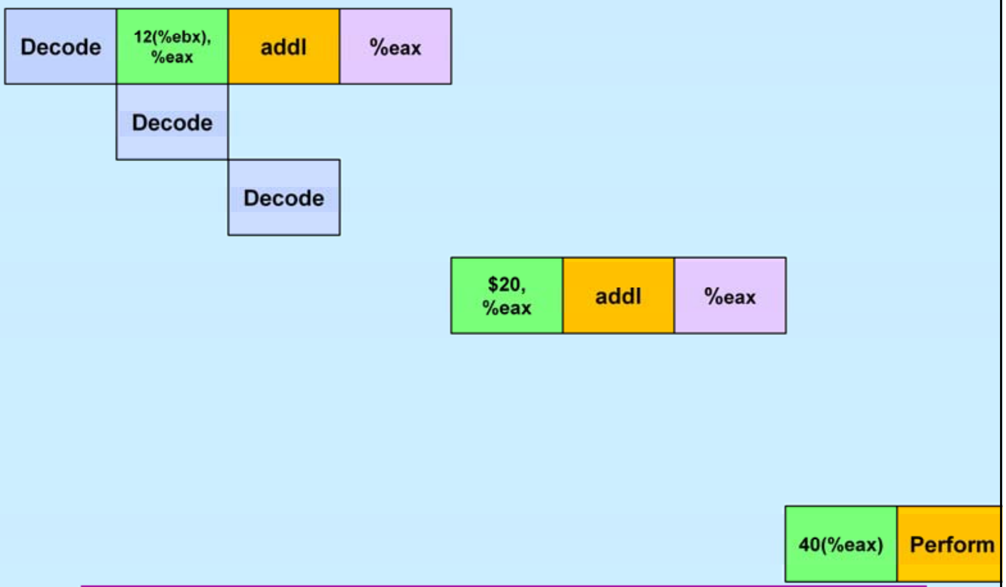


Data Hazards

```
addl 12(%ebx), %eax
addl $20, %eax
movl 40(%eax), %esp
```



Coping

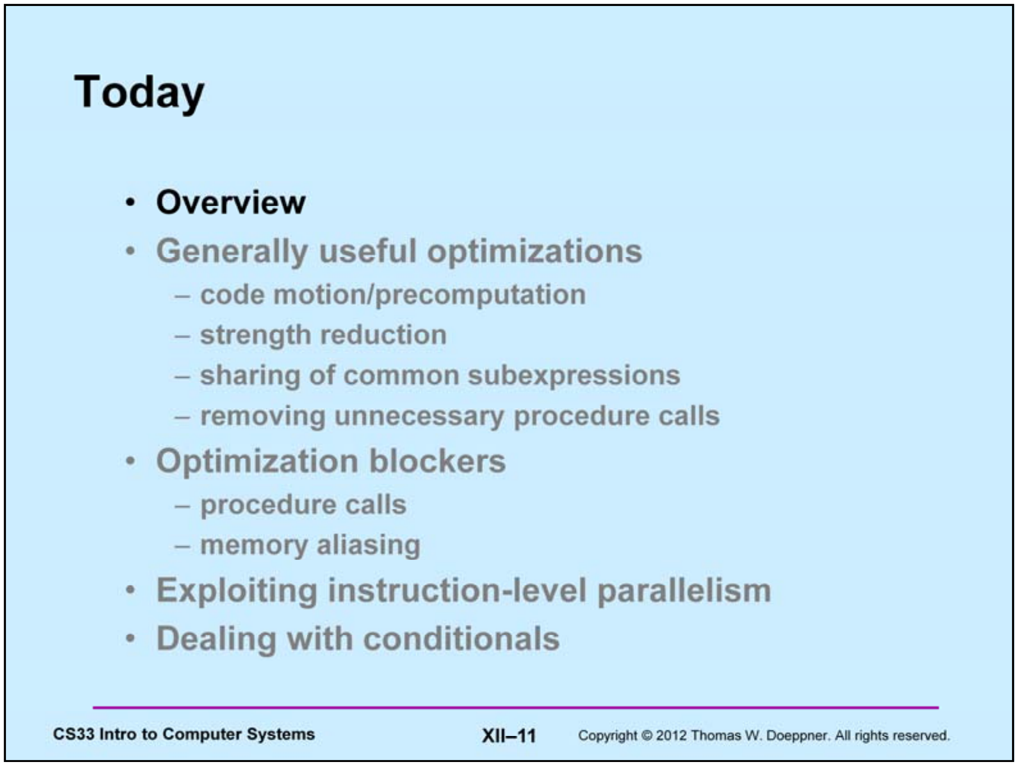

Decode
12(%ebx), %eax
addl
%eax
Decode
Decode
$20, %eax
addl
%eax
40(%eax)
Perform

Control Hazards

```
 movl $0, %ecx
.L2:
 movl %edx, %eax
 andl $1, %eax
 addl %eax, %ecx
 shrl %edx
 jne  .L2 # what goes in the pipeline?
 movl %ecx, %eax
 ...
```

Coping: Guess ...

- Branch prediction
  - assume, for example, that conditional branches are always taken
  - but don't do anything to registers or memory until you know for sure



Supplied by CMU.

Performance Realities

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**

Supplied by CMU.

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - 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

Supplied by CMU.

Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - must not cause any change in program behavior
  - 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

Supplied by CMU.

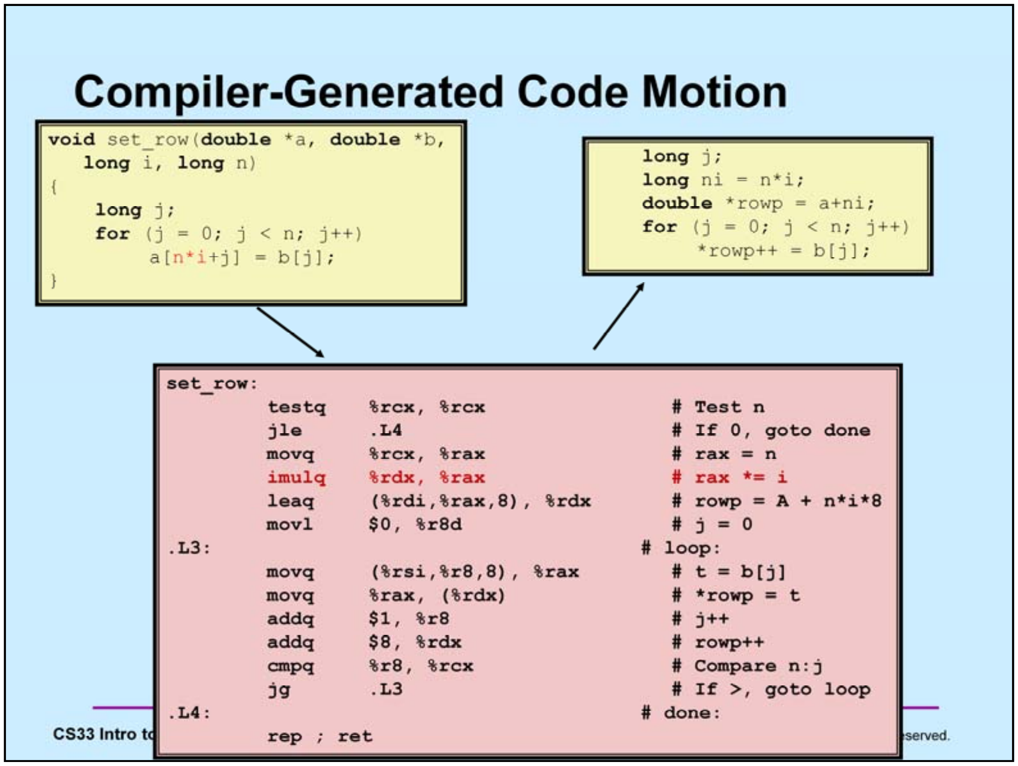
Generally Useful Optimizations

- **Optimizations that you or the compiler 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

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

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


Supplied by CMU.



Supplied by CMU.

Reduction in Strength

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

  `16*x      -->    x << 4`

  – utility is machine-dependent

  – depends on cost of multiply or divide instruction

    » on Intel Nehalem, integer multiply requires 3 CPU cycles
- **Recognize sequence of products**

```
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;
}
```


Supplied by CMU.

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;
```

```
long 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

```
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
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

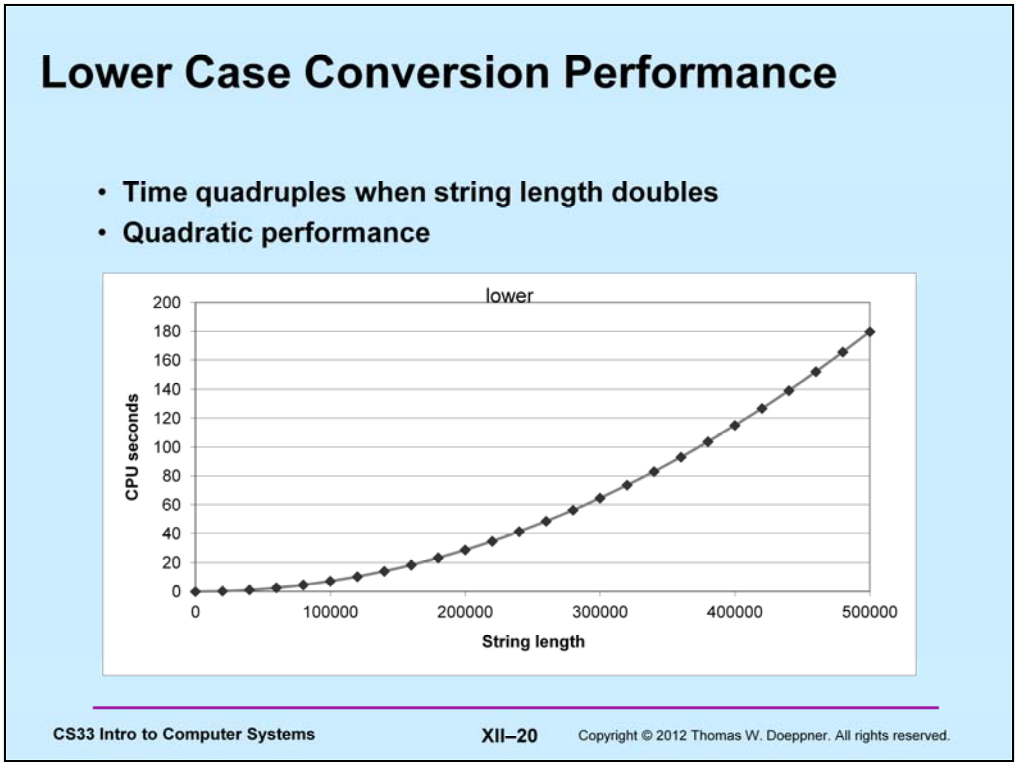

Supplied by CMU.

Optimization Blocker #1: Procedure Calls

- **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');
}
```


Supplied by CMU.



Supplied by CMU.

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**

Supplied by CMU.

Calling 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 null character
- Overall performance, string of length N
  - N calls to strlen
  - overall $O(N^2)$ performance

Supplied by CMU.

Improving Performance

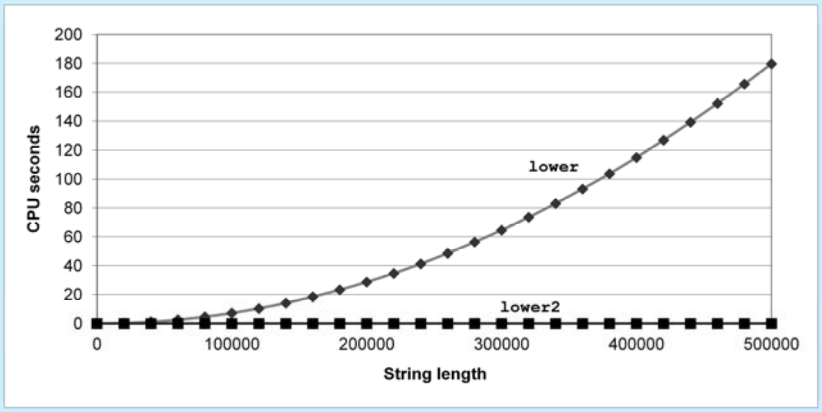
```c
void lower2(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- **Move call to `strlen` outside of loop**
  - since result does not change from one iteration to another
  - form of code motion

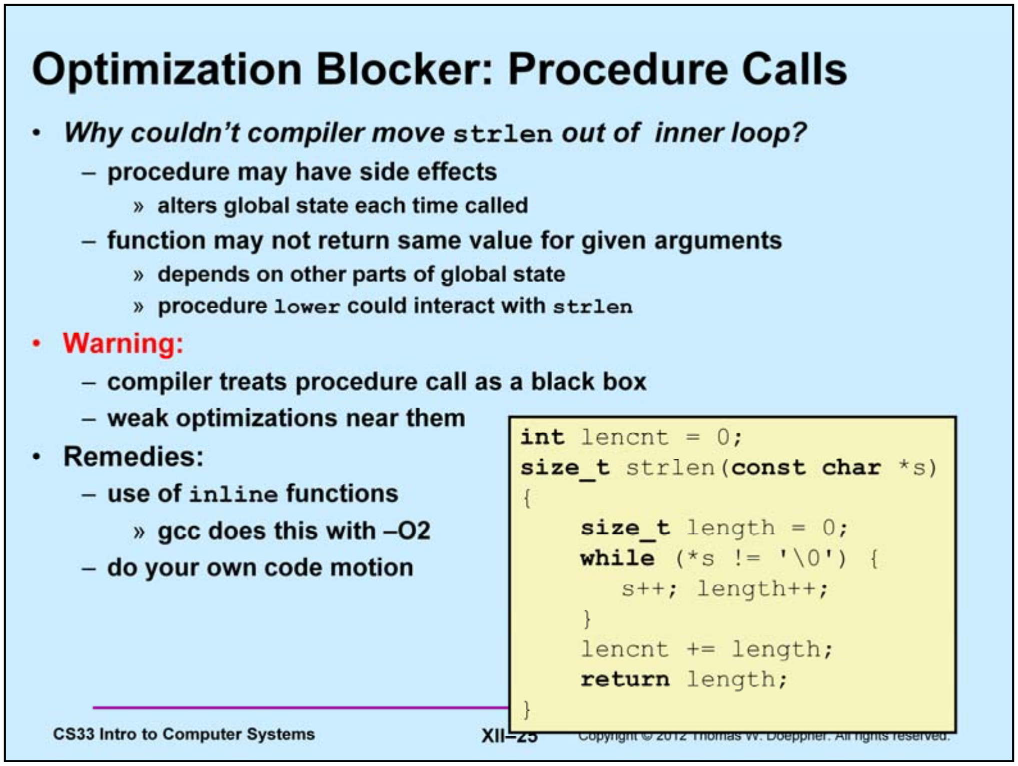
Supplied by CMU.

Lower-Case Conversion Performance

- **Time doubles when string-length doubles**
  - linear performance of lower2



Supplied by CMU.



Supplied by CMU.

Memory Matters

```
/* Sum rows of n X n matrix a
   and store result in vector b  */
void sum_rows1(double *a, double *b, 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];
    }
}
```

```
# sum_rows1 inner loop
.L3:
        addsd   (%rcx), %xmm0           # FP add
        addq    $1, %rax
        addq    $8, %rcx
        cmpq    %rdx, %rax
        movsd   %xmm0, (%rsi,%r8,8)     # FP store
        jne     .L3
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

Supplied by CMU.

Note that *a* is passed as a 1-D array, but interpreted as a 2-D array. This isn't terribly good programming style (gcc, fortunately, refrains from commenting on one's style), but it is definitely the sort of program that gcc must be prepared to deal with.

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store result in vector b  */
void sum_rows1(double *a, double *b, 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[3] = A+3;

sum_rows1(A, B, 3);
```

Value of `B`:

```
init:  [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 22, 16]
i = 2: [3, 22, 224]
```

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

Supplied by CMU.

Removing Aliasing

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

```
# sum_rows2 inner loop
.L6:
        addq     $1, %rax
        addsd    (%rcx), %xmm0    # FP Add
        addq     $8, %rcx
        cmpq     %rdx,%rax
        jne      .L6
```

- **No need to store intermediate results**

Supplied by CMU.

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

Supplied by CMU.

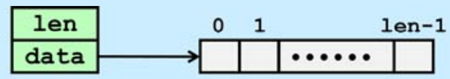
Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**
  - hardware can execute multiple instructions in parallel
- **Performance limited by data dependencies**
- **Simple transformations can have dramatic performance improvement**
  - compilers often cannot make these transformations
  - lack of associativity and distributivity in floating-point arithmetic

Supplied by CMU.

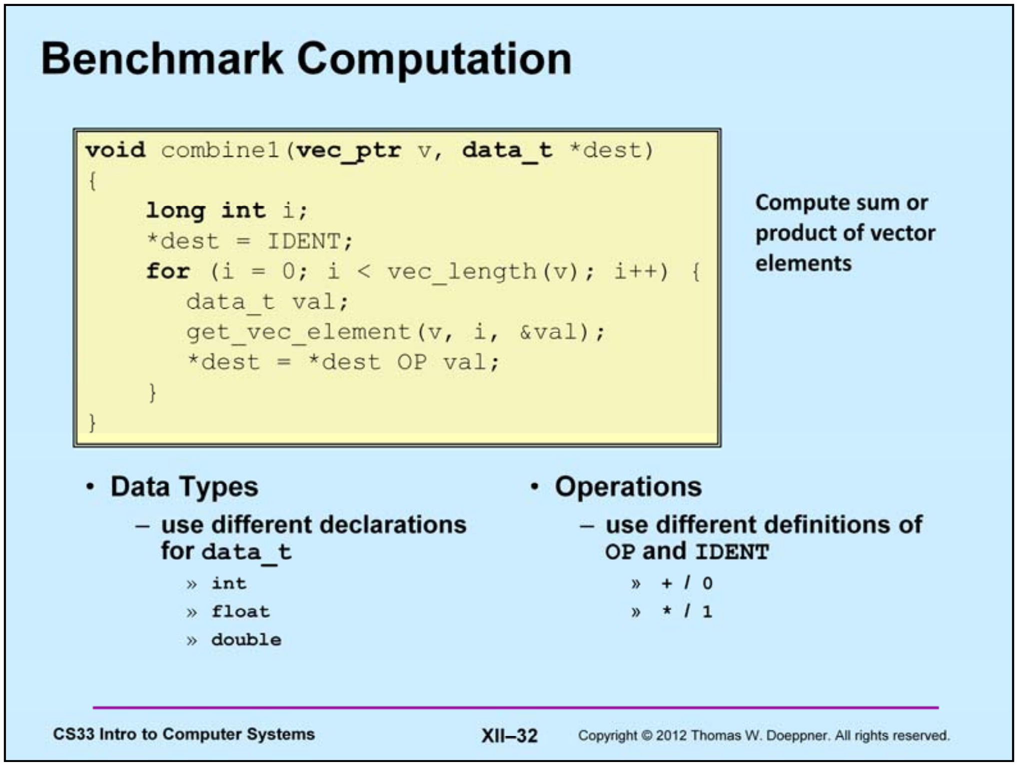
Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    int len;
    data_t *data;
} vec, *vec_ptr;
```




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

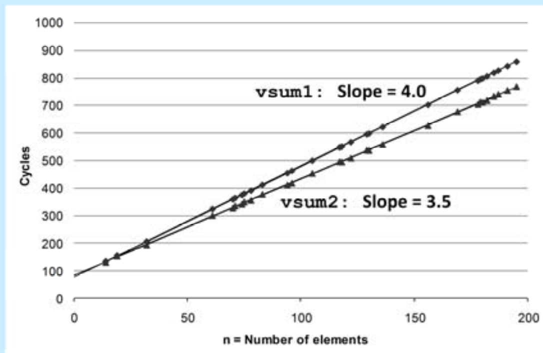

Supplied by CMU.



Supplied by CMU.

Cycles Per Element (CPE)

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



Supplied by CMU.

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;
    }
}
```

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	29.0	29.2	27.4	27.9
Combine1 –O1	12.0	12.0	12.0	13.0

Supplied by CMU.

Move vec_length

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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	29.0	29.2	27.4	27.9
Combine1 –O1	12.0	12.0	12.0	13.0
Combine2	8.03	8.09	10.09	12.08

Supplied by CMU.

Eliminate Procedure Calls

```
void combine3(vec_ptr v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
       *dest = *dest OP data[i];
    }
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine2	8.03	8.09	10.09	12.08
Combine3	6.01	8.01	10.01	12.02

Supplied by CMU.

Eliminate Unneeded Memory References

```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int 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;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	12.0	12.0	12.0	13.0
Combine4	2.0	3.0	3.0	5.0

Supplied by CMU.