Program Optimization

CSE4009: System Programming

Woong Sul

Today

- Overview
- Generally Useful Optimizations
  - Code motion
  - Strength reduction
  - Sharing of common sub-expressions
  - Removing unnecessary procedure calls
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploiting HW-based Parallelism
  - Instruction-level Parallelism
  - Data-level Parallelism

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 and 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

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor and compilers
  - Code motion
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls

1. Code Motion

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

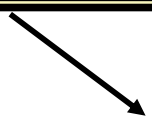
```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

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

Compiler Can Do 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:                                    # loop:
        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:                                    # done:
        rep ; ret
```

2. 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;
}
```

3. 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
```

Optimization Blockers

- Compilers can optimize your code but not always

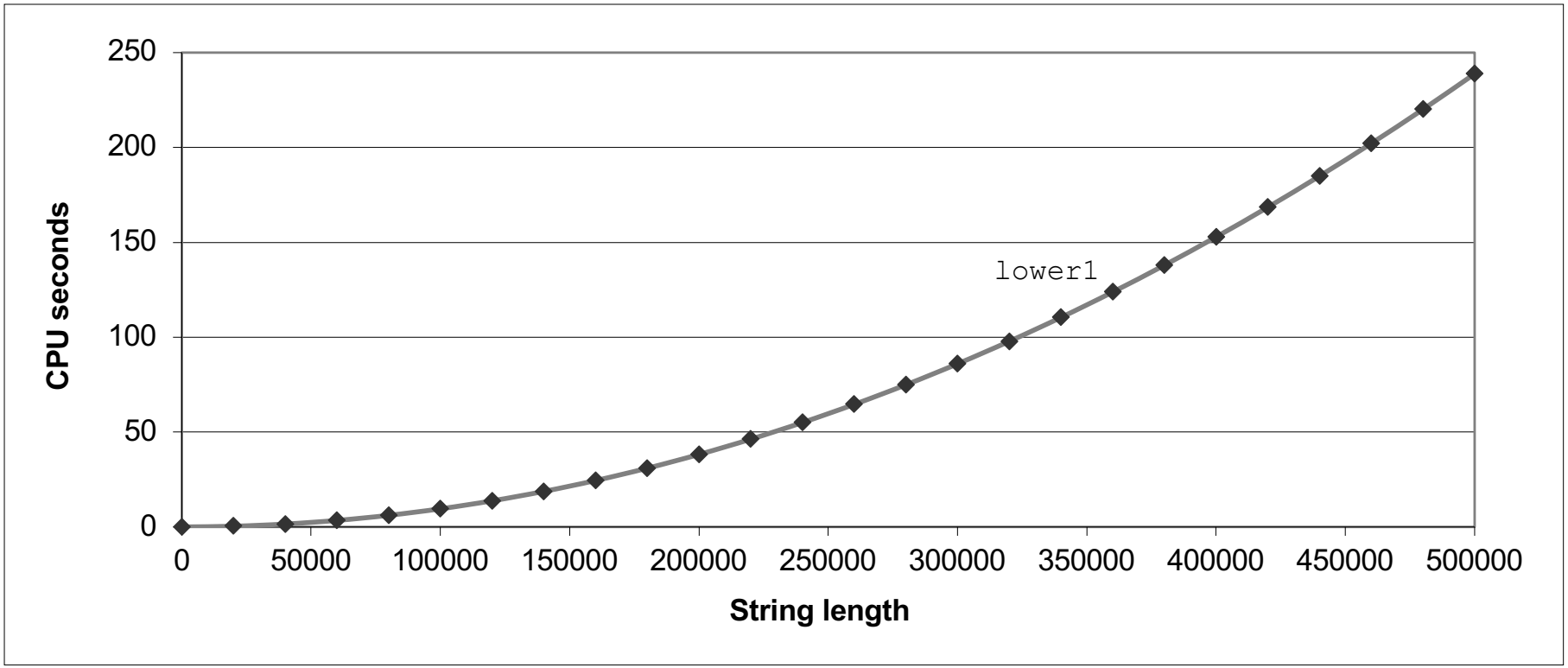
- Procedure to Convert String to Lower Case

```
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');
}
```

Could you tell the performance of this function?

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance ➔ ***O(n²)***



Convert Loop To goto Form

- Secrets behind the scenes

  Loop conditions are checked on every iteration

  `strlen()` executed on every iteration!!

```
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:
}
```

Calling `strlen()` on Every Iteration

- `strlen()` performance ➔ ***O(N)***
  - Scan the entire length, looking for null character
- Overall performance, string of length N ➔ ***O(N^2)***
  - N calls to `strlen()`

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

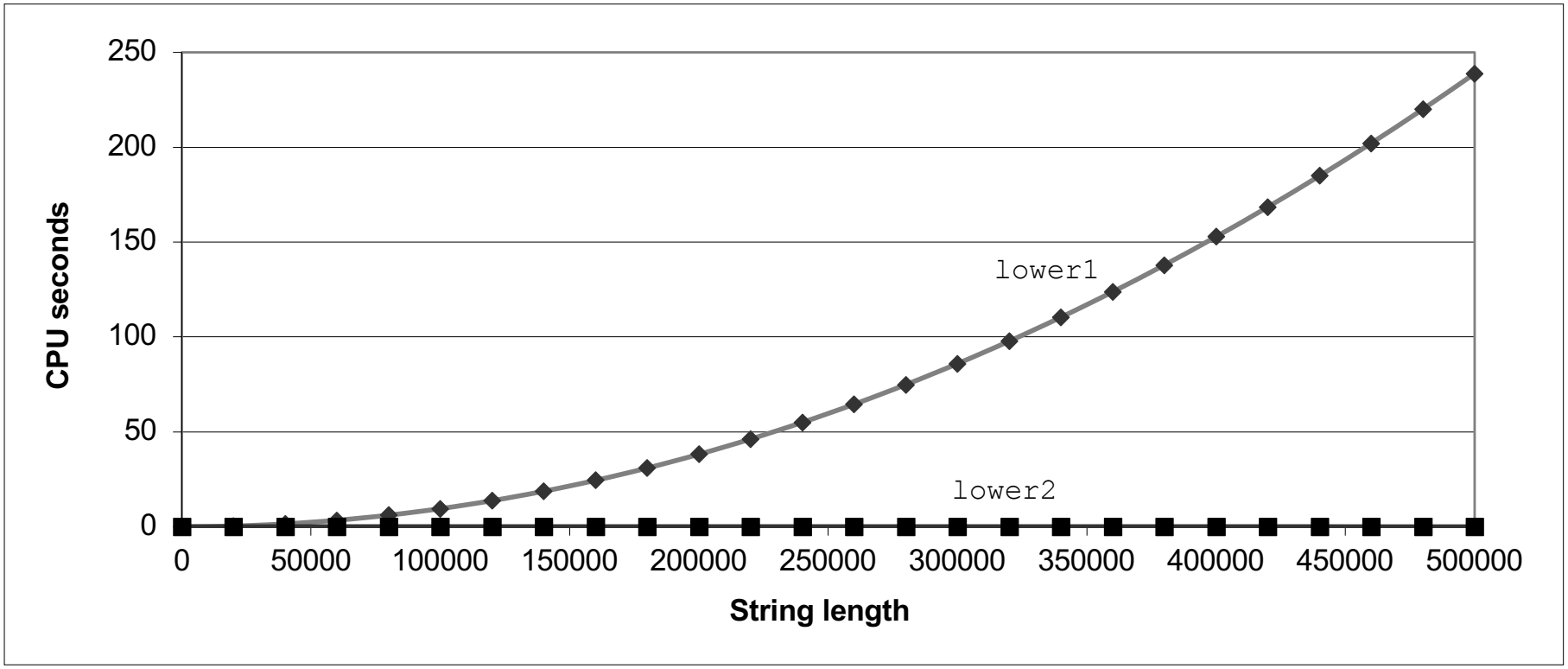
Improving Performance by Code Motion

- Move call to `strlen()` outside of loop
- Since result does not change from one iteration to another

```
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');
}
```

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



Optimization Blocker #1: 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;
}
```

Another Optimization Blocker Example

- Memory access to `b[i]` occurs on every iteration
- Why couldn't compiler optimize this away?

```
/* 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
```

Memory Aliasing

- Why does compiler leave `b[i]` on every iteration?
- Must consider possibility that these updates will affect program behavior

```
/* 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);
```

Value of B:

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

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

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

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

Removing Aliasing

- Compilers NEVER know whether aliasing was intended
- No need to store intermediate results

```
/* 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;
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

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

Value of B:

```
i = 2: [3, 27, 224]
```

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

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0      # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

Optimization Blocker #2: Memory Aliasing

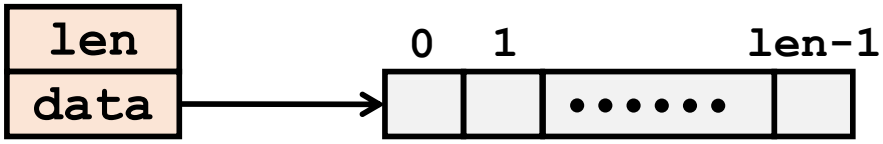
- Aliasing
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - **Your way of telling compiler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

Beyond General Optimizations

- Exploiting ***Instruction-Level Parallelism*** (or ***ILP***)
  - Execute multiple instructions at the same time
  - Reduce average instruction cycle
- 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

- Various data types for vectors (i.e., `data_t`)
  - `int`
  - `long`
  - `float`
  - `double`




```
/* 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;
}
```

Configuring Computations in Benchmark

- **data_t**
  - int
  - long
  - float
  - double
- **OP** and **IDENT**
  - + / 0
  - * / 1

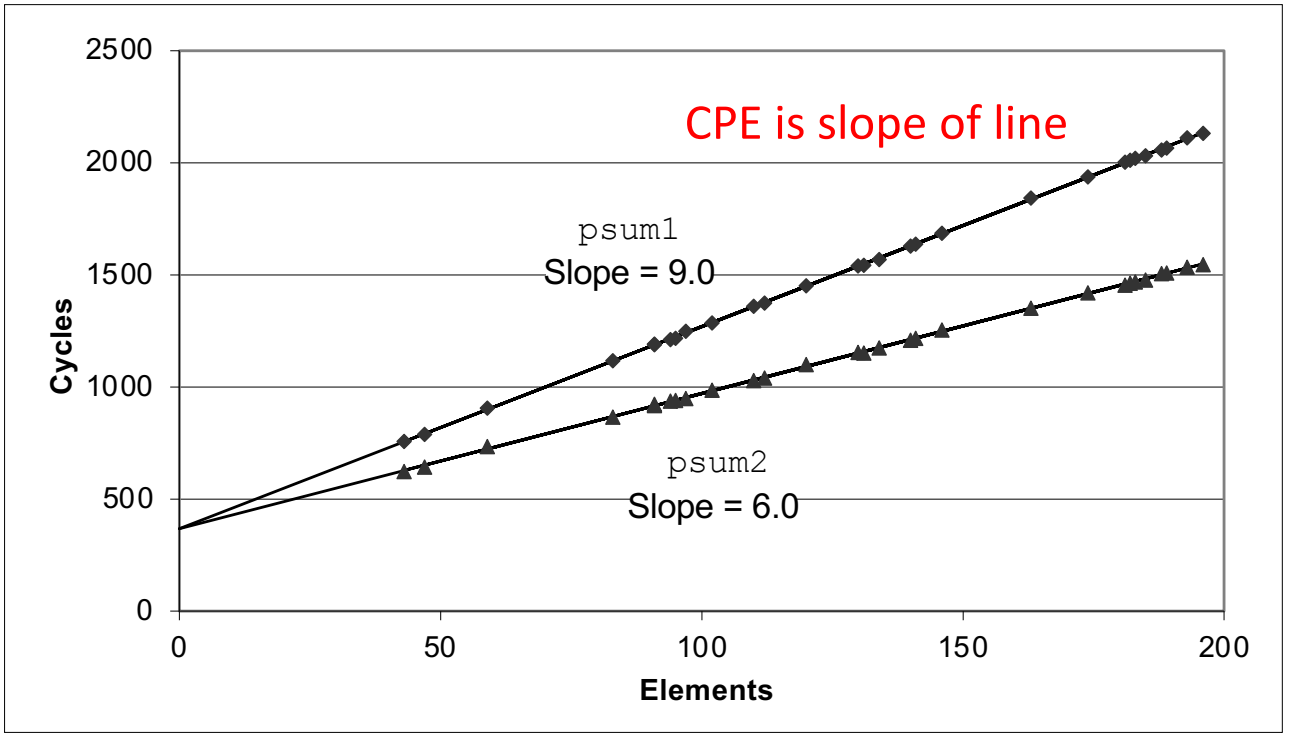
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute *sum* or *product* of vector elements

Cycles Per Element (CPE)

- Performance of program that operates on vectors
  - n = number of elements
  - CPE = cycles per **OP**
  - T = CPE*n + Overhead

  The longer cycles or time, the lower performance



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 (`combine4`)

- Compiler optimizations have some limitation
  - Move `vec_length()` out of loop
  - Avoid bounds check on each cycle
  - Accumulate in temporary

```
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;
}
```

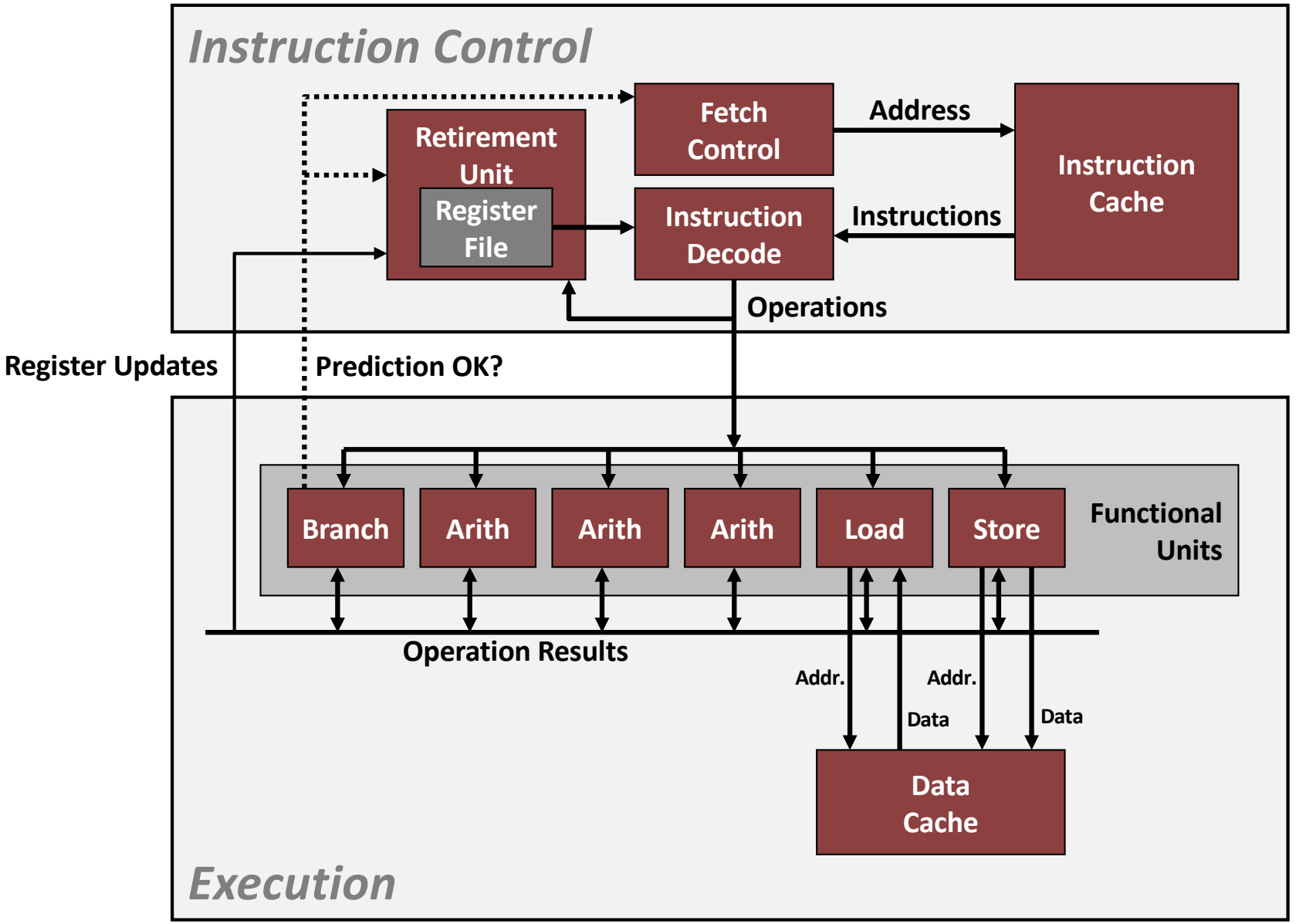
Effect of Basic Optimizations

- Eliminates sources of overhead in loop

```
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;
}
```

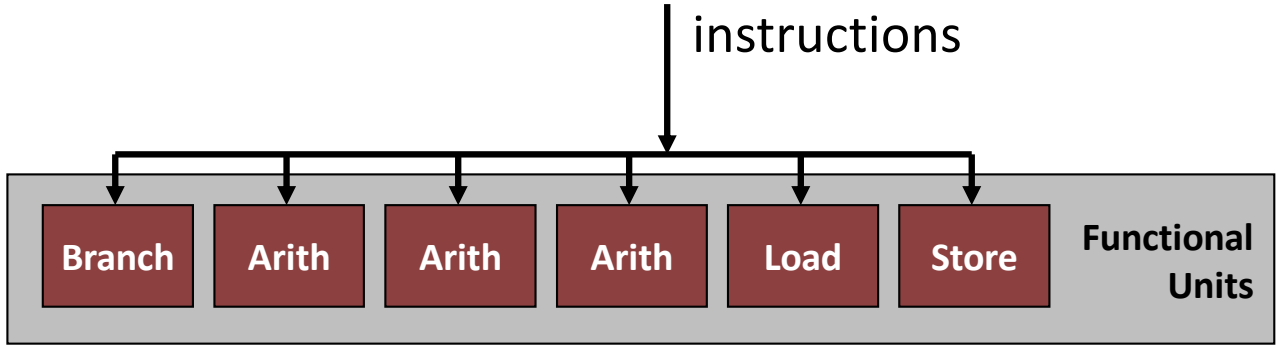
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

Modern CPU Design



Superscalar Processors

- Issue and execute *multiple instructions in one cycle*
  - Fetched from a sequential instruction stream
  - Usually scheduled to execute instructions ***dynamically (by processors)***
    - Instructions whose registers and the functional unit are available

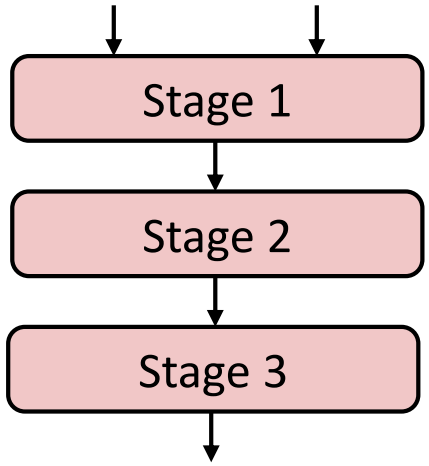


- Benefit
  - Can exploits *instruction level parallelism* that most programs have
- Most modern CPUs are superscalar
  - Intel: since Pentium (1993)

Pipelining in Functional Units

- Divide computation into several stages
- Pass partial computations from one stage to the next stage
  - Stage i can start on new computation once values passed to i+1
  - Example shows 3 mult. in 7 cycles, even though each requires 3 cycles

```
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

Haswell CPU

- Total 8 functional units
- Multiple instructions can execute in parallel

  2 load, with address computation

  1 store, with address computation

  4 integer add, 1 inter multiply

  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 Add	1	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**

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

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

Programming with AVX2

- YMM Registers (total 16 registers, 32 bytes each)
  - 32 single-byte integers

  - 16 16-bit integers

  - 8 32-bit integers

  - 8 single-precision floats

  - 4 double-precision floats

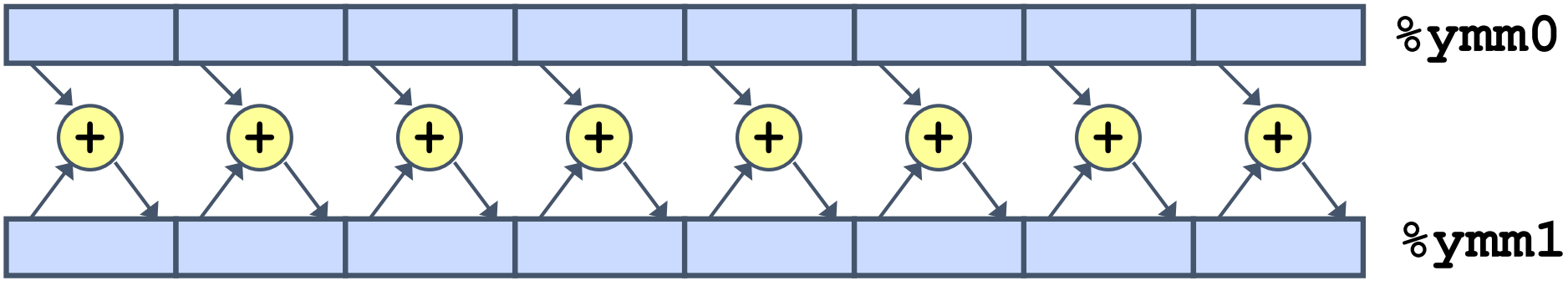
  - 1 single-precision float

  - 1 double-precision float

SIMD Operations

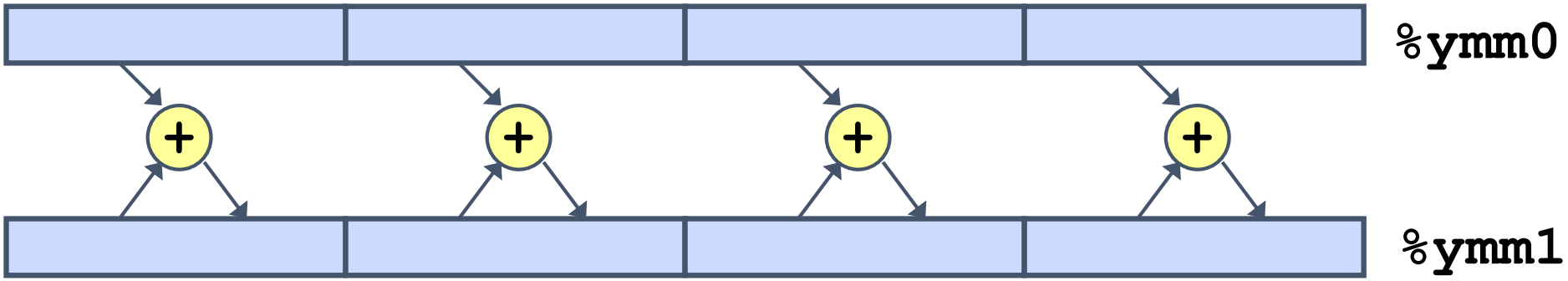
- SIMD Operations: Single Precision

```
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
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec. Throughput Bound	0.06	0.12	0.25	0.12

- Make use of AVX Instructions
  - Parallel operations on multiple data elements (i.e., SIMD)

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