# Lecture 12: Program optimization

Randal E. Bryant and David R. O'Hallaron (CMU)

(Minor changes for CSF by Xin Jin and David Hovemeyer)

February 19, 2024

601.229 Computer Systems Fundamentals

# Today

- **Overview**
- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing

# 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 memory aliasing
  - potential procedure side-effects

# Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - 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 procedures**
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files
- **Most analysis is based only on *static* information**
  - Compiler has difficulty anticipating run-time inputs
- **When in doubt, the compiler must be conservative**

# 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];
```

# Reduction in Strength

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

  `16*x  →  x << 4`

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++) {
  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;
}
```

# 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**

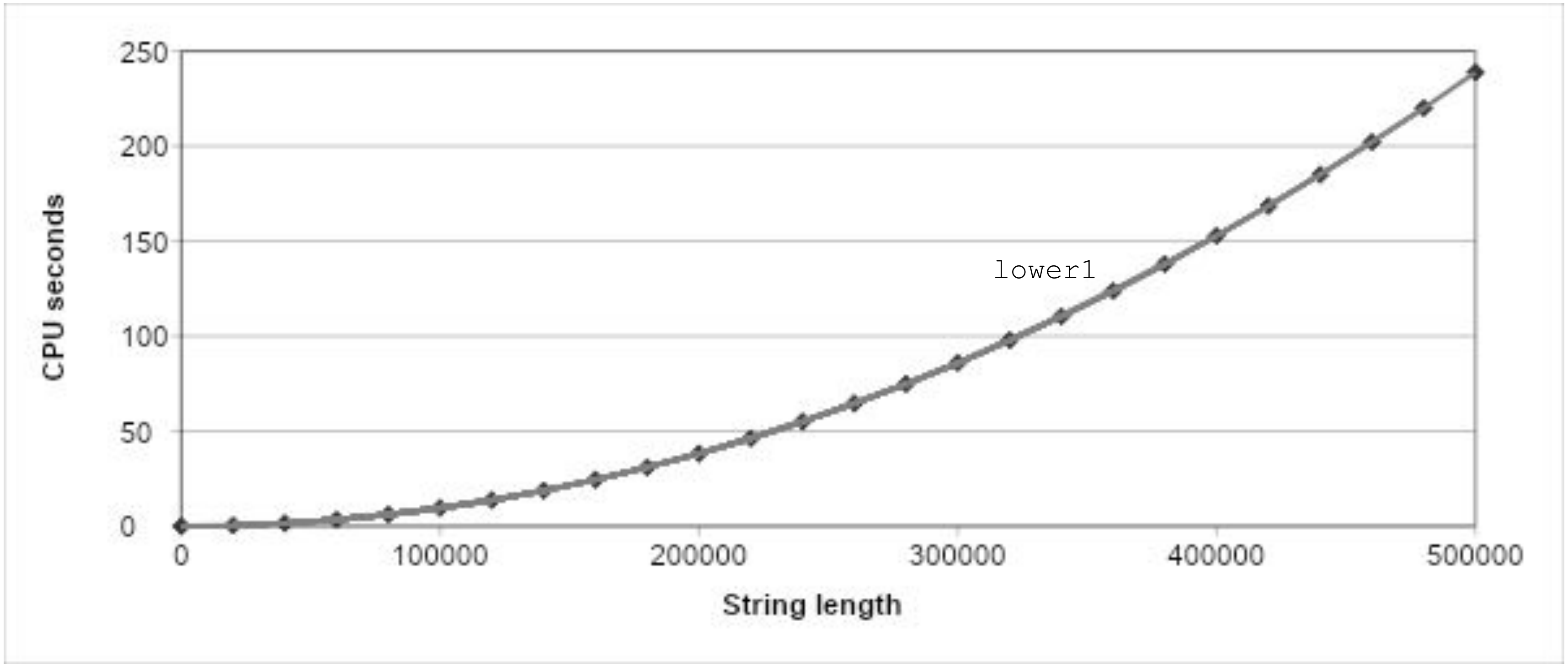
# Optimization Blocker #1: Procedure Calls

- **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

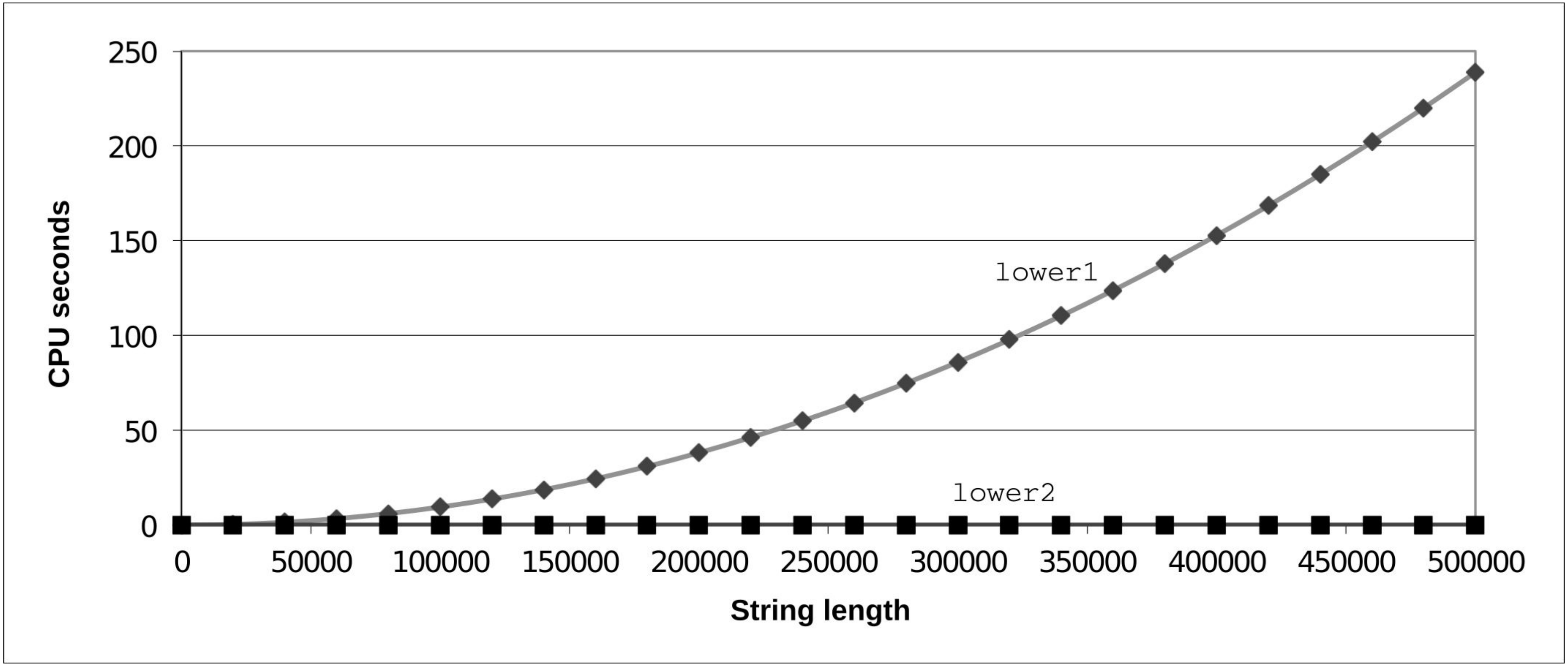
# Improving Performance

```
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
- 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 treats 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 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];
    }
}
```

```
# 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 = A+3;

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

**Value of B:**

```
init:   [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

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

# 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

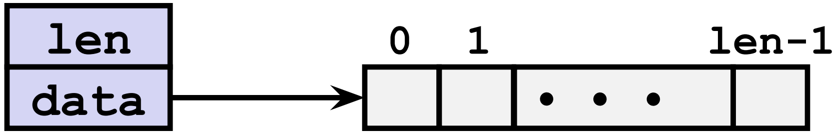
- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

# 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 yield dramatic performance improvement**
  - 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;
```




```
/* 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;
}
```

- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

# Benchmark Computation

```
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**

## Data Types

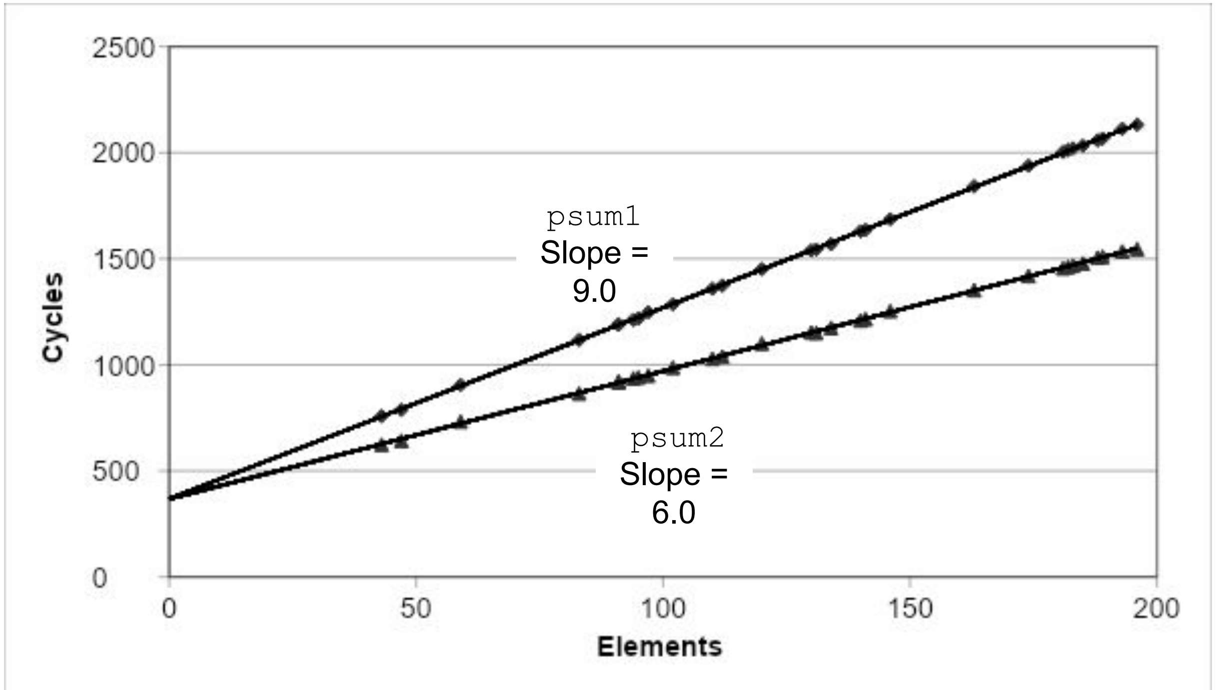
- Use different declarations for `data_t`
- `int`
- `long`
- `float`
- `double`

## 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**
- **T = 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 | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| Combine1 –O1 | 10.12 | 10.12 | 10.17 | 11.14 |

# 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

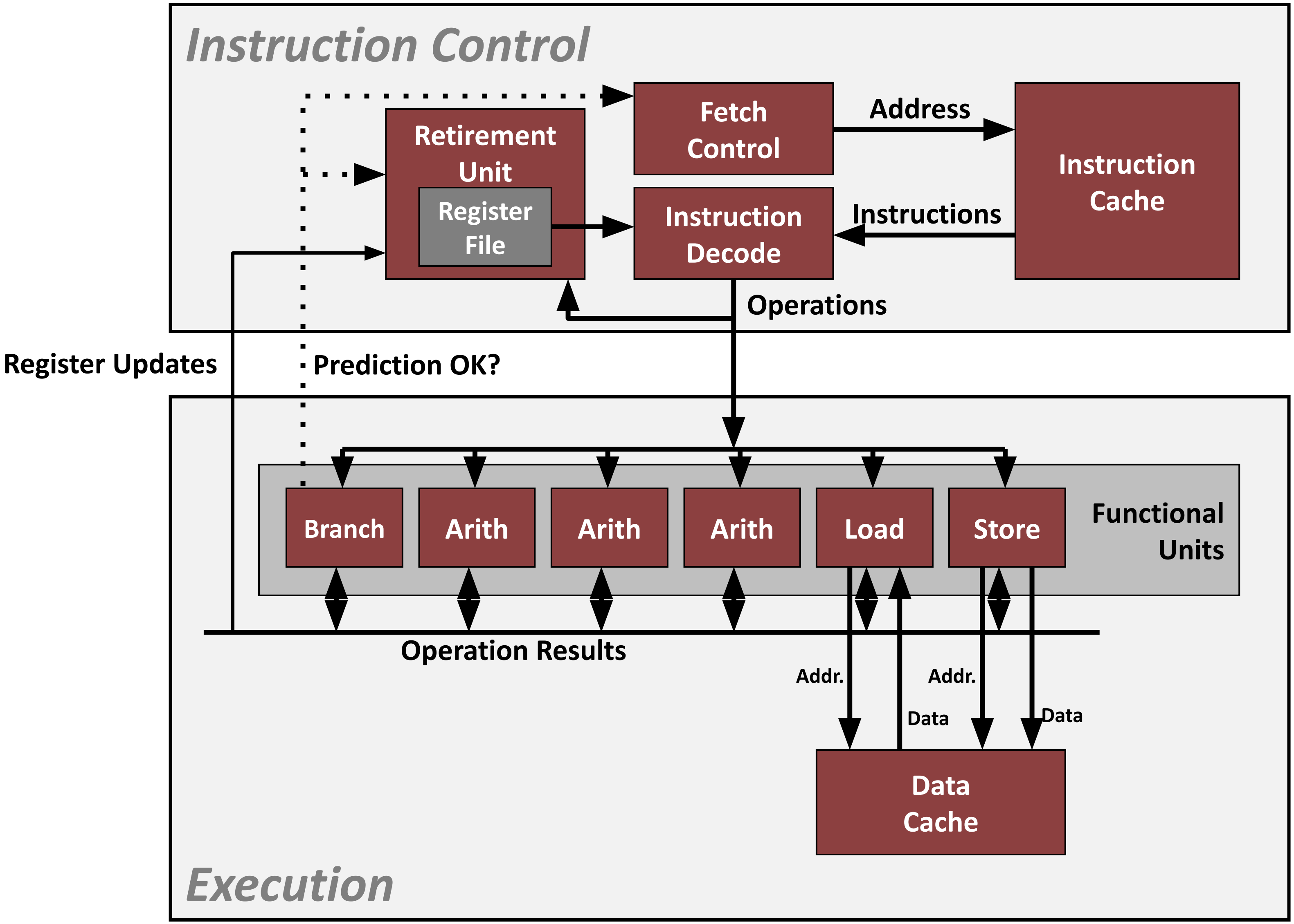
# Effect of Basic Optimizations

```
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 | |
|---|---|---|---|---|
| Operation | 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

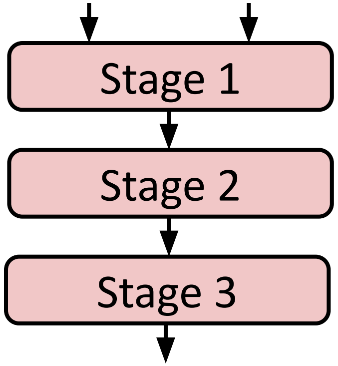
- **Definition:** A superscalar processor can issue and execute ***multiple instructions in one cycle***. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- Benefit: without programming effort, superscalar processor can take advantage of the ***instruction level parallelism*** that most programs have

- Most modern 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

# Haswell CPU

- **8 Total Functional Unit**
- **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**

| ***Instruction*** | ***Latency*** | ***Cycles/Issue*** |
|---|---|---|
| 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** |

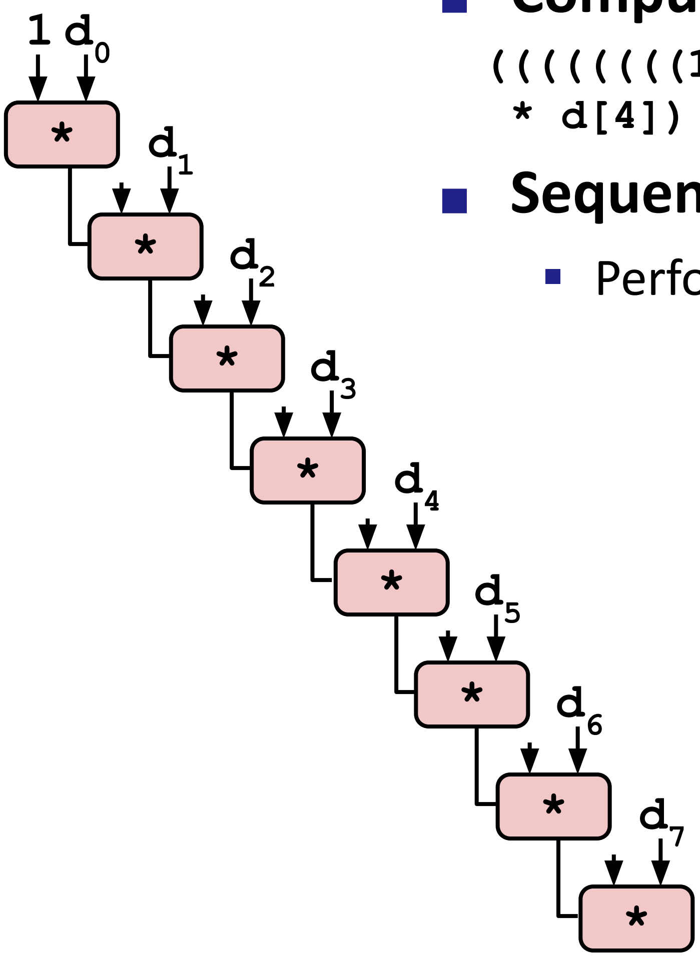
# x86-64 Compilation of Combine4

- **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 | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| Combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| Latency Bound | 1.00 | 3.00 | 3.00 | 5.00 |

# 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 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 |
| **Latency Bound** | 1.00 | 3.00 | 3.00 | 5.00 |

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

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

# Getting High Performance

- **Good compiler and flags**
- **Don't do anything stupid**
  - 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 (Covered later in course)