Lecture 9:
Memory Layout, Worms,
Program Optimization

Computer Architecture and
Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012

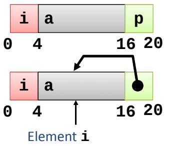
Last time: structures

- C Code

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```

What does it do?

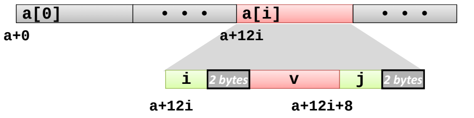


```
 # %edx = r
 movl (%edx),%ecx        # r->i
 leal 0(,%ecx,4),%eax    # 4*(r->i)
 leal 4(%edx,%eax),%eax  # r+4+4*(r->i)
 movl %eax,16(%edx)      # Update r->p
```

Last time : alignment

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8
  - Resolved during linking

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

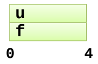


```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

Last time: unions

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```



```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

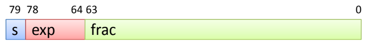
```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(float) u` ?

Same as `(unsigned) f` ?

Last time: FPU stack (x87)

- FPU register format (80 bit extended precision)



- FPU registers
  - 8 registers `%st(0)` - `%st(7)`
  - Logically form stack
  - Top: `%st(0)`
  - Bottom disappears (drops out) after too many pushes

Last time: SSE3 registers

- All caller saved
- `%xmm0` for floating point return value

%xmm0	Argument #1	%xmm8
%xmm1	Argument #2	%xmm9
%xmm2	Argument #3	%xmm10
%xmm3	Argument #4	%xmm11
%xmm4	Argument #5	%xmm12
%xmm5	Argument #6	%xmm13
%xmm6	Argument #7	%xmm14
%xmm7	Argument #8	%xmm15

Summary

- Arrays in C
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - No bounds checking
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Overlay declarations
  - Way to circumvent type system
- Floating point
  - x87: stack machine
  - SSE3: 16 FP registers, mirrors integer architecture

Today

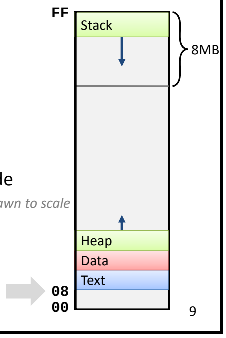
- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls

IA32 Linux memory layout

- Stack
  - Runtime stack (8MB limit)
- Heap
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only

not drawn to scale

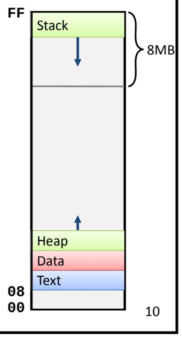
Upper 2 hex digits
= 8 bits of address



Memory allocation example

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?

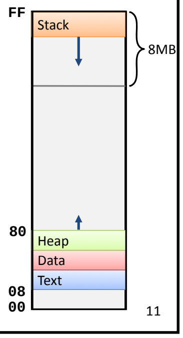


IA32 example addresses

address range ~2^{32}

$esp	`0xffffbcd0`
p3	`0x65586008`
p1	`0x55585008`
p4	`0x1904a110`
p2	`0x1904a008`
&p2	`0x18049760`
beyond	`0x08049744`
big_array	`0x18049780`
huge_array	`0x08049760`
main()	`0x080483c6`
useless()	`0x08049744`
final **malloc()**	`0x006be166`

malloc() is dynamically linked
address determined at runtime

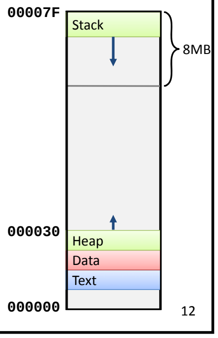


x86-64 example addresses

address range ~2^{47}

$rsp	`0x7ffffff8d1f8`
p3	`0x2aaabaadd010`
p1	`0x2aaaaaadc010`
p4	`0x000011501120`
p2	`0x000011501010`
&p2	`0x000010500a60`
beyond	`0x000000500a44`
big_array	`0x000010500a80`
huge_array	`0x000000500a50`
main()	`0x000000400510`
useless()	`0x000000400500`
final **malloc()**	`0x00386ae6a170`

malloc() is dynamically linked
address determined at runtime



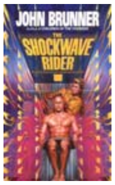
Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls

Worms and Viruses

- Worm: A program that
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- Virus: Code that
  - Add itself to other programs
  - Cannot run independently
- Both are (usually) designed to spread among computers and to wreak havoc

Early worms



- Term coined in 1975 by John Brunner
  - First "cyberpunk" novel: The Shockwave Rider
- Mid-1970s: research into benign worms at BBN and Xerox PARC
  - Network of Alto machines at PARC
  - Shoch, J. F. and Hupp, J. A. 1982. The "worm" programs—early experience with a distributed computation. Commun. ACM 25, 3 (Mar. 1982), 172-180.
- November 1988: Robert Morris' Worm
  - First Internet worm, attacked thousands of hosts
  - Morris now professor of Computer Science at MIT
  - Awarded the SIGOPS Mark Weiser award recently

… and the rest is history.

String library code

- Implementation of Unix function **`gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - No way to specify limit on number of characters to read
- Similar problems with other Unix functions
  - **`strcpy`**: Copies string of arbitrary length
  - **`scanf`**, **`fscanf`**, **`sscanf`**, when given **`%s`** conversion specification

Vulnerable buffer code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

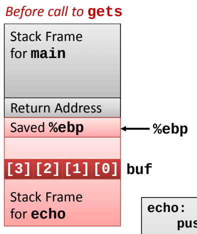
```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

Buffer overflow disassembly

```
080484f0 <echo>:
 80484f0:  55                push   %ebp
 80484f1:  89 e5             mov    %esp,%ebp
 80484f3:  53                push   %ebx
 80484f4:  8d 5d f8          lea    0xfffffff8(%ebp),%ebx
 80484f7:  83 ec 14          sub    $0x14,%esp
 80484fa:  89 1c 24          mov    %ebx,(%esp)
 80484fd:  e8 ae ff ff ff    call   80484b0 <gets>
 8048502:  89 1c 24          mov    %ebx,(%esp)
 8048505:  e8 8a fe ff ff    call   8048394 <puts@plt>
 804850a:  83 c4 14          add    $0x14,%esp
 804850d:  5b                pop    %ebx
 804850e:  c9                leave
 804850f:  c3                ret
```

```
 80485f2:  e8 f9 fe ff ff    call   80484f0 <echo>
 80485f7:  8b 5d fc          mov 0xfffffffc(%ebp),%ebx
 80485fa:  c9                leave
 80485fb:  31 c0             xor    %eax,%eax
 80485fd:  c3                ret
```

Buffer overflow stack

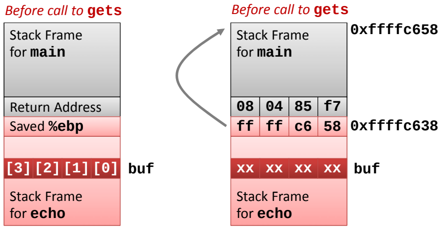


```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp                # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx                # Save %ebx
    leal  -8(%ebp),%ebx       # Compute buf as %ebp-8
    subl  $20, %esp           # Allocate stack space
    movl  %ebx, (%esp)        # Push buf on stack
    call  gets                # Call gets
    . . .
```

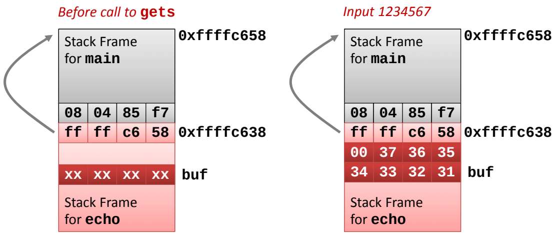
Buffer overflow stack example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffc658
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f7
```



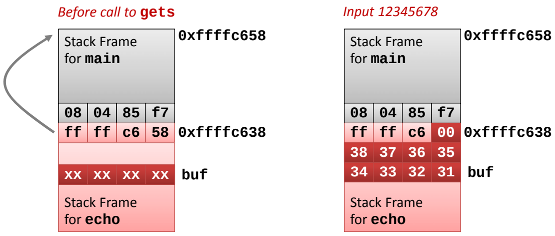
```
80485f2:call 80484f0 <echo>
80485f7:mov  0xfffffffc(%ebp),%ebx # Return Point
```

Buffer overflow example #1



Overflow buf, but no problem

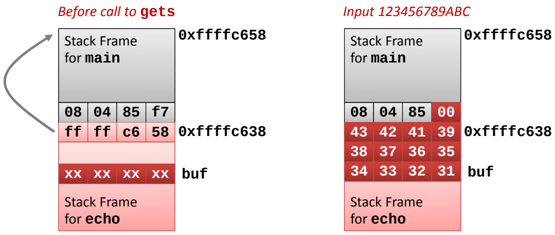
Buffer overflow example #2



Base pointer corrupted

```
. . .
804850a: 83 c4 14  add    $0x14,%esp  # deallocate space
804850d: 5b        pop    %ebx        # restore %ebx
804850e: c9        leave              # movl %ebp, %esp; popl %ebp
804850f: c3        ret                # Return
```

Buffer overflow example #3



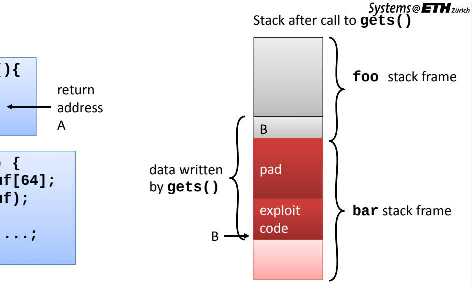
Return address corrupted

```
80485f2:  call 80484f0 <echo>
80485f7:  mov  0xfffffffc(%ebp),%ebx # Return Point
```

Malicious use of buffer overflow

```
void foo(){
  bar();
  ...
}
```

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When **bar()** executes **ret**, will jump to exploit code

Exploits based on buffer overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- Internet worm (one vector)
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger droh@cs.cmu.edu`**
  - Worm attacked fingerd server by sending phony argument:
    - **`finger “exploit-code padding new-return-address”`**
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Avoiding overflow vulnerability

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- Use library routines that limit string lengths
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don’t use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable integer

System-level protections

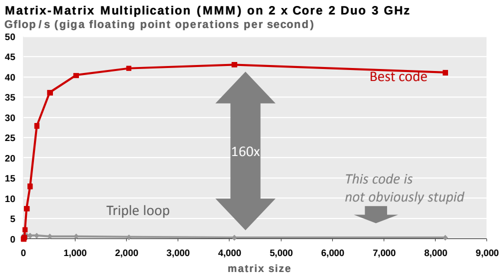
- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code
- Nonexecutable code segments
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
  - Add explicit “execute” permission

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

Today

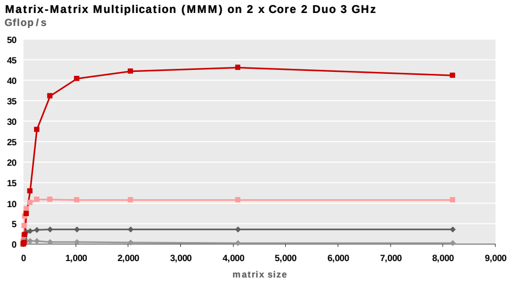
- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - **Overview**
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls

Example matrix multiplication



- Standard desktop computer, compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- *What is going on?*

MMM plot: analysis



- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- ***Effect: more instruction level parallelism, better register use, less L1/L2 cache misses, less TLB misses***

Harsh reality

- *There's more to runtime performance than asymptotic complexity*
- *One can easily loose 10x, 100x in runtime or even more*
- What matters:
  - Constants (100n and 5n is both O(n), but ….)
  - Coding style (unnecessary proc. calls, unrolling, reordering, …)
  - Algorithm structure (locality, instruction level parallelism, …)
  - Data representation (complicated structs or simple arrays)

Harsh reality

- Must optimize at multiple levels:
  - Algorithm
  - Data representations
  - Procedures
  - Loops
- Must understand system to optimize performance
  - How programs are compiled and executed
    - Execution units, memory hierarchy
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

---

Optimizing compilers

-O

- Use optimization flags,
  default can be no optimization (-O0)!
- Good choices for gcc: -O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers
  - icc is usually faster than gcc

Example

```
double a[4][4];
double b[4][4];
double c[4][4]; # set to zero

/* Multiply 4 x 4 matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < 4; i++)
         for (j = 0; j < 4; j++)
              for (k = 0; k < 4; k++)
                   c[i*4+j] += a[i*4 + k]*b[k*4 + j];
}
```

- Compiled without flags:
  ~1300 cycles
- Compiled with –O3 –m64 -march=… –fno-tree-vectorize
  ~150 cycles
- Core 2 Duo, 2.66 GHz

Optimizing compilers

- Compilers are good at: mapping program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Compilers are not good at: improving 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
- Compilers are not good at: overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

Limitations of optimizing compilers

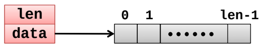
- *If in doubt, the compiler is conservative*
- Operate under fundamental constraints
  - Must not change program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- Behavior that may be obvious to the 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

Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - – Overview
  - – **Removing unnecessary procedure calls**
  - – Code motion/precomputation
  - – Strength reduction
  - – Sharing of common subexpressions
  - – Optimization blocker: Procedure calls
  - – Optimization blocker: Memory aliasing

Example: Data type for vectors

```
/* data structure for vectors */
typedef struct{
    int len;
    double *data;
} vec;
```



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

Example: Summing vector elements

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

Bound check unnecessary in sum_elements

Why?

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = vec_length(v);
  *res = 0.0;
  double val;

  for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
    *res += val;
  }
  return *res;
}
```

Overhead for every fp +:

- One fct call
- One <
- One >=
- One ||
- One memory variable access

Slowdown:
probably 10x or more

Removing procedure call

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = vec_length(v);
  *res = 0.0;
  double val;

  for (i = 0; i < n; i++) {
    get_vec_element(v, i, &val);
    *res += val;
  }
  return res;
}
```

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
  int i;
  n = vec_length(v);
  *res = 0.0;
  double *data = get_vec_start(v);

  for (i = 0; i < n; i++)
    *res += data[i];
  return *res;
}
```

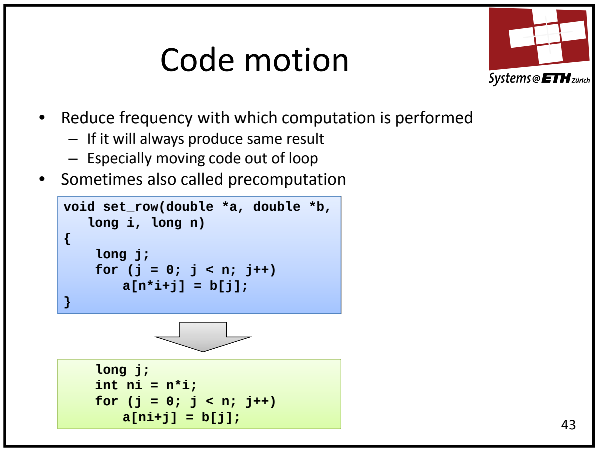
Removing procedure calls

- Procedure calls can be very expensive
- Bound checking can be very expensive
- Abstract data types can easily lead to inefficiencies
  - – Usually avoided for in superfast numerical library functions
- Watch your innermost loop!
- Get a feel for overhead versus actual computation being performed

---

Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - – Overview
  - – Removing unnecessary procedure calls
  - – **Code motion/precomputation**
  - – Strength reduction
  - – Sharing of common subexpressions
  - – Optimization blocker: Procedure calls
  - – Optimization blocker: Memory aliasing

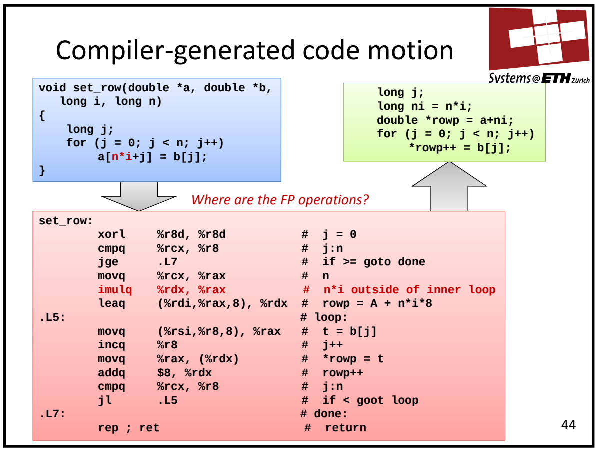


Code motion

- Reduce frequency with which computation is performed
  - If it will always produce same result
  - Especially moving code out of loop
- Sometimes also called precomputation

```
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];
```



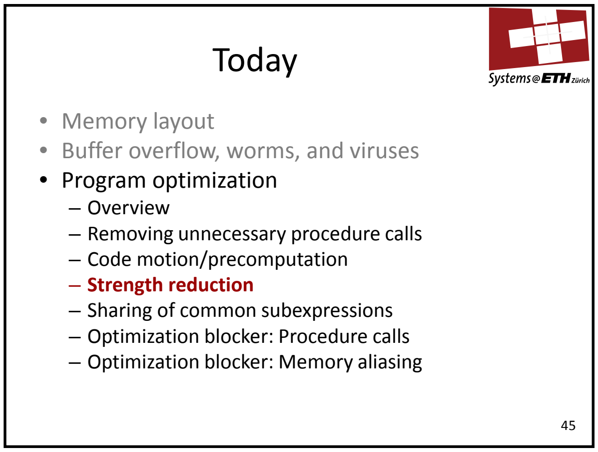
Compiler-generated code motion

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

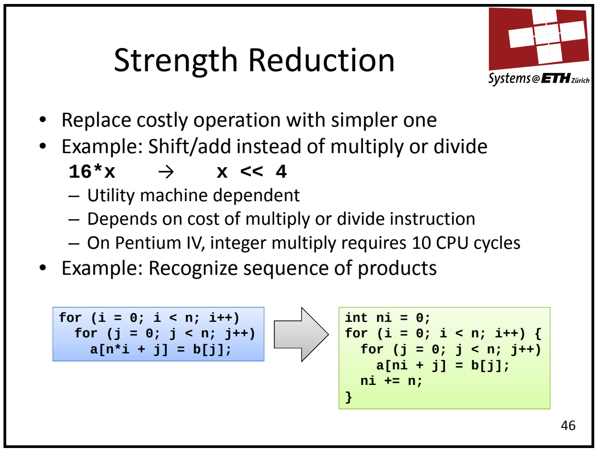
Where are the FP operations?

```
set_row:
        xorl    %r8d, %r8d          #  j = 0
        cmpq    %rcx, %r8           #  j:n
        jge     .L7                 #  if >= goto done
        movq    %rcx, %rax          #  n
        imulq   %rdx, %rax          #  n*i outside of inner loop
        leaq    (%rdi,%rax,8), %rdx #  rowp = A + n*i*8
.L5:                                # loop:
        movq    (%rsi,%r8,8), %rax  #  t = b[j]
        incq    %r8                 #  j++
        movq    %rax, (%rdx)        #  *rowp = t
        addq    $8, %rdx            #  rowp++
        cmpq    %rcx, %r8           #  j:n
        jl      .L5                 #  if < goot loop
.L7:                                # done:
        rep ; ret                   #  return
```



Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - **Strength reduction**
  - Sharing of common subexpressions
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing

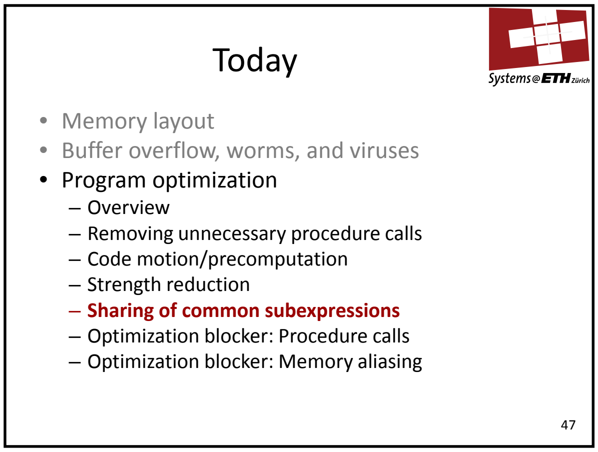


Strength Reduction

- Replace costly operation with simpler one
- Example: Shift/add instead of multiply or divide
  `16*x    →    x << 4`
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium IV, integer multiply requires 10 CPU cycles
- Example: Recognize sequence of products

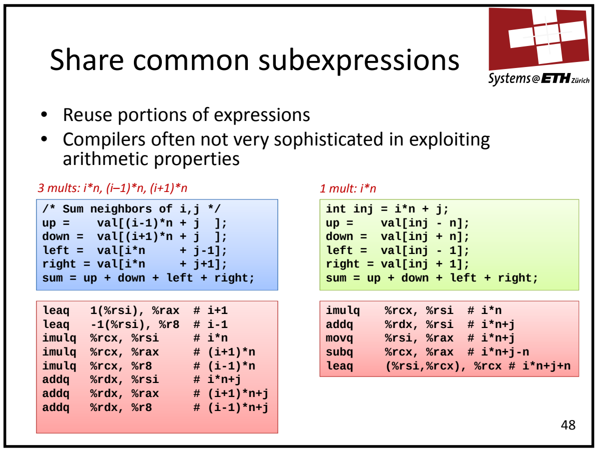
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

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



Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - **Sharing of common subexpressions**
  - Optimization blocker: Procedure calls
  - Optimization blocker: Memory aliasing



Share common subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

*3 mults: i*n, (i–1)*n, (i+1)*n*

```
/* 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;
```

```
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
```

*1 mult: i*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;
```

```
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
```

Today

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Overview
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - **Optimization blocker: Procedure calls**
  - Optimization blocker: Memory aliasing

Optimization blocker #1: procedure calls

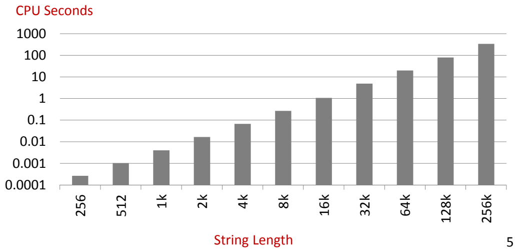
- Procedure to convert string to lower case

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

Extracted from CMU lab submissions, Fall 1998

Performance

- Time quadruples when string length doubles
- Quadratic performance



Why is that?

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

- String length is called in every iteration!
  - And `strlen` is O(n), so `lower` is O(n^2)

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

Improving performance

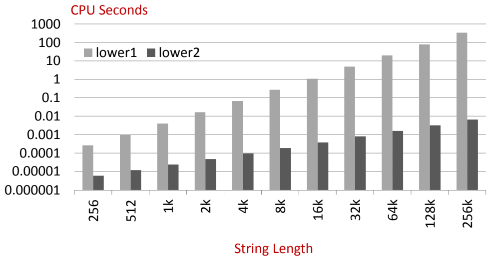
```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
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/precomputation

Performance

- Lower2: Time doubles when double string length
- Linear performance



Optimization blocker: Procedure calls

- Why couldn't compiler move `strlen` out of inner loop?
  - Procedure may have side effects
  - Function may not return same value for given arguments
    - Could depend on other parts of global state
    - Procedure `lower` could interact with `strlen`
- Compiler usually treats procedure call as a black box that cannot be analyzed
  - Consequence: conservative in optimizations
- Remedies:
  - Inline the function if possible
  - 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;
}
```

Moral: collaborate with the compiler!

- Turn on optimization!
- Let the compiler do what it's good at.
- Remove obstacles to optimizer
- Do it yourself if necessary
- Sometimes at odds with abstraction and encapsulation
  - That's the tradeoff…

Next time: Advanced C

- Operators
- Function pointers
- Typedefs and structures
- `goto`
- Assertions
- Are arrays the same as pointers?
- `setjmp()/longjmp()`
- Coroutines