CS 33

Machine Programming (6)

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.

Tail Recursion

```
int factorial(int x) {
  if (x == 1)
    return x;
  else
    return
      x*factorial(x-1);
}
```

```
int factorial(int x) {
  return f2(x, 1);
}

int f2(int a1, int a2) {
  if (a1 == 1)
    return a2;
  else
    return
      f2(a1-1, a1*a2);
}
```

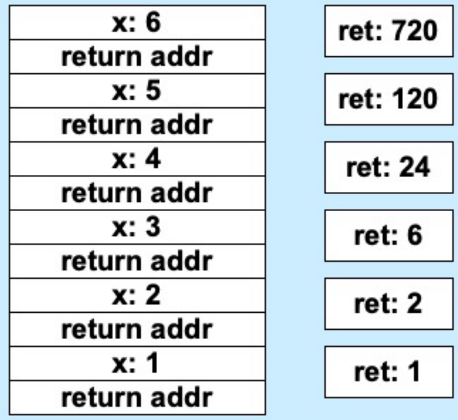

The slide shows two implementations of the factorial function. Both use recursion. In the version on the left, the result of each recursive call is used within the invocation that issued the call. In the second, the result of each recursive call is simply returned. This is known as *tail recursion*.

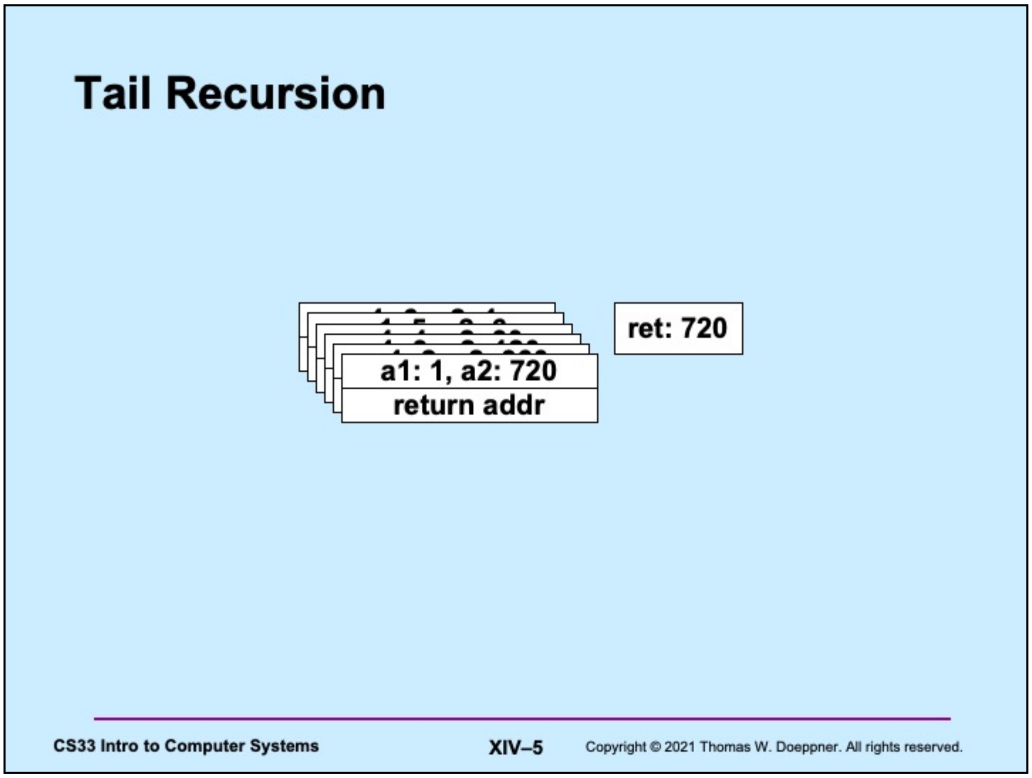
No Tail Recursion (1)

x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr

Here we look at the stack usage for the version without tail recursion. Note that we have as many stack frames as the value of the argument; the results of the calls are combined after the stack reaches its maximum size.

No Tail Recursion (2)





With tail recursion, since the result of the recursive call is not used by the issuing stack frame, it’s possible to reuse the issuing stack frame to handle the recursive invocation. Thus rather than push a new stack frame on the stack, the current one is written over. Thus the entire sequence of recursive calls can be handled within a single stack frame.

Code: gcc –O1

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        subq    $8, %rsp
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        call    f2        # recursive call!
        addq    $8, %rsp
.L5:
        rep
        ret
```


This is the result of compiling the tail-recursive version of factorial using gcc with the –O1 flag. This flags turns on a moderate level of code optimization, but not enough to cause the stack frame to be reused.

Code: gcc –O2

```
f2:
        cmpl    $1, %edi
        movl    %esi, %eax
        je      .L8
.L12:
        imull   %edi, %eax
        subl    $1, %edi            loop!
        cmpl    $1, %edi
        jne     .L12
.L8:
        rep
        ret
```


Here we've compiled the program using the –O2 flag, which turns on additional optimization (at the cost of increased compile time), with the result that the recursive calls are optimized away — they are replaced with a loop.

Why not always compile with –O2? For "production code" that is bug-free (assuming this is possible), this is a good idea. But this and other aggressive optimizations make it difficult to relate the runtime code with the source code. Thus, a runtime error might occur at some point in the program's execution, but it is impossible to determine exactly which line of the source code was in play when the error occurred.

Computer Architecture and Optimization (1)

What You Need to Know to Write Better Code

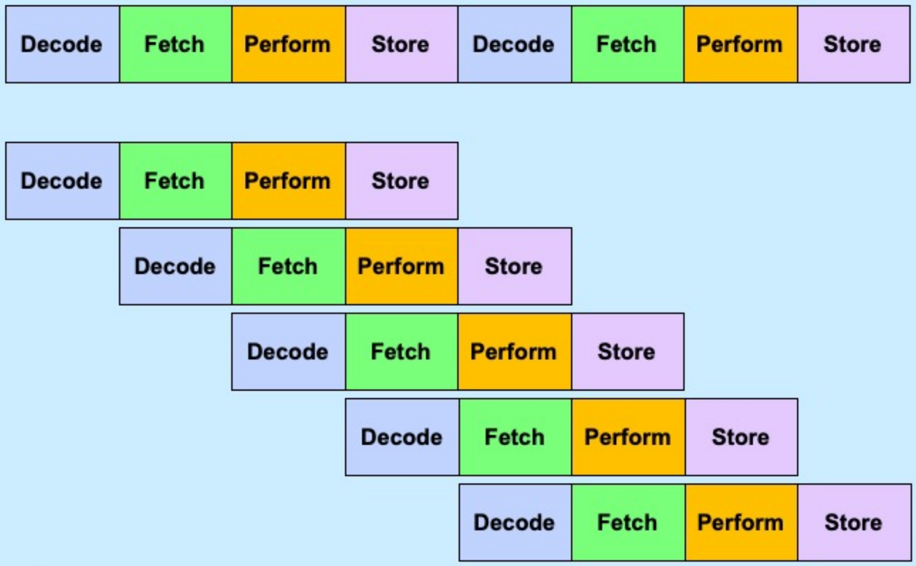
Simplistic View of Processor

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

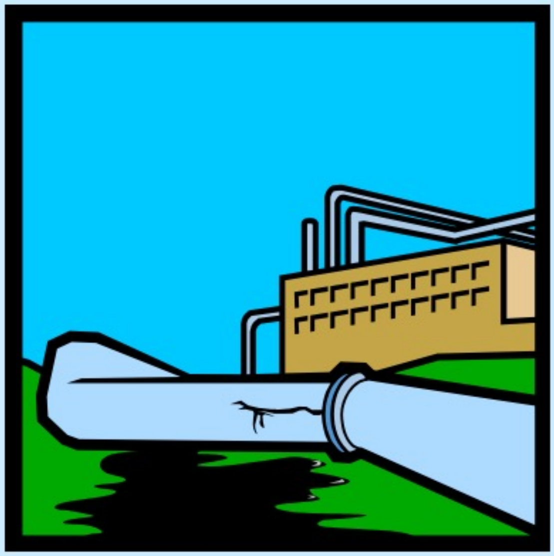
Pipelines


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, 3.2 nanoseconds
    - » 3.2 ns latency
  - 312.5 million instructions/second (MIPS)
- Pipelined
  - each instruction still takes 3.2 ns
    - » latency still 3.2 ns
  - an instruction completes every .8 ns
    - » 1.25 billion instructions/second (GIPS) throughput

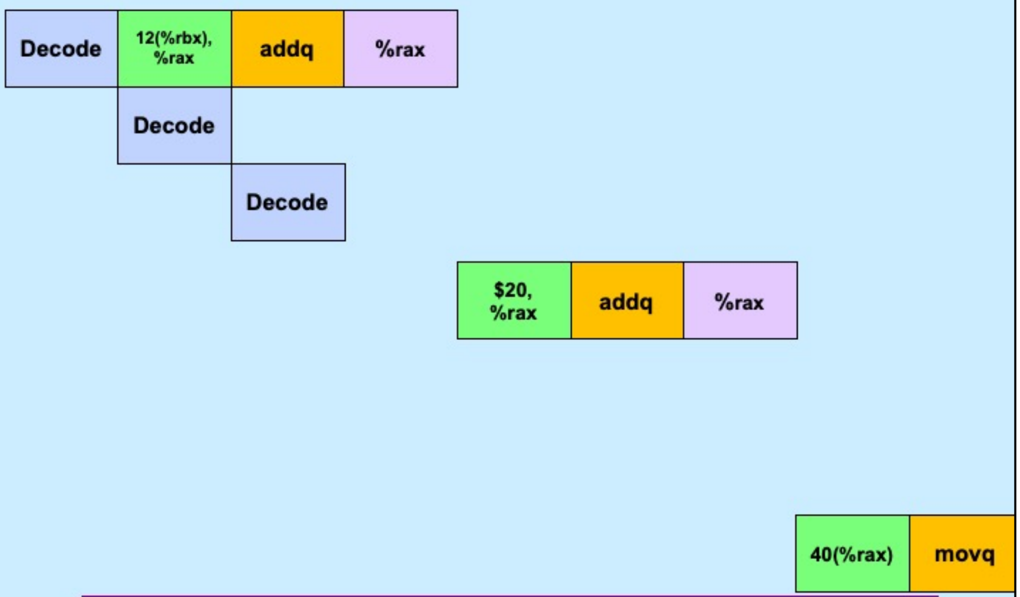
Hazards ...



Data Hazards

```
addq 12(%rbx), %rax
addq $20, %rax
movq 40(%rax), %rsp
```

Coping


Decode
12(%rbx), %rax
addq
%rax
Decode
Decode
$20, %rax
addq
%rax
40(%rax)
movq

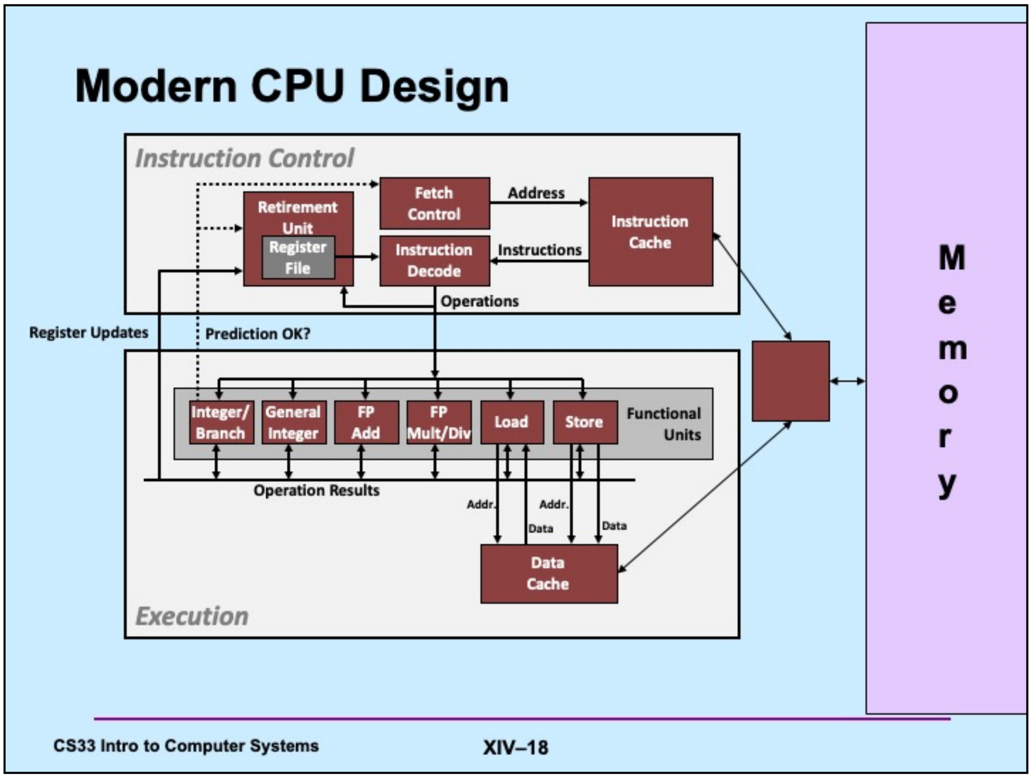
Control Hazards

```
    movl $0, %ecx
.L2:
    movl %edx, %eax
    andl $1, %eax
    addl %eax, %ecx
    shrl $1, %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

Modern processors have sophisticated algorithms for doing "branch prediction".



Adapted from slide supplied by CMU.

Note that the functional units operate independently of one another. Thus, for example, the floating-point add unit can be working on one instruction, which the general integer unit can be working on another. Thus, there are additional possibilities for parallel execution of instructions.

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, functions, 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)
  - 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 function 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 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 functions**
  - 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(long *a, long *b,
   long i, long n){
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

→

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


Supplied by CMU.

Compiler-Generated Code Motion

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

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

```
set_row:
        testq   %rcx, %rcx              # Test n
        jle     .L1                     # If 0, goto done
        imulq   %rcx, %rdx              # i *= n
        leaq    (%rdi,%rdx,8), %rdi     # rowp = A + n*i*8
        movl    $0, %eax                # j = 0
.L3:                                   # loop:
        movq    (%rsi,%rax,8), %rdx     # t = b[j]
        movq    %rdx, (%rdi,%rax,8)     # rowp[j] = t
        addq    $1, %rax                # j++
        cmpq    %rcx, %rax              # Compare n:j
        jg      .L3                     # If >, goto loop
.L1:                                   # done:
        rep ; ret
```


Supplied by CMU, updated for current gcc. This code was compiled by gcc with the –O1 flag, specifying that the compiler perform a small amount of code optimization.

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 some Intel processors, multiplies are 3x longer than adds

- **Recognize sequence of products**

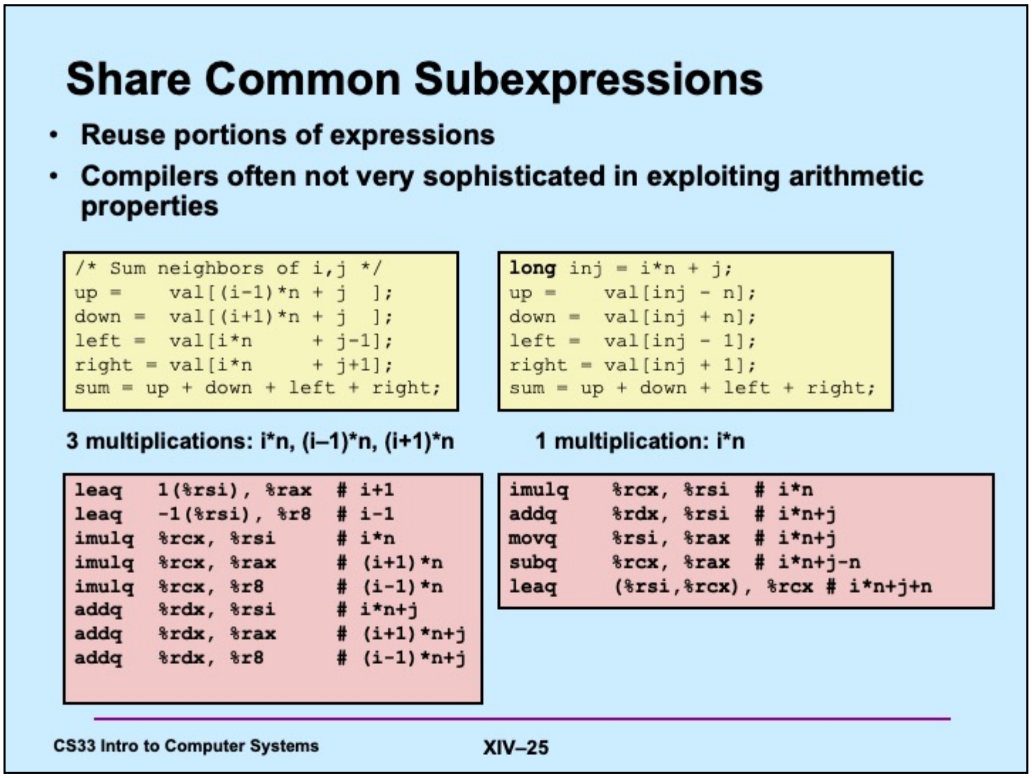
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
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.

gcc does optimizations of the sort shown here.



Supplied by CMU.

The code in the lower-left box is what gcc produced for the code in the upper left box. On the right is a much better version that was done by hand.

Quiz 1

The fastest means for evaluating

```
n*n + 2*n + 1
```

requires exactly:

a) 2 multiplies and 2 additions

b) one multiply and two additions

c) one multiply and one addition

d) three additions

Optimization Blocker #1: Function Calls

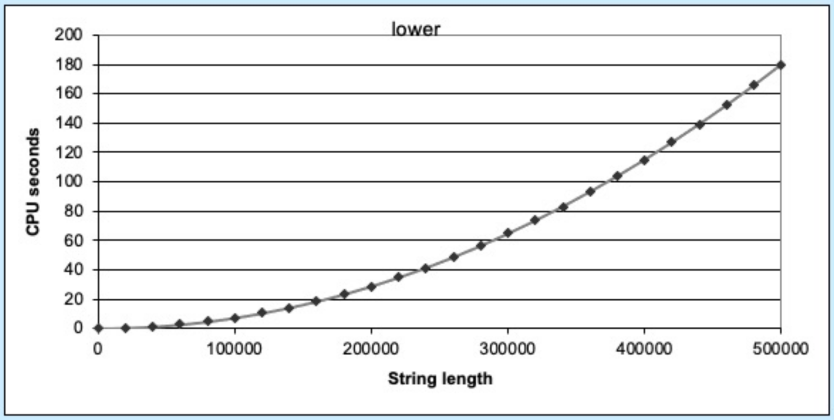
- **Function to convert string to lower case**

```
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.

Lower Case Conversion Performance

- Time quadruples when string length doubles
- Quadratic performance



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

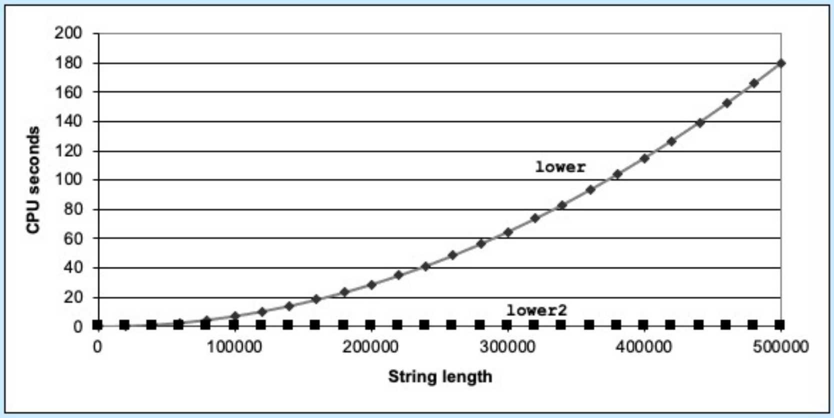
```
void lower2(char *s){
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- **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**



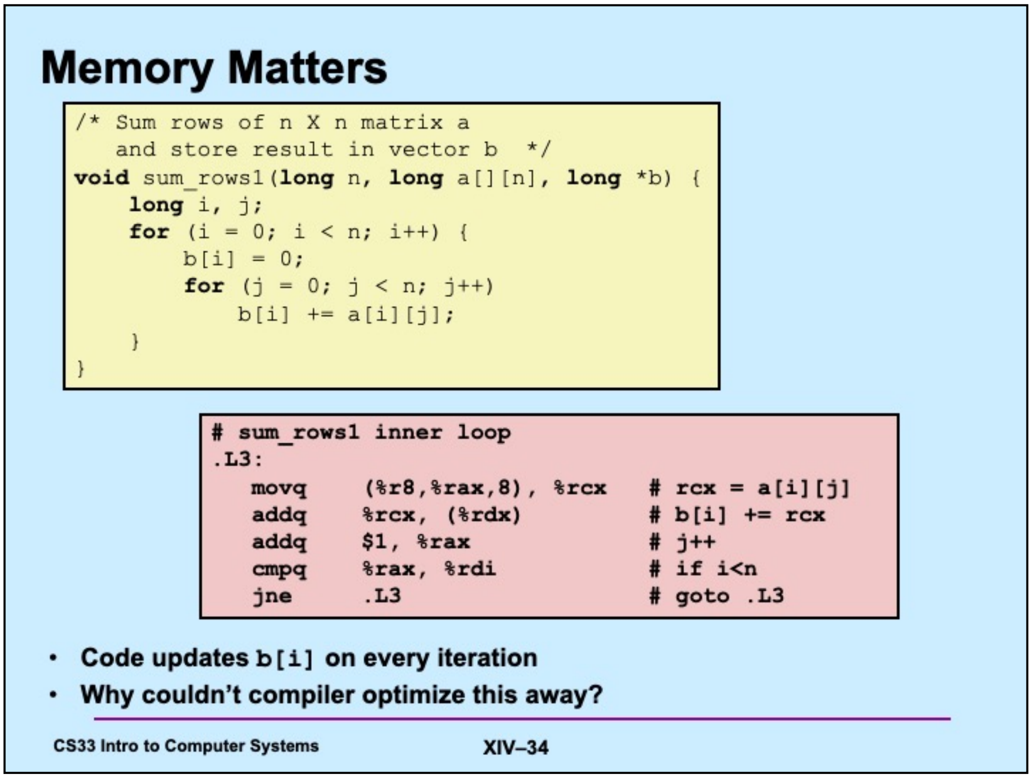
Supplied by CMU.

Optimization Blocker: Function Calls

- ***Why couldn't compiler move `strlen` out of inner loop?***
  - function 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
    - » function `lower` could interact with `strlen`
- **Warning:**
  - compiler treats function call as a black box
  - weak optimizations near them
- **Remedies:**
  - use of `inline` functions
    - » gcc does this with –O2
  - do your own code motion

```
int lencnt = 0;
size_t strlen(const char *s){
    size_t length = 0;
    while (*s != '\0') {
       s++; length++;
    }
    lencnt += length;
    return length;
}
```


Supplied by CMU.



Based on a slide supplied by CMU.

The issue here is whether it's really necessary to update the memory holding b[i] on every iteration of the inner for loop. Couldn't the value of b[i] be put in a register, updated there, then written to memory after the loop completes? Keep in mind that storing to memory is much more time-intensive than storing to a register.

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store result in vector b   */
void sum_rows1(long n, long a[][n], long *b) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i][j];
    }
}
```

```
int A[3][3] =
  {{ 0,   1,   2},
   { 4,   8,  16},
   {32,  64, 128}};

int *B = &A[1][0];

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

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, updated for current gcc.

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store result in vector b  */
void sum_rows1(long n, long a[][n], long *b) {
    long i, j;
    for (i = 0; i < n; i++) {
        long val = 0;
        for (j = 0; j < n; j++)
            val += a[i][j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L4:
   addq    (%r8, %rax, 8)), %rcx
   addq    $1, %rdi
   cmpq    %rcx, %rdi
   jne     .L4
```

- **No need to store intermediate results**

Supplied by CMU.

Note that the programmer is implicitly assuming that the memory pointed to by a and b don't overlap.

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.

C99 to the Rescue

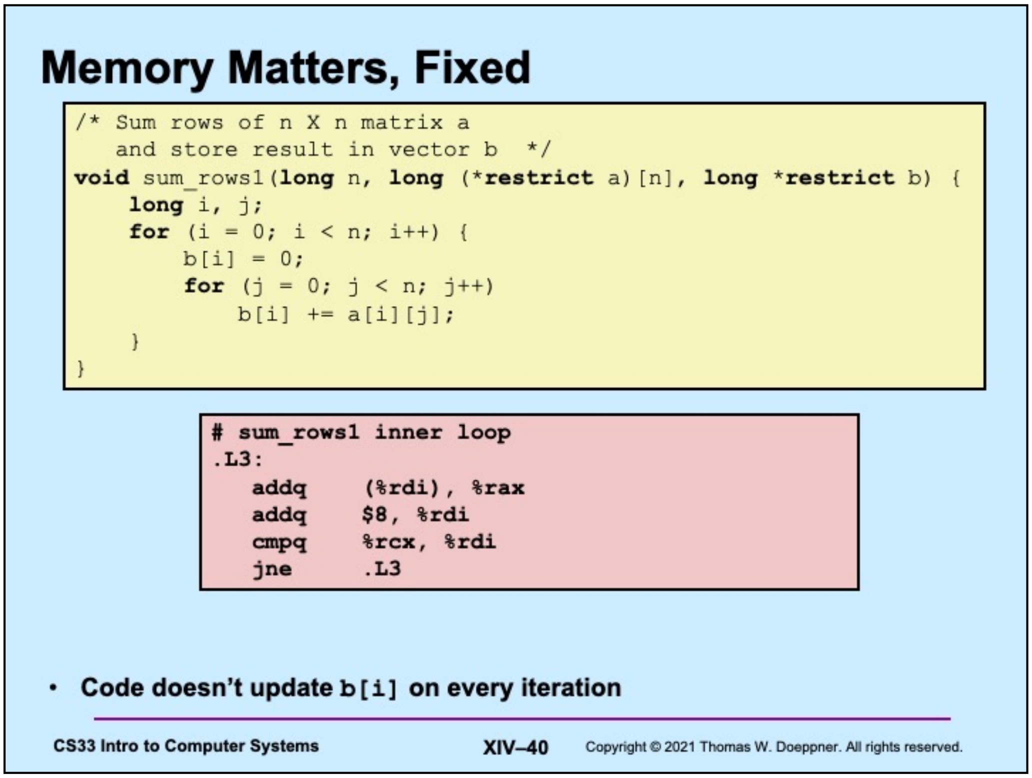
- **New attribute**
  - **restrict**
    - » applied to a pointer, tells the compiler that the object pointed to will be accessed only via this pointer
    - » compiler thus doesn't have to worry about aliasing
    - » but the programmer does ...
    - » syntax

```
int *restrict pointer;
```

Pointers and Arrays

- **long** a[][n]
  - a is a 2-D array of longs, the size of each row is n
- **long** (*b)[n]
  - b is a pointer to a 1-D array of size n

- a and b are of the same type



Note: we must give gcc the flag "-std=gnu99" for this to be compiled.

Observe that

```
long (*a)[n]
```

declares a to be a pointer to an array of n longs.

Thus

```
long (*restrict a)[n]
```

declares a to be a restricted pointer to an array of n longs

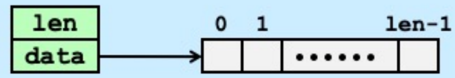
Exploiting Instruction-Level Parallelism

- **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: Datatype for Vectors

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




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

/* return length of vector */
int vec_length(vec_ptr_t v) {
     return v->len;
}
```


Supplied by CMU.

Benchmark Computation

```
void combine1(vec_ptr_t 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;
    }
}
```

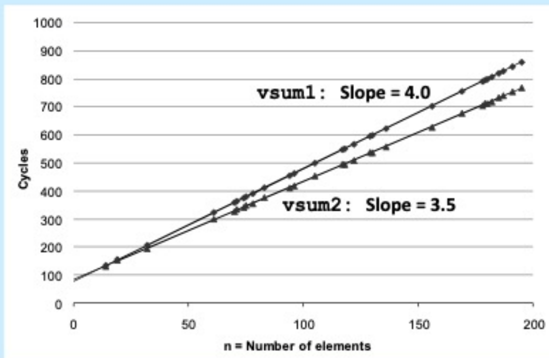
Compute sum or product of vector elements

- **Data Types**
  - use different declarations for `data_t`
    - » `int`
    - » `float`
    - » `double`
- **Operations**
  - use different definitions of `OP` and `IDENT`
    - » `+, 0`
    - » `*, 1`

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_t 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.

The times given in the table are in cycles/element.

Move vec_length

```
void combine2(vec_ptr_t 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 Function Calls

```
void combine3(vec_ptr_t 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];
    }
}
```

```
data_t *get_vec_start(
    vec_ptr v) {
  return v->data;
}
```

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_t 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.

Quiz 2

Combine4 is pretty fast; we've done all the "obvious" optimizations. How much faster will we be able to make it? (Hint: it involves taking advantage of pipelining and multiple functional units on the chip.)

a) 1× (it's already as fast as possible)
b) 2× – 4×
c) 16× – 64×