Code Optimization I : Machine-Independent Optimizations

COMP400727: Introduction to Computer Systems

Hao Li
Xi'an Jiaotong University

Back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums
and vacuum tubes,
Real Programmers wrote in machine code.

Not FORTRAN. Not RATFOR. Not, even,
assembly language.

Machine Code.

Raw, unadorned, inscrutable hexadecimal numbers. Directly.

— "The Story of Mel, a Real Programmer"
Ed Nather, 1983

很久以前，我还是个电脑白吃

很久以前，那还是我用win98的时候我系统崩溃了，因为我是电脑白吃，我朋友给我介绍了一个高手来帮我修电脑。

他看了一下电脑，问我有没有98的盘，我说没有。他想了一下，叫我把固定电话拿给他，我想修电脑要电话干什么，但人家是高手，我也不好说什么，就把电话拔下来给他了。

他把电话线空着的一头接在电脑的一个插孔内，然后进入了dos，然后就开始在电话上不停的按着键，他按键的速度非常快，但是只按0，1两个键，我搞不懂这有什么用，但也不敢问，看了半个多小时，他还是不停的按这两个键，我渐渐的有些困，我问他这东西要搞多久，他说要几个小时，我给他倒了杯茶，就一个人去隔壁睡觉了。

醒来的时候，一看已经过了4个多小时，我起身到隔壁，看见他正在98里面调试，过了一会儿，他说，你试试，我坐上椅子用了一下，真的好了，我当时也不懂电脑，谢过人家就走了。

后来我慢慢对电脑有了了解，终于了解，原来当时那位高手是用机器语言编了一个98系统，我后来问我朋友那位高手的下落，我朋友说前几年去了美国之后，杳无音讯....

很久以前，我还是个电脑白吃

很久以前，那还是我用winXP的时候有次我系统崩溃了，因为我是电脑白吃，我朋友给我介绍了一个高手来帮我修电脑。

他看了一下电脑，问我有没有XP的安装盘，我说没有。

他想了一下，叫我把一张空的DVD刻录盘和一根针拿给他，我想修电脑要刻录盘和针干什么，但人家是高手，我也不好说什么，就把DVD刻录盘拿一张来给他了。

他把针头对着刻录盘戳，他戳的速度非常快，但是只戳深或浅，我搞不懂这有什么用，但也不敢问，看了半个多小时，他还是不停的戳着DVD刻录盘，我渐渐的有些困，我问他这东西要搞多久，他说要几个小时，我给他倒了杯茶，就一个人去隔壁睡觉了。

醒来的时候，一看已经过了4个多小时，我起身到隔壁，看见他正在Xp里面调试，还装上了office、photoshop、迅雷、魔兽世界等软件……过了一会儿，他说，你试试，我坐上椅子用了一下，真的好了，我当时也不懂电脑，谢过人家就走了。

后来我慢慢对电脑有了了解，终于了解，原来当时那位高手是针头刻了一个单面双层的DVD，在里面刻上了Xp、office、photoshop、魔兽世界的安装程序，我后来问我朋友那位高手的下落，我朋友说前几年去了美国之后，杳无音讯….

很久以前，我还是个电脑白吃

很久以前，那还是我用winXP的时候有次我不小心把D盘格了，因为我是电脑白吃，我朋友给我介绍了一个高手来帮我恢复数据。

他看了一下电脑，问我有没有备份过Ghost，我说没有。

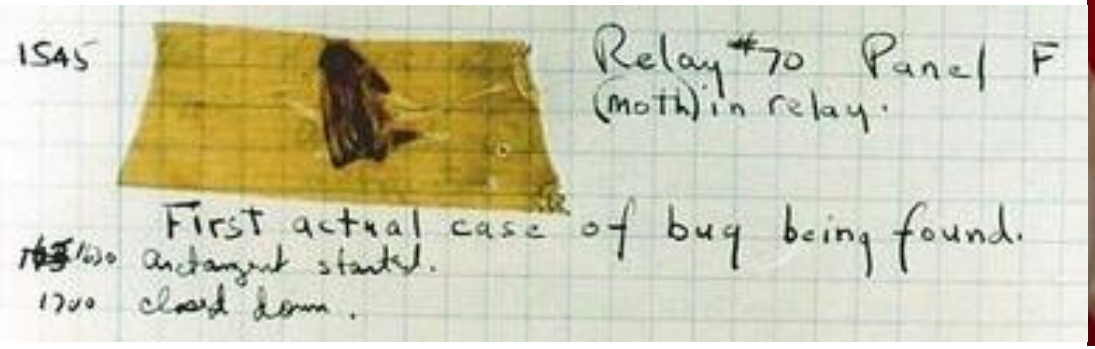
他想了一下，叫我把一块磁铁拿给他，还问我D盘里有什么东西，我想修电脑要磁铁干什么，但人家是高手，我也不好说什么，就把磁铁给他了，还告诉他我D盘里全是火影忍者的动画。

他把硬盘拆了，用磁铁在上面画圈，他画圈的速度非常快，但是只有碰到或不碰到，我搞不懂这有什么用，但也不敢问，看了半个多小时，他还是不停的在硬盘上画着圈，我渐渐的有些困，我问他这东西要搞多久，他说要几个小时，我给他倒了杯茶，就一个人去隔壁睡觉了。

醒来的时候，一看已经过了4个多小时，我起身到隔壁，看见他正在D盘里面调试，里面全是火影忍者的动画片，过了一会儿，他说，你试试，我坐上椅子用了一下，真的好了，我当时也不懂电脑，谢过人家就走了。

后来我慢慢对电脑有了了解，终于了解，原来当时那位高手是用磁铁直接在硬盘上写数据，凭着惊人的记忆力将火影忍者的动画片都写入了硬盘，我后来问我朋友那位高手的下落，我朋友说前几年去了美国之后，杳无音讯....

- **Rear Admiral Grace Hopper (1906-1992)**
  - Invented first compiler in 1951 (technically it was a linker)
  - Coined "compiler" (and "bug")
  - Compiled for Harvard Mark I
  - Eventually led to COBOL (which ran the world for years)
  - "I decided data processors ought to be able to write their programs in English, and the computers would translate them into machine code"



- **John Backus (1924-2007)**
  - Led team at IBM invented the first commercially available compiler in 1957
  - Compiled FORTRAN code for the IBM 704 computer
  - FORTRAN still in use today for high performance code
  - "Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, I started work on a programming system to make it easier to write programs"

- **Fran Allen (1932-2020)**
  - Pioneer of many optimizing compilation techniques
  - Wrote a paper simply called "Program Optimization" in 1966
  - "This paper introduced the use of graph-theoretic structures to encode program content in order to automatically and efficiently derive relationships and identify opportunities for optimization"
  - First woman to win the ACM Turing Award (the "Nobel Prize of Computer Science"), in 2006

Today

- **Principle and Goals**
- **Generally Useful Optimizations**
  - Constant folding
  - Strength reduction
  - Sharing of common subexpressions
  - Code motion/precomputation
- **Example: Bubblesort**
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing

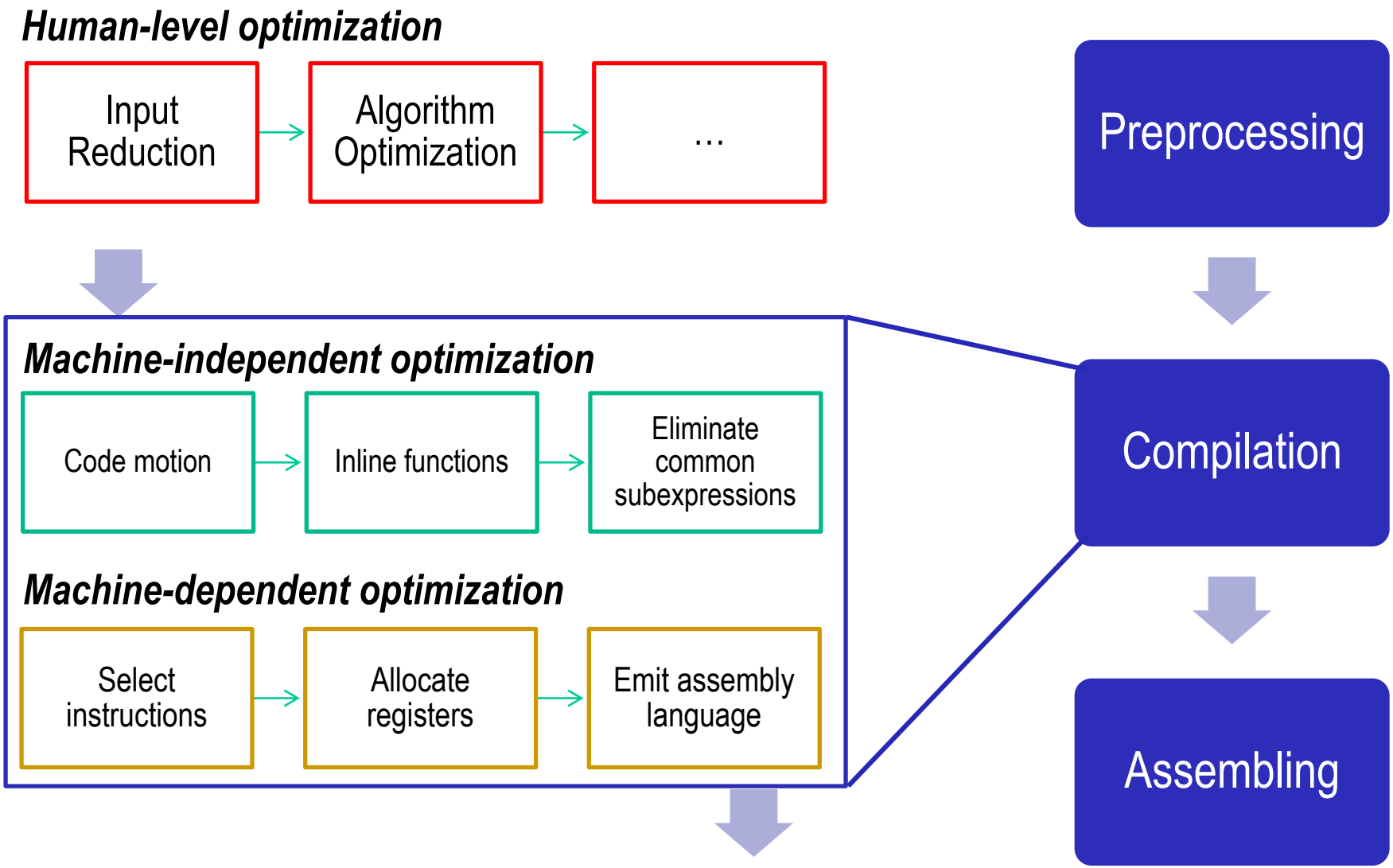
Goals of compiler optimization

- **Minimize number of instructions**
  - Don't do calculations more than once
  - Don't do unnecessary calculations at all
  - Avoid slow instructions (multiplication, division)
- **Avoid waiting for memory**
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns
  - Load data from memory early, and only once
- **Avoid branching**
  - Don't make unnecessary decisions at all
  - Make it easier for the CPU to predict branch destinations
  - "Unroll" loops to spread cost of branches over more instructions

Limits to compiler optimization

- **Generally cannot improve algorithmic complexity**
  - Only constant factors, but those can be worth 10x or more…
- **Must not cause *any* change in program behavior**
  - Programmer may not care about "edge case" behavior, but compiler does not know that
  - Exception: language may declare some changes acceptable
- **Usually only analyze one function at a time**
  - Whole-program analysis is usually too expensive
  - Exception: *inlining* merges many functions into one
- **Cannot anticipate run-time inputs**
  - "Worst case" performance can be just as important as "normal"
  - Especially for code exposed to *malicious* input (e.g. network servers)

Multiple Levels of Optimizations



Today

- Principle and Goals
- **Generally Useful Optimizations**
  - Constant folding
  - Strength reduction
  - Sharing of common subexpressions
  - Code motion/precomputation
- Example: Bubblesort
- Optimization Blockers
  - Procedure calls
  - Memory aliasing

Next several slides done live…

- https://godbolt.org/z/Es5s8qsvj

- Go to Godbolt (the compiler explorer) to play around with C and the resulting assembly generated under different compiler optimizations (change the flag from –O3 to –Og, etc. to see more or less aggressive optimization).

Constant folding

- **Do arithmetic in the compiler**

```
long mask = 0xFF << 8;      →
long mask = 0xFF00;
```

- **Any expression with constant inputs can be folded**
- **Might even be able to remove library calls…**

```
size_t namelen = strlen("Harry Bovik");    →
size_t namelen = 11;
```

Dead code elimination

- **Don't emit code that will never be executed**

```
if (0) { puts("Kilroy was here"); }
if (1) { puts("Only bozos on this bus"); }
```

- **Don't emit code whose result is overwritten**

```
x = 23;
x = 42;
```

- **These may look silly, but...**
  - Can be produced by other optimizations
  - Assignments to x might be far apart

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* 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;
```

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* 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`

```
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;
```

1 multiplication: `i*n`

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* 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
...
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size; i-cache)

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Always true **Does nothing** **Can constant fold**

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

```
int func(int y) {
  int tmp = 0;
  if (y != 0) tmp = y - 1;

  if (y != -1) tmp += y;
  return tmp;
}
```

Puzzles - 1

before

```
t1 = x * z;
t2 = a + b;
t3 = p & t2;
t4 = x * z;
t5 = a - z;
```

after

```
t1 = x * z;
t2 = a + b;
t3 = p % t2;
t5 = a - z;
```

```
int x = 5 + 7 + c;
```

```
int x = 12 + c;
```

```
int b = 5;
int c = b * 2;
int z = a[c];
```

```
int z = a[10];
```

Puzzles - 2

before

```
int x = a + 23;
z = a + y;
printf("%d,%d", z, y);
```

```
a = b;
z = a + x;
x = z - b;
```

after

```
z = a + y;
printf("%d,%d", z, y);
```

```
z = a + x
```

Puzzles - 3

before

```
p = 100;
for(i = 0; i < p; i++)
{
  a = b + 40;
  if (p/a == 0)
    printf("%d",p);
}
```

after

```
p = 100;
a = b + 40;
for(i = 0; i < p; i++)
{
  if (p/a == 0)
    printf("%d",p);
}
```

```
while((x+y)>n)
{
     ……;
}
```

```
int t = x + y;
while(t > n)
{
     ……;
}
```

Puzzles - 4

before

```
for(i=0;i<5;i++)
{
  for(j=0;j<5;j++)
  {
    x[i][j]=0;
  }
}

for(i=0;i<5;i++)
{
  x[i][i]=1;
}
```

after

```
for(i=0;i<5;i++)
{
  for(j=0;j<5;j++)
  {
    x[i][j]=0;
  }
  x[i][i]=1;
}
```

Today

- **Principle and Goals**
- **Generally Useful Optimizations**
  - Constant folding
  - Strength reduction
  - Sharing of common subexpressions
  - Code motion/precomputation
- **Example: Bubblesort**
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing

Optimization Example: Bubblesort

- **Bubblesort program that sorts an array `A` that is allocated in static storage:**
  - an element of `A` requires four bytes
  - elements of `A` are numbered 1 through `n` (`n` is a variable)
  - `A[j]` is in location `&A+4*(j-1)`

```
for (i = n-1; i >= 1; i--) {
  for (j = 1; j <= i; j++)
    if (A[j] > A[j+1]) {
      temp = A[j];
      A[j] = A[j+1];
      A[j+1] = temp;
    }
}
```

Translated (Pseudo) Code

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]    // A[j]
     t4 := j+1
     t5 := t4-1
     t6 := 4*t5
     t7 := A[t6]    // A[j+1]
     if t3<=t7 goto L3
```

```
for (i = n-1; i >= 1; i--) {
  for (j = 1; j <= i; j++)
    if (A[j] > A[j+1]) {
      temp = A[j];
      A[j] = A[j+1];
      A[j+1] = temp;
    }
}
```

```
     t8 := j-1
     t9 := 4*t8
     temp := A[t9]   // temp:=A[j]
     t10 := j+1
     t11:= t10-1
     t12 := 4*t11
     t13 := A[t12]   // A[j+1]
     t14 := j-1
     t15 := 4*t14
     A[t15] := t13   // A[j]:=A[j+1]
     t16 := j+1
     t17 := t16-1
     t18 := 4*t17
     A[t18]:=temp    // A[j+1]:=temp
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Instructions

29 in outer loop

25 in inner loop

Redundancy in Address Calculation

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]     // A[j]
     t4 := j+1
     t5 := t4-1
     t6 := 4*t5
     t7 := A[t6]     // A[j+1]
     if t3<=t7 goto L3
```

```
     t8 :=j-1
     t9 := 4*t8
     temp := A[t9]     // temp:=A[j]
     t10 := j+1
     t11:= t10-1
     t12 := 4*t11
     t13 := A[t12]     // A[j+1]
     t14 := j-1
     t15 := 4*t14
     A[t15] := t13     // A[j]:=A[j+1]
     t16 := j+1
     t17 := t16-1
     t18 := 4*t17
     A[t18]:=temp      // A[j+1]:=temp
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Redundancy Removed

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]      // A[j]
     t6 := 4*j
     t7 := A[t6]      // A[j+1]
     if t3<=t7 goto L3
     t8 :=j-1
     t9 := 4*t8
     temp := A[t9]    // temp:=A[j]
     t12 := 4*j
     t13 := A[t12]    // A[j+1]
     A[t9]:= t13      // A[j]:=A[j+1]
     A[t12]:=temp     // A[j+1]:=temp
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Instructions
20 in outer loop
16 in inner loop

More Redundancy

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]      // A[j]
     t6 := 4*j
     t7 := A[t6]      // A[j+1]
     if t3<=t7 goto L3
```

```
     t8 :=j-1
     t9 := 4*t8
     temp := A[t9]    // temp:=A[j]
     t12 := 4*j
     t13 := A[t12]    // A[j+1]
     A[t9]:= t13      // A[j]:=A[j+1]
     A[t12]:=temp     // A[j+1]:=temp
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Redundancy Removed

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]     // old_A[j]
     t6 := 4*j
     t7 := A[t6]     // A[j+1]
     if t3<=t7 goto L3
```

```
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

```
A[t2] := t7    // A[j]:=A[j+1]
A[t6] := t3    // A[j+1]:=old_A[j]
```

Instructions
15 in outer loop
11 in inner loop

Redundancy in Loops

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]     // A[j]
     t6 := 4*j
     t7 := A[t6]     // A[j+1]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Redundancy Eliminated

```
     i := n-1
L5:  if i<1 goto L1
     j := 1
L4:  if j>i goto L2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]      // A[j]
     t6 := 4*j
     t7 := A[t6]      // A[j+1]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  j := j+1
     goto L4
L2:  i := i-1
     goto L5
L1:
```

```
     i := n-1
L5:  if i<1 goto L1
     t2 := 0
     t6 := 4
     t19 := 4*i
L4:  if t6>t19 goto L2
     t3 := A[t2]
     t7 := A[t6]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  t2 := t2+4
     t6 := t6+4
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Final Pseudo Code (after strength reduction)

```
     i := n-1
L5:  if i<1 goto L1
     t2 := 0
     t6 := 4
     t19 := i << 2
L4:  if t6>t19 goto L2
     t3 := A[t2]
     t7 := A[t6]
     if t3<=t7 goto L3
     A[t2] := t7
     A[t6] := t3
L3:  t2 := t2+4
     t6 := t6+4
     goto L4
L2:  i := i-1
     goto L5
L1:
```

Instructions
Before Optimizations
29 in outer loop
25 in inner loop

Instructions
After Optimizations
15 in outer loop
9 in inner loop

- These were **Machine-Independent Optimizations**.
- Will be followed by **Machine-Dependent Optimizations**, including allocating temporaries to registers, converting to assembly code

Today

- **Principle and Goals**
- **Generally Useful Optimizations**
  - Constant folding
  - Strength reduction
  - Sharing of common subexpressions
  - Code motion/precomputation
- **Example: Bubblesort**
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing

Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
  - Often prevents optimizations that affect only "edge case" behavior
- **Behavior obvious to the programmer is not obvious to compiler**
  - e.g., Data range may be more limited than types suggest (short vs. int)
- **Most analysis is only within a procedure**
  - Whole-program analysis is usually too expensive
  - Sometimes compiler does interprocedural analysis **within** a file (new GCC)
- **Most analysis is based only on *static* information**
  - Compiler has difficulty anticipating run-time inputs
- **When in doubt, the compiler must be conservative**

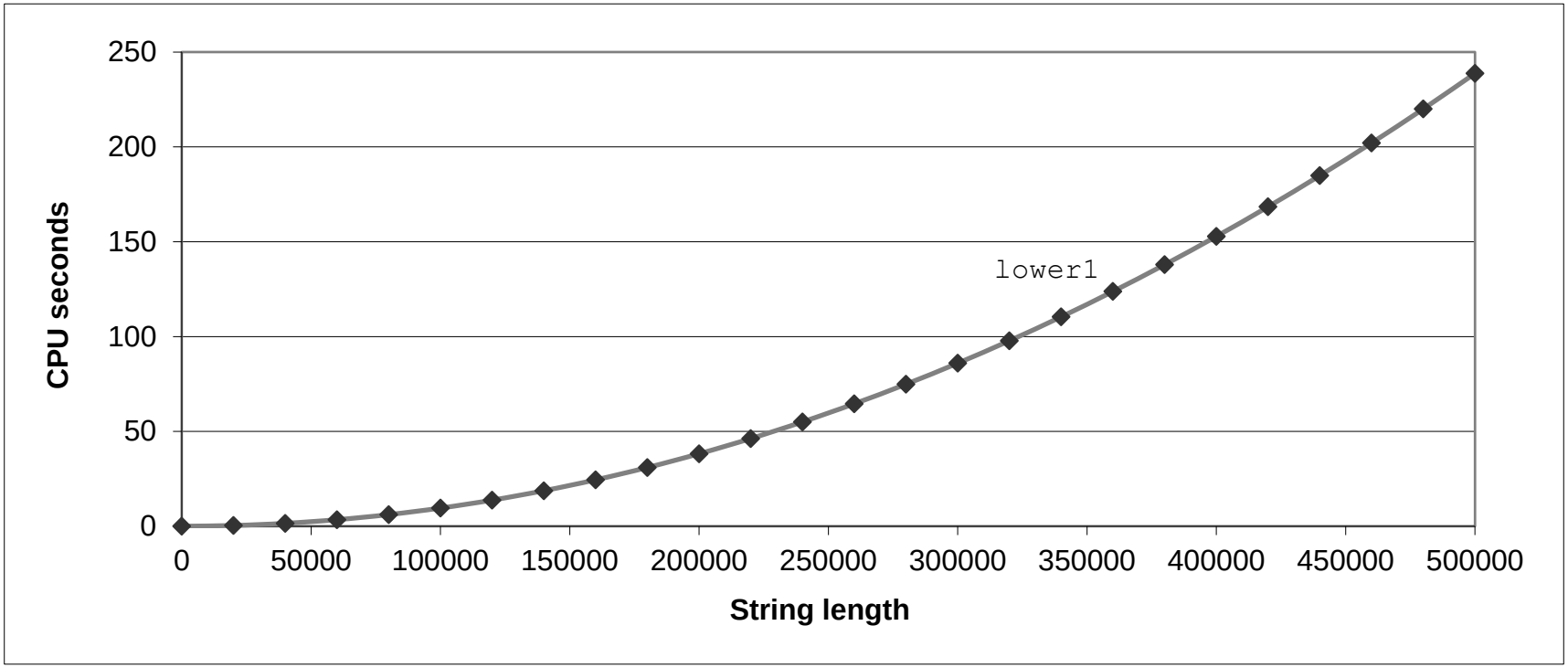
Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

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

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Convert Loop To Goto Form

```
void lower(char *s)
{
   size_t 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

Calling Strlen

```
/* My version of 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
  - Require times N, N-1, N-2, …, 1
  - Overall $O(N^2)$ performance

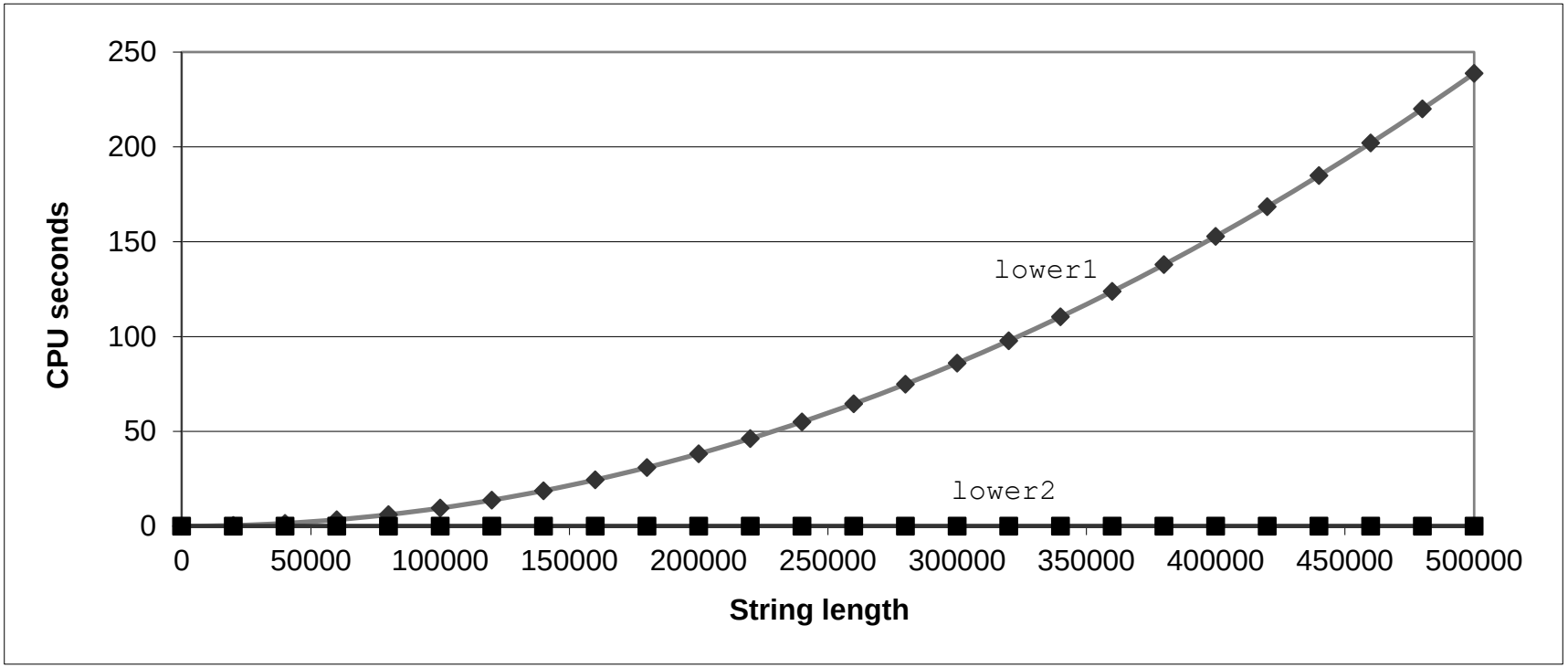
Improving Performance

```
void lower(char *s)
{
  size_t i;
  size_t 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
- Legal 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 couldn't compiler move `strlen` out of 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`**
- **Warning:**
  - Compiler may treat procedure call as a black box
  - Weak optimizations near them
- **Remedies:**
  - Use of inline functions
    - GCC does this with –O1
      - Within single file
  - Do your own code motion

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

Memory Matters

```
/* Sum rows of n X n matrix a
   and store 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
.L4:
        movsd   (%rsi,%rax,8), %xmm0        # FP load
        addsd   (%rdi), %xmm0               # FP add
        movsd   %xmm0, (%rsi,%rax,8)        # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

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

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store 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);
```

```
double A[9] =
  { 0,   1,   2,
    3,  22, 224},
   32,  64, 128};
```

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

Removing Aliasing

```
/* Sum rows is of n X n matrix a
   and store 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
.L10:
        addsd   (%rdi), %xmm0      # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

- No need to store intermediate results

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