



Code Optimization

18-213/18-613: Introduction to Computer Systems
26th Lecture, December 6, 2022

Announcements

- **Lab 7 (proxylab) due Fri Dec 9 (last turn-in: Sat Dec 10)**
- **Thurs Dec 8 Lecture: Future of Computing**
 - Guest lectures by 3 Ph.D. students highlighting the latest breakthroughs in systems research
- **Final Exam Review, Wed Dec 14, time TBD**
 - Room TBD or over Zoom
- **Final Exam, Fri Dec 16, 5:30-8:30 pm ET**
 - Pittsburgh: TEP 2610, 2611, 2612, 2700, 2701, 2702
 - SV: TBD

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 modern processors + memory systems operate
 - 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 (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 procedure side-effects
 - potential memory aliasing

Today

■ Generally Useful Optimizations

CSAPP 5.1

- Code motion/precomputation
- Strength reduction
- Sharing of common subexpressions
- Example: Bubblesort

■ Optimization Blockers

CSAPP 5.1

- Procedure calls
- Memory aliasing

■ Exploiting Instruction-Level Parallelism

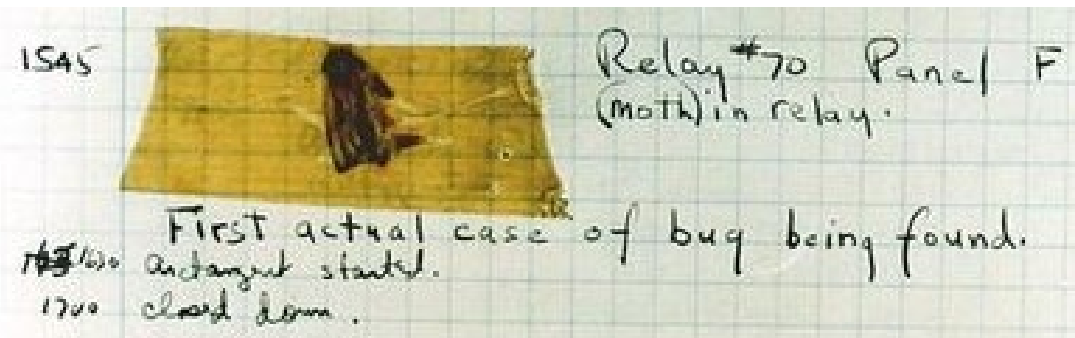
CSAPP 5.2-5.10

■ Dealing with Conditionals

CSAPP 5.11

■ 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”



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];
```


Compiler-Generated Code Motion (-O1)

```
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];
```

```
set_row:
    testq    %rcx, %rcx           # Test n
    jle     .L1                  # If <= 0, goto done
    imulq   %rcx, %rdx           # ni = n*i
    leaq   (%rdi,%rdx,8), %rdx   # rowp = A + ni*8
    movl   $0, %eax             # j = 0
.L3:
    movsd  (%rsi,%rax,8), %xmm0  # t = b[j]
    movsd  %xmm0, (%rdx,%rax,8)  # M[A+ni*8 + j*8] = t
    addq   $1, %rax             # j++
    cmpq   %rcx, %rax           # j:n
    jne    .L3                  # if !=, goto loop
.L1:
    rep ; ret                    # done:
```

Strength Reduction

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
 - $16 * x \rightarrow x \ll 4$
 - Utility is machine dependent
 - Depends on cost of multiply or divide instruction
 - Intel Nehalem: integer multiply takes 3 CPU cycles, add is 1 cycle¹
- Recognize sequence of products

```

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

```

¹https://www.agner.org/optimize/instruction_tables.pdf

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: $i*n$, $(i-1)*n$, $(i+1)*n$

```

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
...

```

```

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: $i*n$

```

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
...

```

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

- **Generally Useful Optimizations**
 - Code motion/precomputation
 - Strength reduction
 - Sharing of common subexpressions
 - Example: Bubblesort
- **Optimization Blockers**
 - Procedure calls
 - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- **Dealing with Conditionals**

- **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”**



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**

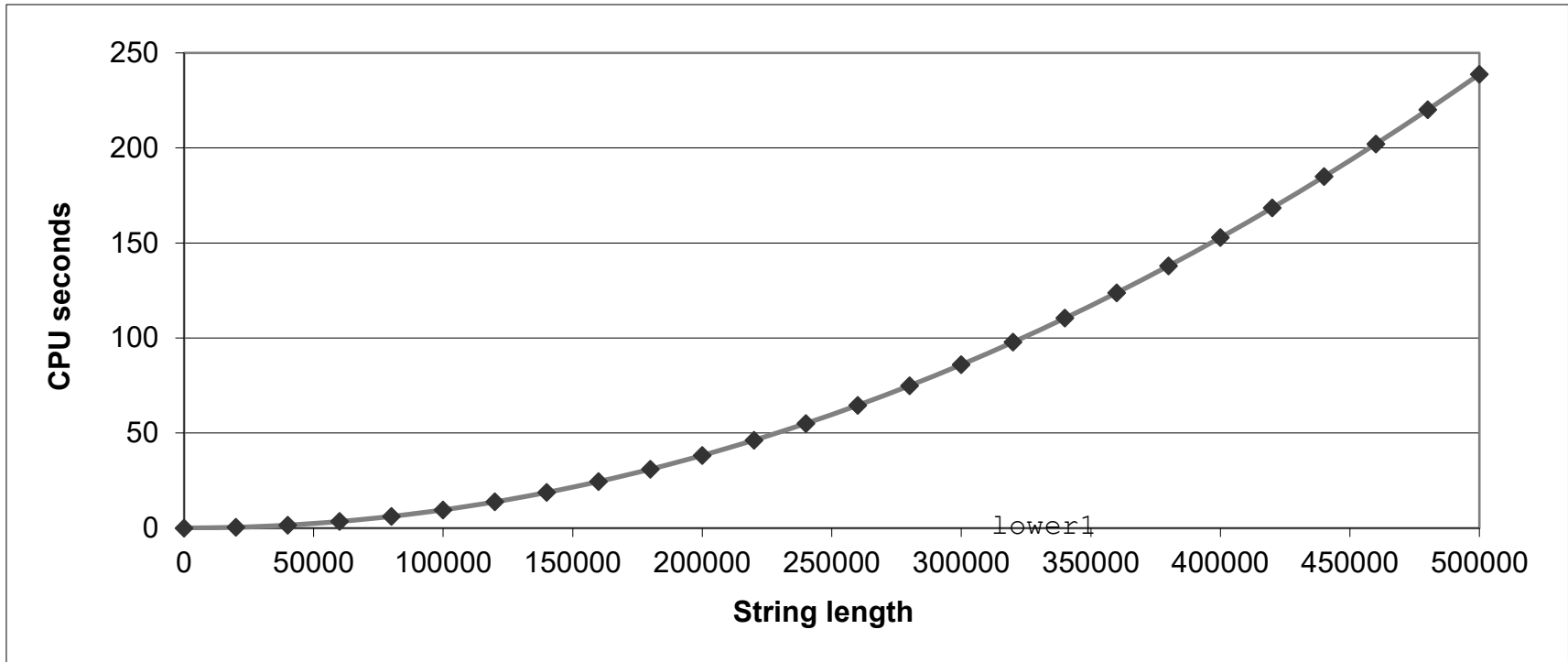
Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```
void lower1(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Extracted from 213 lab submissions

Lower Case Conversion Performance



- Time quadruples when double string length
- Quadratic performance

Calling Strlen

```
void lower1(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' &&
            s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

```
/* 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

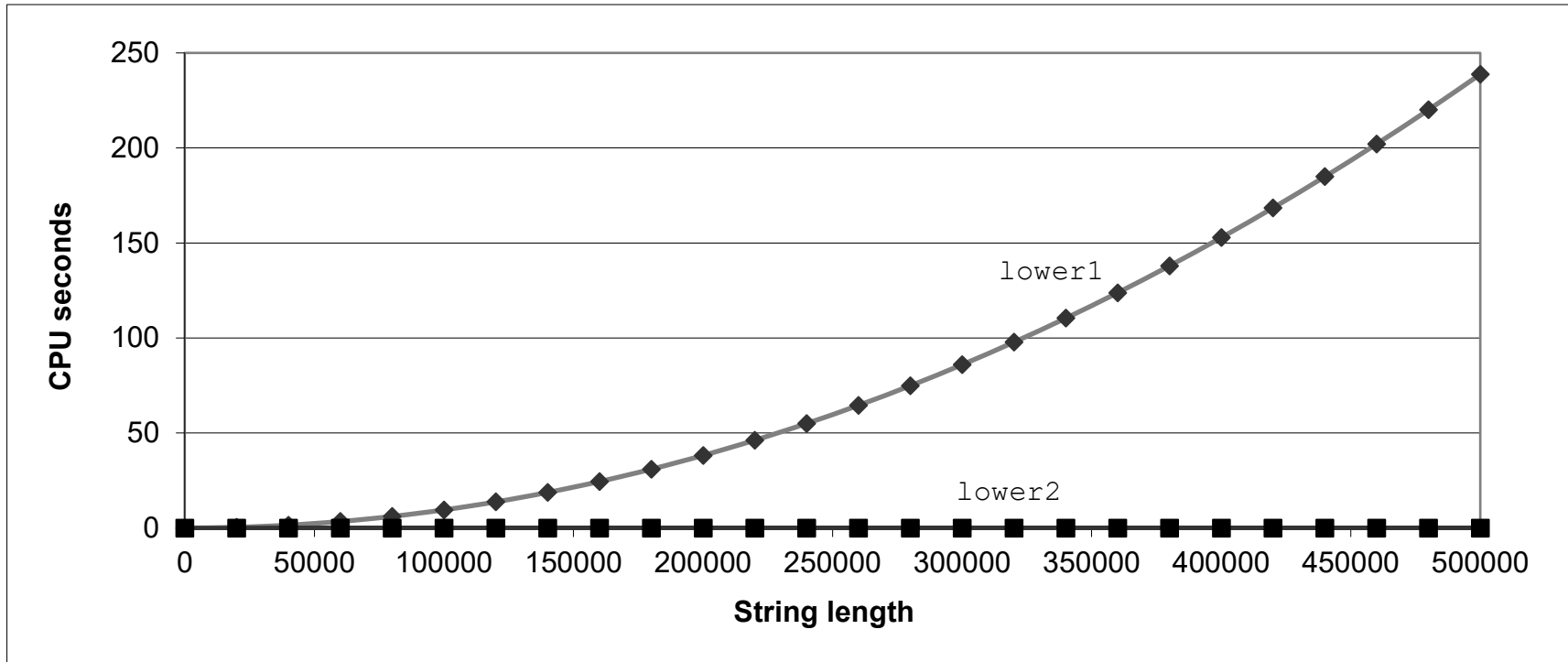
- N calls to strlen (called every time through the loop)
- Require times N, N-1, N-2, ..., 1
- Overall $O(N^2)$ performance

Improving Performance

```
void lower2(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 `lower1` 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

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

Optimization Blocker #2: Memory Aliasing

```

/* 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**

Today

- **Generally Useful Optimizations**
 - Code motion/precomputation
 - Strength reduction
 - Sharing of common subexpressions
 - Example: Bubblesort
- **Optimization Blockers**
 - Procedure calls
 - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- **Dealing with Conditionals**

■ 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

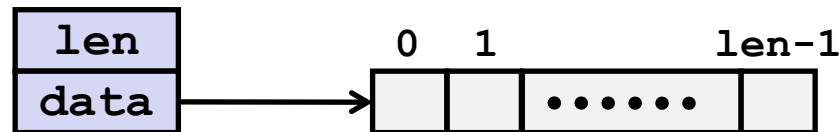


Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**
 - Hardware can execute multiple instructions in parallel
- **Performance limited by data dependencies**
- **Simple transformations can cause big speedups**
 - Compilers often cannot make these transformations
 - Lack of associativity and distributivity in floating-point arithmetic

Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */  
typedef struct{  
    size_t len;  
    data_t *data;  
} vec;
```



■ Data Types

- Use different declarations for `data_t`
- `int`
- `long`
- `float`
- `double`

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

Benchmark Computation

```
void combinel(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

■ Data Types

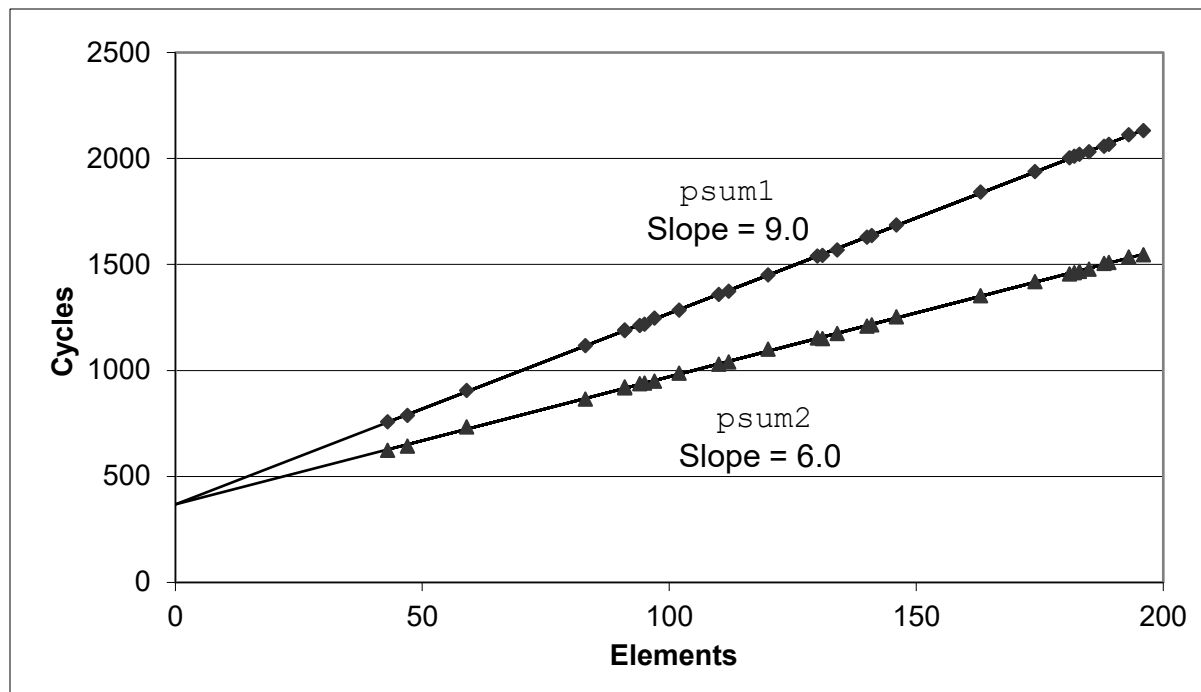
- Use different declarations for `data_t`
- `int`
- `long`
- `float`
- `double`

■ Operations

- Use different definitions of `OP` and `IDENT`
- `+` / `0`
- `*` / `1`

Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: **CPE = cycles per OP**
- **Cycles = CPE * n + Overhead**
 - CPE is slope of line



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 | |
|----------------------|---------|-------|-----------|-------|
| | Add | Mult | Add | Mult |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 -O1 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine1 -O3 | 4.5 | 4.5 | 6 | 7.8 |

Results in CPE (cycles per element)

Basic Optimizations

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

- Move `vec_length` out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Effect of Basic Optimizations

```

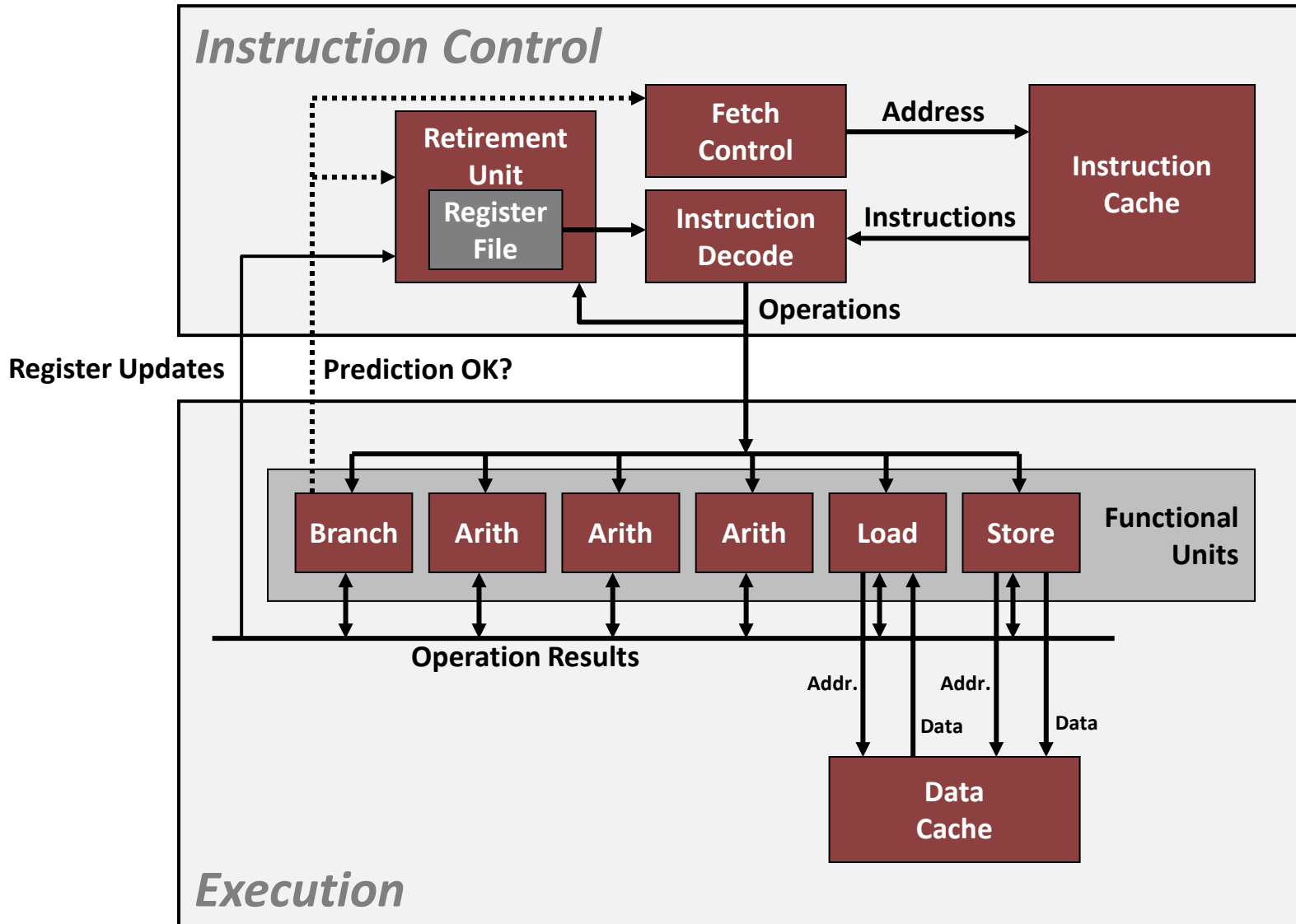
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}

```

| Method | Integer | | Double FP | |
|--------------|---------|-------|-----------|-------|
| | Add | Mult | Add | Mult |
| Combine1 -O1 | 10.12 | 10.12 | 10.17 | 11.14 |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |

- Eliminates sources of overhead in loop

Modern CPU Design

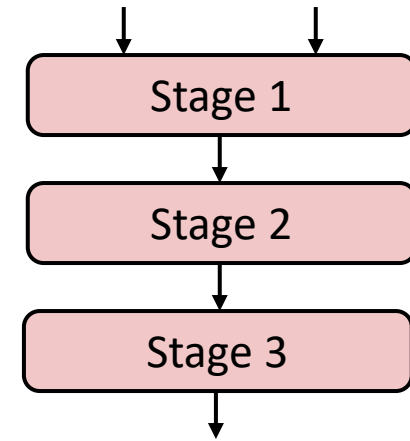


Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have.
- **Most CPUs are superscalar.**
 - Intel: since Pentium (1993)

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



| | Time | | | | | | |
|---------|------|-----|-----|-----|-------|-------|-------|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Stage 1 | a*b | a*c | | | p1*p2 | | |
| Stage 2 | | a*b | a*c | | | p1*p2 | |
| Stage 3 | | | a*b | a*c | | | p1*p2 |

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to $i+1$
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Quiz Time!

Canvas Quiz: Day 26 – Optimization

Haswell CPU

- **8 Total Functional Units**
- **Multiple instructions can execute in parallel**

2 load, with address computation

1 store, with address computation

4 integer

2 FP multiply

1 FP add

1 FP divide

- **Some instructions take > 1 cycle, but can be pipelined**

| <i>Instruction</i> | <i>Latency</i> | <i>Cycles/Issue</i> |
|--------------------------------|----------------|---------------------|
| Load / Store | 4 | 1 |
| Integer Multiply | 3 | 1 |
| Integer/Long Divide | 3-30 | 3-30 |
| Single/Double FP Multiply | 5 | 1 |
| Single/Double FP Add | 3 | 1 |
| Single/Double FP Divide | 3-15 | 3-15 |

x86-64 Compilation of Combine4

■ Inner Loop (Case: Integer Multiply)

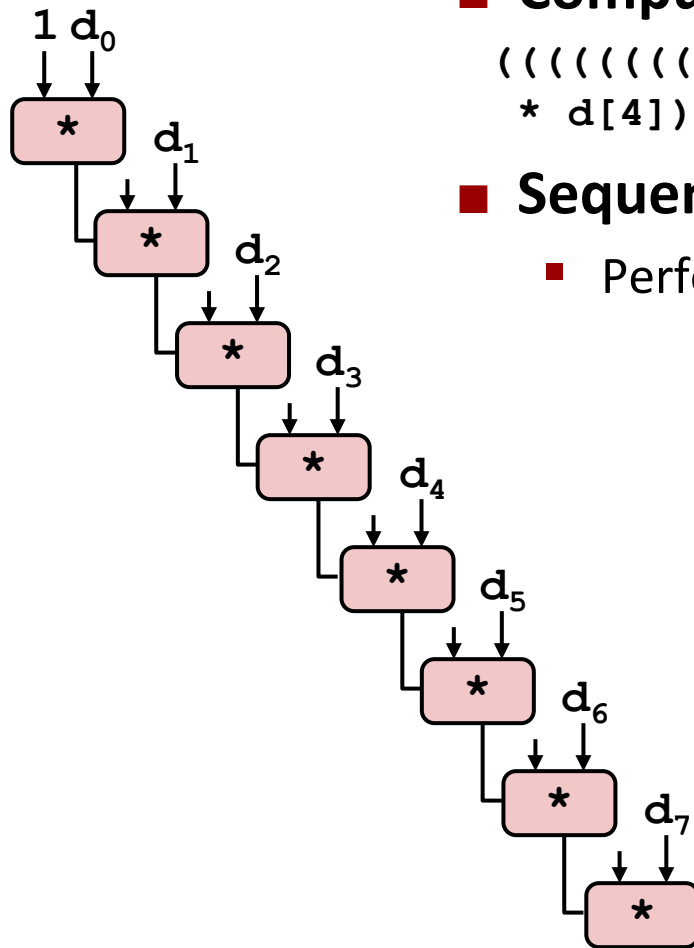
```

.L519:                                # Loop:
    imull  (%rax,%rdx,4), %ecx        # t = t * d[i]
    addq   $1, %rdx                  # i++
    cmpq   %rdx, %rbp                # Compare length:i
    jg     .L519                      # If >, goto Loop

```

| Method | Integer | | Double FP | |
|----------------------|-------------|-------------|-------------|-------------|
| | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| <i>Latency Bound</i> | <i>1.00</i> | <i>3.00</i> | <i>3.00</i> | <i>5.00</i> |

Combine4 = Serial Computation (OP = *)



■ Computation (length=8)

```
((((( ((( (1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7]))
```

■ Sequential dependence

- Performance: determined by latency of OP

Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per loop iteration

Effect of Loop Unrolling

| Method | Integer | | Double FP | |
|----------------------|-------------|-------------|-------------|-------------|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| <i>Latency Bound</i> | <i>1.00</i> | <i>3.00</i> | <i>3.00</i> | <i>5.00</i> |

- **Helps integer add**
 - Achieves latency bound
- **Others don't improve. *Why?***
 - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Can this change the result of the computation?
- Yes, for FP. *Why?*

Effect of Separate Accumulators

| Method | Integer | | Double FP | |
|-------------------------|---------|------|-----------|------|
| | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| Unroll 2x2 | 0.81 | 1.51 | 1.51 | 2.51 |
| <i>Latency Bound</i> | 1.00 | 3.00 | 3.00 | 5.00 |
| <i>Throughput Bound</i> | 0.50 | 1.00 | 1.00 | 0.50 |

4 func. units for int +,
2 func. units for load
Why Not .25?

1 func. unit for FP +
3-stage pipelined FP +

2 func. units for FP *,
2 func. units for load
5-stage pipelined FP *

■ Nearly 2x speedup for Int *, FP +, FP *

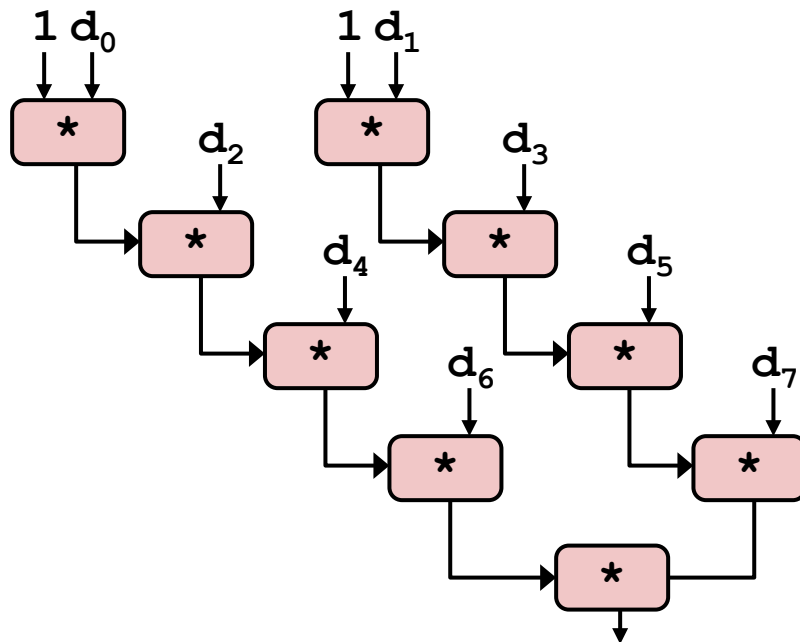
- Reason: Breaks sequential dependency

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

- Why is that? (next slide)

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



■ What changed:

- Two independent “streams” of operations

■ Overall Performance

- N elements, D cycles latency/op
- Should be $(N/2+1)*D$ cycles:
CPE = D/2
- CPE matches prediction!

What Now?

Unrolling & Accumulating

■ Idea: Can choose L Unrolling Factor and K Accumulators

- L must be a multiple of K

■ Case: Double *

- Latency bound: 5.00. Throughput bound: 0.50

| Accumulators | FP * | Unrolling Factor L | | | | | | | |
|--------------|------|--------------------|------|------|------|------|------|------|------|
| | K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 5.01 | 5.01 | 5.01 | 5.01 | 5.01 | 5.01 | 5.01 | 5.01 | |
| 2 | | 2.51 | | 2.51 | | 2.51 | | | |
| 3 | | | 1.67 | | | | | | |
| 4 | | | | 1.25 | | 1.26 | | | |
| 6 | | | | | 0.84 | | | | 0.88 |
| 8 | | | | | | 0.63 | | | |
| 10 | | | | | | | 0.51 | | |
| 12 | | | | | | | | | 0.52 |

Achievable Performance

| Method | Integer | | Double FP | |
|-------------------------|-------------|-------------|-------------|-------------|
| Operation | Add | Mult | Add | Mult |
| Best | 0.54 | 1.01 | 1.01 | 0.52 |
| <i>Latency Bound</i> | <i>1.00</i> | <i>3.00</i> | <i>3.00</i> | <i>5.00</i> |
| <i>Throughput Bound</i> | <i>0.50</i> | <i>1.00</i> | <i>1.00</i> | <i>0.50</i> |

- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

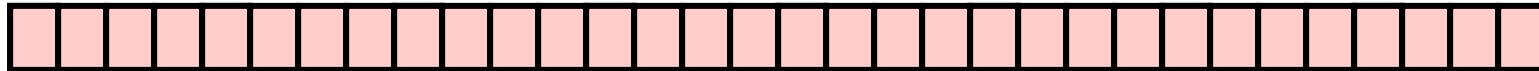
Can we do even better?

Programming with AVX2

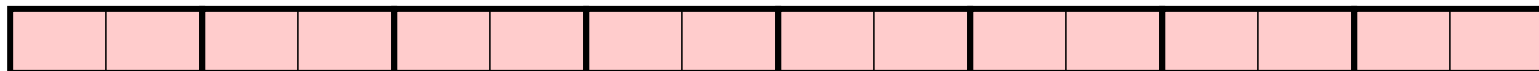
YMM Registers

■ 16 total, each 32 bytes

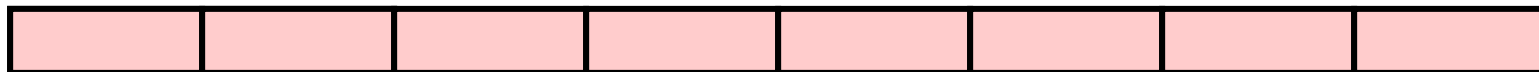
■ 32 single-byte integers



■ 16 16-bit integers



■ 8 32-bit integers



■ 8 single-precision floats



■ 4 double-precision floats



■ 1 single-precision float



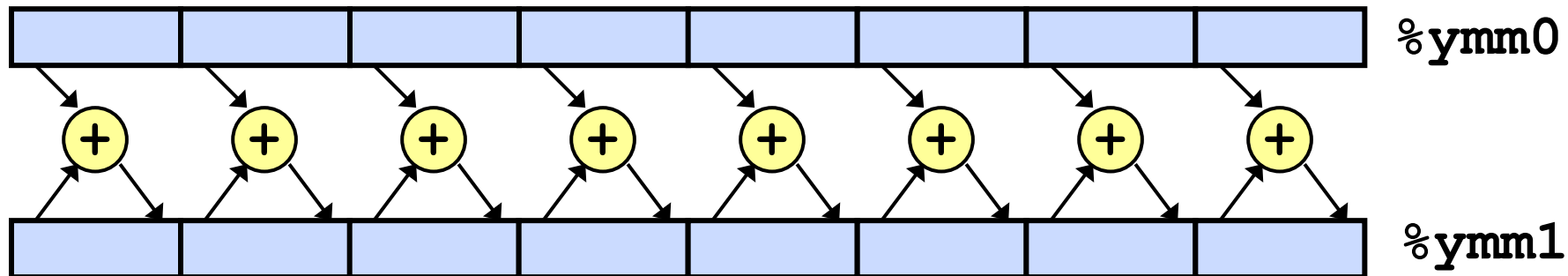
■ 1 double-precision float



SIMD Operations

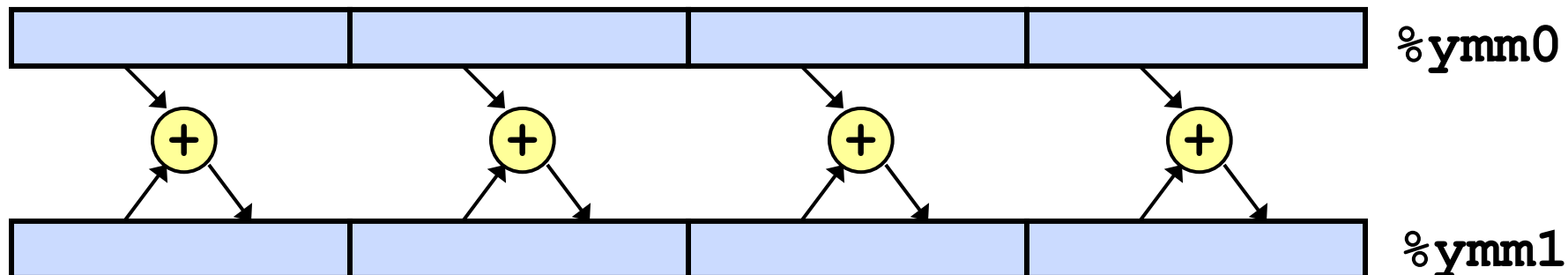
■ SIMD Operations: Single Precision

```
vaddps %ymm0, %ymm1, %ymm1
```



■ SIMD Operations: Double Precision

```
vaddpd %ymm0, %ymm1, %ymm1
```



Using Vector Instructions

| Method | Integer | | Double FP | |
|-----------------------------|-------------|-------------|-------------|-------------|
| Operation | Add | Mult | Add | Mult |
| Scalar Best | 0.54 | 1.01 | 1.01 | 0.52 |
| Vector Best | 0.06 | 0.24 | 0.25 | 0.16 |
| <i>Latency Bound</i> | <i>0.50</i> | <i>3.00</i> | <i>3.00</i> | <i>5.00</i> |
| <i>Throughput Bound</i> | <i>0.50</i> | <i>1.00</i> | <i>1.00</i> | <i>0.50</i> |
| <i>Vec Throughput Bound</i> | <i>0.06</i> | <i>0.12</i> | <i>0.25</i> | <i>0.12</i> |

■ Make use of AVX Instructions

- Parallel operations on multiple data elements
- See Web Aside OPT:SIMD on CS:APP web page

Today

- **Generally Useful Optimizations**
 - Code motion/precomputation
 - Strength reduction
 - Sharing of common subexpressions
 - Example: Bubblesort
- **Optimization Blockers**
 - Procedure calls
 - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- **Dealing with Conditionals**

What About Branches?

■ Challenge

- **Instruction Control Unit** must work well ahead of **Execution Unit** to generate enough operations to keep EU busy

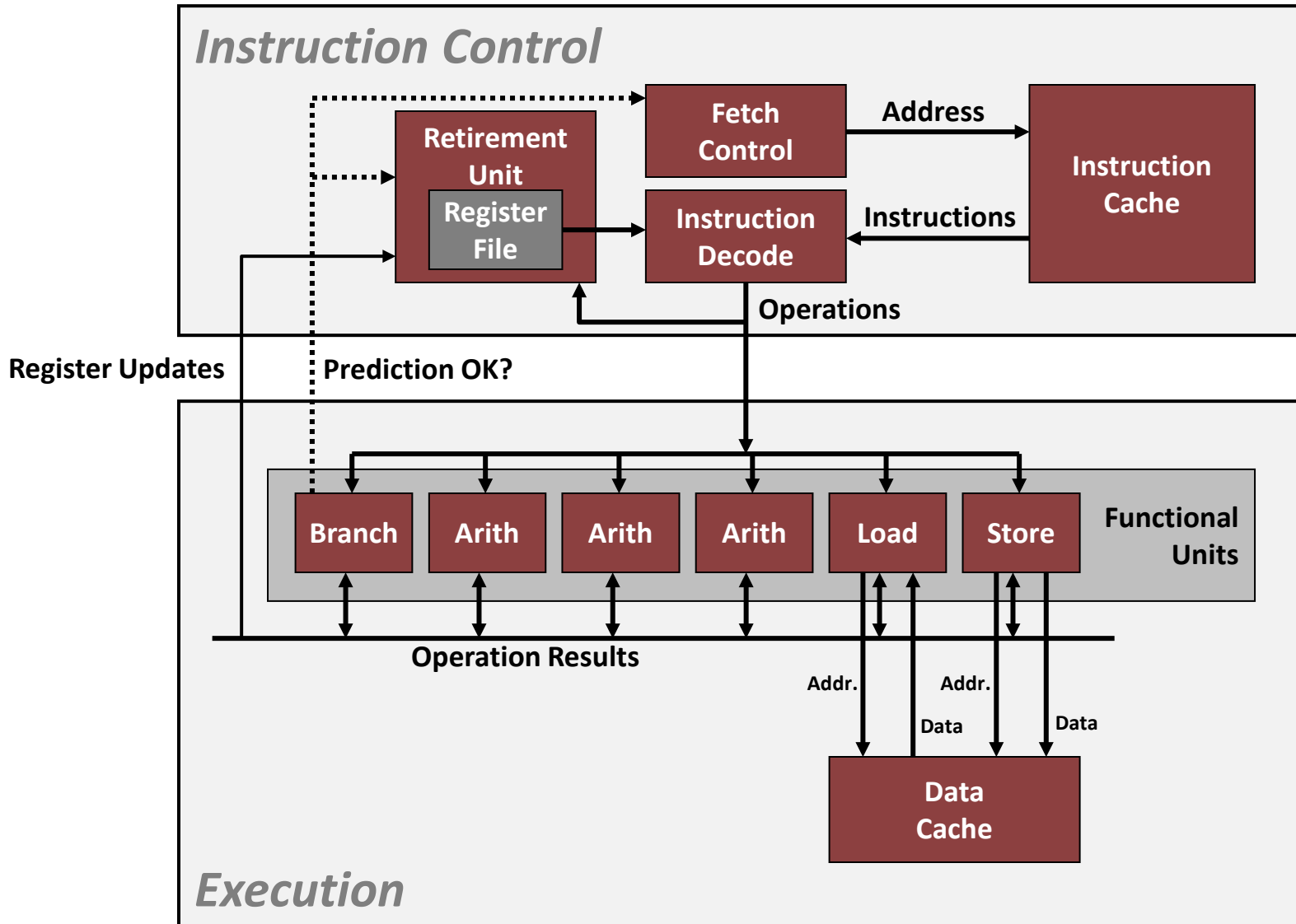
```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax
. . .
404685:  repz  retq
```

} Executing

← How to continue?

- When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design



Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
 - Branch Taken: Transfer control to branch target
 - Branch Not-Taken: Continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax
. . .
404685:  repz  retq
```

Branch Not-Taken

Branch Taken

Branch Prediction

■ Idea

- Guess which way branch will go
- Begin executing instructions at predicted position
 - But don't actually modify register or memory data

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

. . .

404685:  repz  retq
```

Predict Taken

} **Begin
Execution**

Branch Prediction Through Loop

```

401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029

```

i = 98

Assume
vector length = **100**

Predict Taken (OK)

```

401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029

```

i = 99

Predict Taken
(Oops)

```

401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029

```

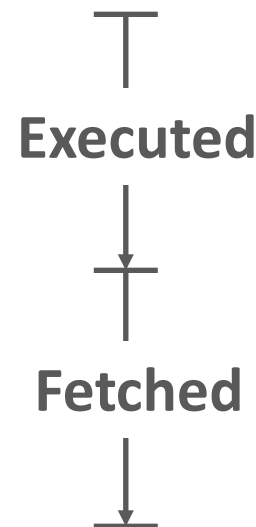
i = 100

Read
invalid
location

```

401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029

```

i = 101

Branch Misprediction Invalidation

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 98

Assume
vector length = **100**

Predict Taken (OK)

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 99

Predict Taken
(Oops)

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 100

Invalidate

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 101

Branch Misprediction Recovery

```
401029: vmulsd (%rdx), %xmm0, %xmm0
```

```
40102d: add     $0x8, %rdx
```

```
401031: cmp     %rax, %rdx
```

```
401034: jne     401029
```

```
401036: jmp     401040
```

```
. . .
```

```
401040: vmovsd %xmm0, (%r12)
```

i = 99

Definitely not taken

Reload
Pipeline

■ Performance Cost

- Multiple clock cycles on modern processor
- Can be a major performance limiter

Branch Prediction Numbers

- **Default behavior:**
 - Backwards branches are often loops so predict taken
 - Forwards branches are often if so predict not taken
- **Predictors average better than 95% accuracy**
 - Most branches are already predictable.
- **Annual branch predictor contests at top Computer Architecture conferences (2010-2016)**
 - Metrics: Size of branch predictor tables
Mispredictions per kilo-instruction (MPKI)
 - 2016 Winners (<https://www.jilp.org/cbp2016/>)
 - Size 8KB: MPKI=4.1
 - Size 64KB: MPKI=3.3

Summary: Getting High Performance

- **Good compiler and flags**
- **Don't do anything sub-optimal**
 - Watch out for hidden algorithmic inefficiencies
 - Write compiler-friendly code
 - Watch out for optimization blockers:
procedure calls & memory references
 - Look carefully at innermost loops (where most work is done)
- **Tune code for machine**
 - Exploit instruction-level parallelism
 - Avoid unpredictable branches
 - Make code cache friendly

ADDITIONAL SLIDES

Loop Unrolling with Reassociation (2x1a)

```

void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}

```

Compare to before

$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$

- Can this change the result of the computation?
- Yes, for FP. *Why?*

Effect of Reassociation

| Method | Integer | | Double FP | |
|-------------------------|-------------|-------------|-------------|-------------|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1 | 1.01 | 3.01 | 3.01 | 5.01 |
| Unroll 2x1a | 1.01 | 1.51 | 1.51 | 2.51 |
| <i>Latency Bound</i> | <i>1.00</i> | <i>3.00</i> | <i>3.00</i> | <i>5.00</i> |
| <i>Throughput Bound</i> | <i>0.50</i> | <i>1.00</i> | <i>1.00</i> | <i>0.50</i> |

4 func. units for int +,
2 func. units for load
Why Not .25?

1 func. unit for FP +
3-stage pipelined FP +

2 func. units for FP *,
2 func. units for load
5-stage pipelined FP *

■ Nearly 2x speedup for Int *, FP +, FP *

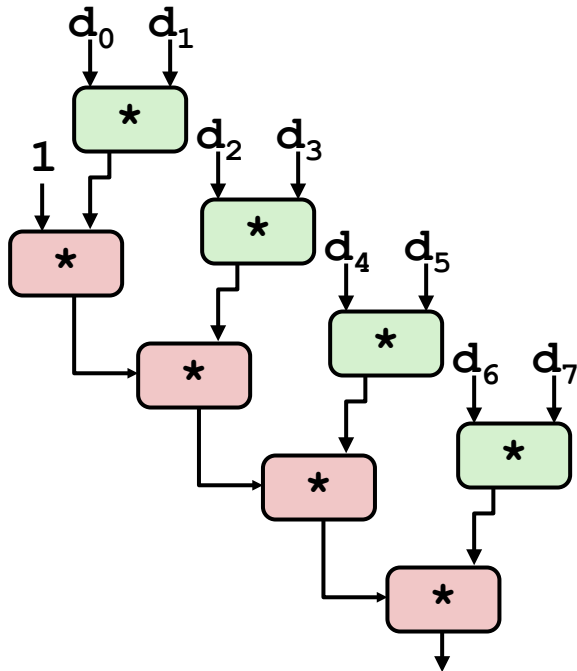
- Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

- Why is that? (next slide)

Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



■ What changed:

- Ops in the next iteration can be started early (no dependency)

■ Overall Performance

- N elements, D cycles latency/op
- $(N/2+1)*D$ cycles:
 $CPE = D/2$

Unrolling & Accumulating: Int +

■ Case

- Intel Haswell
- Integer addition
- Latency bound: 1.00. Throughput bound: 0.50

| FP * | Unrolling Factor L | | | | | | | | |
|---------------------|--------------------|------|------|------|------|------|------|------|------|
| | K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| <i>Accumulators</i> | 1 | 1.27 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 | |
| | 2 | | 0.81 | | 0.69 | | 0.54 | | |
| | 3 | | | 0.74 | | | | | |
| | 4 | | | | 0.69 | | 1.24 | | |
| | 6 | | | | | 0.56 | | | 0.56 |
| | 8 | | | | | | 0.54 | | |
| | 10 | | | | | | | 0.54 | |
| | 12 | | | | | | | | 0.56 |