

# Lecture Note 8.

## Optimization and Monitoring

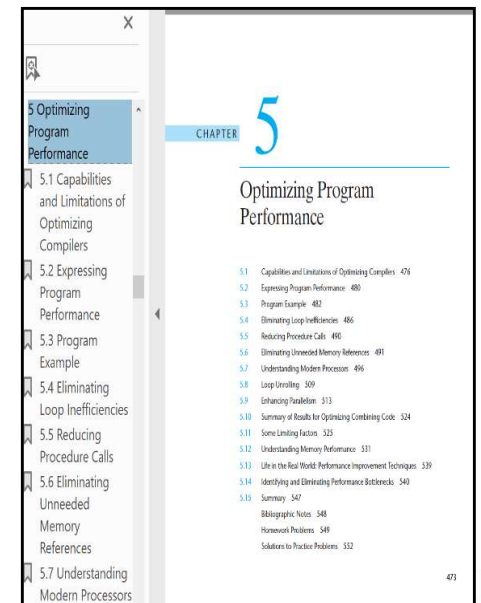
November 26, 2025

Jongmoo Choi  
Dept. of Software  
Dankook University

<http://embedded.dankook.ac.kr/~choijm>

# Objectives

- Find out the concept of optimization
- Understand how to express performance
- Discuss CPU independent optimization techniques
  - ✓ Eliminating Loop Inefficiencies
  - ✓ Reducing Procedure Calls
  - ✓ Eliminating Unneeded Memory Reference
- Discuss CPU dependent optimization techniques
  - ✓ Reducing Loop Overhead
  - ✓ Enhancing Parallelism
  - ✓ Grasping the Effect of Branch Prediction
- Memory hierarchy and performance effect
- Explore Linux performance monitoring tools
- Refer to Chapter 5 in the CSAPP



# Introduction (1/2)

## ■ Writing an optimized program

- ✓ Select the appropriate algorithms and data structures for the considered problems.
- ✓ Divide a job into portions that can be computed in parallel (e.g. Map-Reduce programming model in Google)
- ✓ Generate an efficient code

```
a = a * 4;  vs  a = a << 2;
```

```
for (i=0; i < 10; i++)  
{  
    a = 5;  
    b[i] = c[i] + a;  
}
```



# Introduction (2/2)

---

## ■ Considerations

- ✓ Tradeoff : readability vs optimization
- ✓ Deeply CPU dependent
  - Frequently analyzed with assembly language (LN 6)
  - Need to understand the CPU internals (LN 7)
- ✓ Not straightforward
  - Small changes can cause major performance difference
  - Some promising techniques prove inefficient



# Expressing Program Performance (1/2)

## ■ Example program

```
void vsum1(int n)
{
    int i;

    for (i=0; i<n; i++)
        c[i] = a[i] + b[i];
}

void vsum2(int n)
{
    int i;

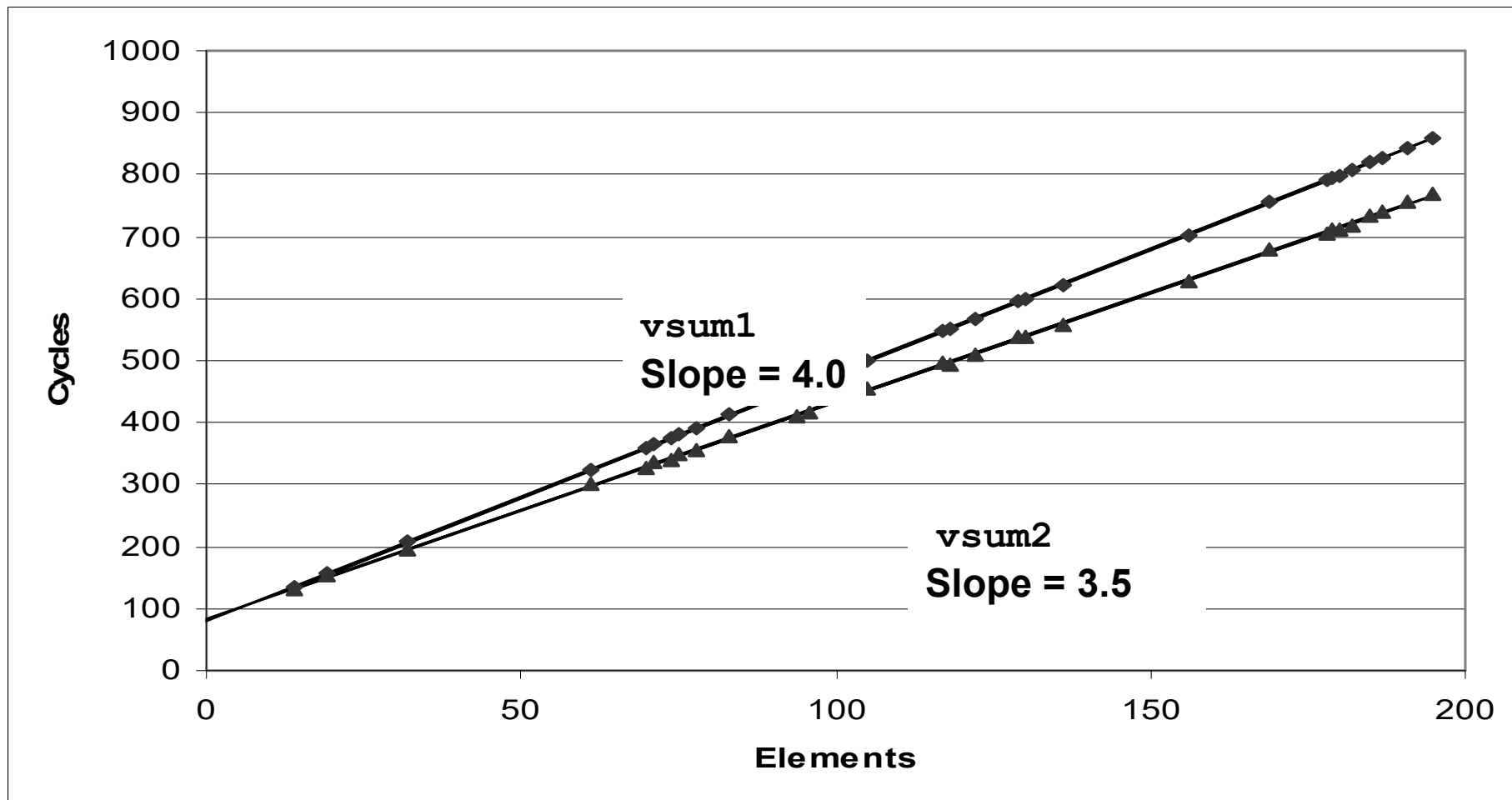
    for (i=0; i<n; i+=2) {           // loop unrolling
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] + b[i+1];
    }
}
```

(Source: CSAPP (1<sup>st</sup> Ed.), Figure 5.1)



# Expressing Program Performance (2/2)

## ■ CPE (Cycles Per Element)



✓ Using a least square fit mechanism

■  $\text{vsum1} = 80 + 4.0 \cdot n$  vs  $\text{vsum2} = 83.5 + 3.5 \cdot n$

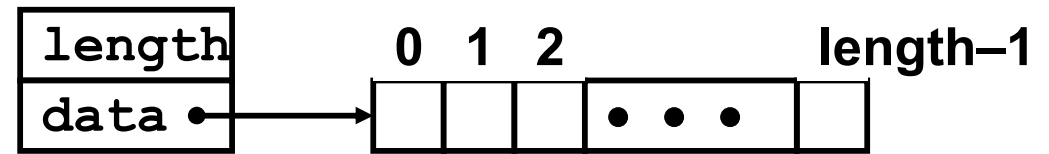


# Optimization Practice (1/11)

## ■ Basic Program

### ✓ Data type

```
typedef struct {  
    int len;  
    data_t *data;  
} vec_rec, *vec_ptr;  
// data_t ≡ int or float
```



### ✓ Procedures

- `vec_ptr new_vec(int len)`
  - Create a vector of specified length
- `int get_vec_element(vec_ptr v, int index, data_t *dest)`
  - Retrieve a vector element, store at \*dest
  - Return 0 if out of bounds, 1 if successful
- `int vec_length(vec_ptr v)`
  - Return the vector length
- `int *get_vec_start(vec_ptr v)`
  - Return pointer to start of vector data



# Optimization Practice (2/11)

## ■ Basic Program

```
vec_ptr new_vec(int len)           // from “Computer Systems: A Programmer’s Perspective”
{
    vec_ptr result = (vec_ptr) malloc (sizeof(vec_rec));
    if (!result)
        return NULL;
    result->len = len;
    if (len > 0) {
        data_t *data = (data_t *)calloc(len, sizeof(data_t));
        if (!data) {
            free((void *) result); return NULL;
        }
        result->data = data;
    }
    else
        result->data = NULL;
    return result;
}

int get_vec_element (vec_ptr v, int index, data_t *dest)
{
    if (index < 0 || index >= v->len)
        return 0;
    *dest = v->data[index];
    return 1;
}

int vec_length(vec_ptr v)
{
    return v->len;
}
```



# Optimization Practice (3/11)

## ■ combine1

```
void combine1(vec_ptr v, data_t *dest)
{
    int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OPER val;
    }
}

/*
    data_t: int, float (double)

    #define OPER +
    #define IDENT 0
    or
    #define OPER *
    #define IDENT 1
*/
```



# Optimization Practice (4/11)

## ■ Results of combine1

✓ from Pentium III (P6 microarchitecture)

| Functions | Method      | Integer |       | Float Pointing |        |
|-----------|-------------|---------|-------|----------------|--------|
|           |             | +       | ×     | +              | ×      |
| combine1  | Unoptimized | 42.06   | 41.86 | 41.44          | 160.00 |
| combine1  | O2          | 31.25   | 33.25 | 31.25          | 143.00 |



# Optimization Practice (5/11)

## ■ combine2

- ✓ eliminating loop inefficient (**code motion**)

```
void combine2(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OPER val;
    }
}
```



# Optimization Practice (6/11)

## ■ Results of combine2

- ✓ eliminating loop inefficient (**code motion**)

| Functions | Method      | Integer |       | Float Pointing |        |
|-----------|-------------|---------|-------|----------------|--------|
|           |             | +       | ×     | +              | ×      |
| combine1  | O2          | 31.25   | 33.25 | 31.25          | 143.00 |
| combine2  | Code motion | 20.66   | 21.25 | 21.25          | 135.00 |



## ■ combine3

- ✓ reducing procedure calls

```
data_t *get_vec_start(vec_ptr v)
{
    return v->data;
}

void combine3(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OPER data[i];
    }
}
```



# Optimization Practice (8/11)

## ■ Results of combine3

- ✓ reducing procedure calls

| Functions | Method             | Integer |       | Float Pointing |        |
|-----------|--------------------|---------|-------|----------------|--------|
|           |                    | +       | ×     | +              | ×      |
| combine1  | O2                 | 31.25   | 33.25 | 31.25          | 143.00 |
| combine2  | Code motion        | 20.66   | 21.25 | 21.25          | 135.00 |
| combine3  | Direct data access | 6.00    | 9.00  | 8.00           | 117.00 |

☞ may impair program modularity



## ■ combine4

- ✓ eliminating unneeded memory references

```
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t x = IDENT;

    for (i = 0; i < length; i++) {
        x = x OPER data[i];
    }
    *dest = x;
}
```



# Optimization Practice (10/11)

- Results of combine4
  - ✓ eliminating unneeded memory references

| Functions | Method                  | Integer |       | Float Pointing |        |
|-----------|-------------------------|---------|-------|----------------|--------|
|           |                         | +       | ×     | +              | ×      |
| combine1  | O2                      | 31.25   | 33.25 | 31.25          | 143.00 |
| combine2  | Code motion             | 20.66   | 21.25 | 21.25          | 135.00 |
| combine3  | Direct data access      | 6.00    | 9.00  | 8.00           | 117.00 |
| combine4  | Accumulate in temporary | 2.00    | 4.00  | 3.00           | 5.00   |



# Optimization Practice (11/11)

## ■ Results of combine4

- ✓ eliminating unneeded memory references

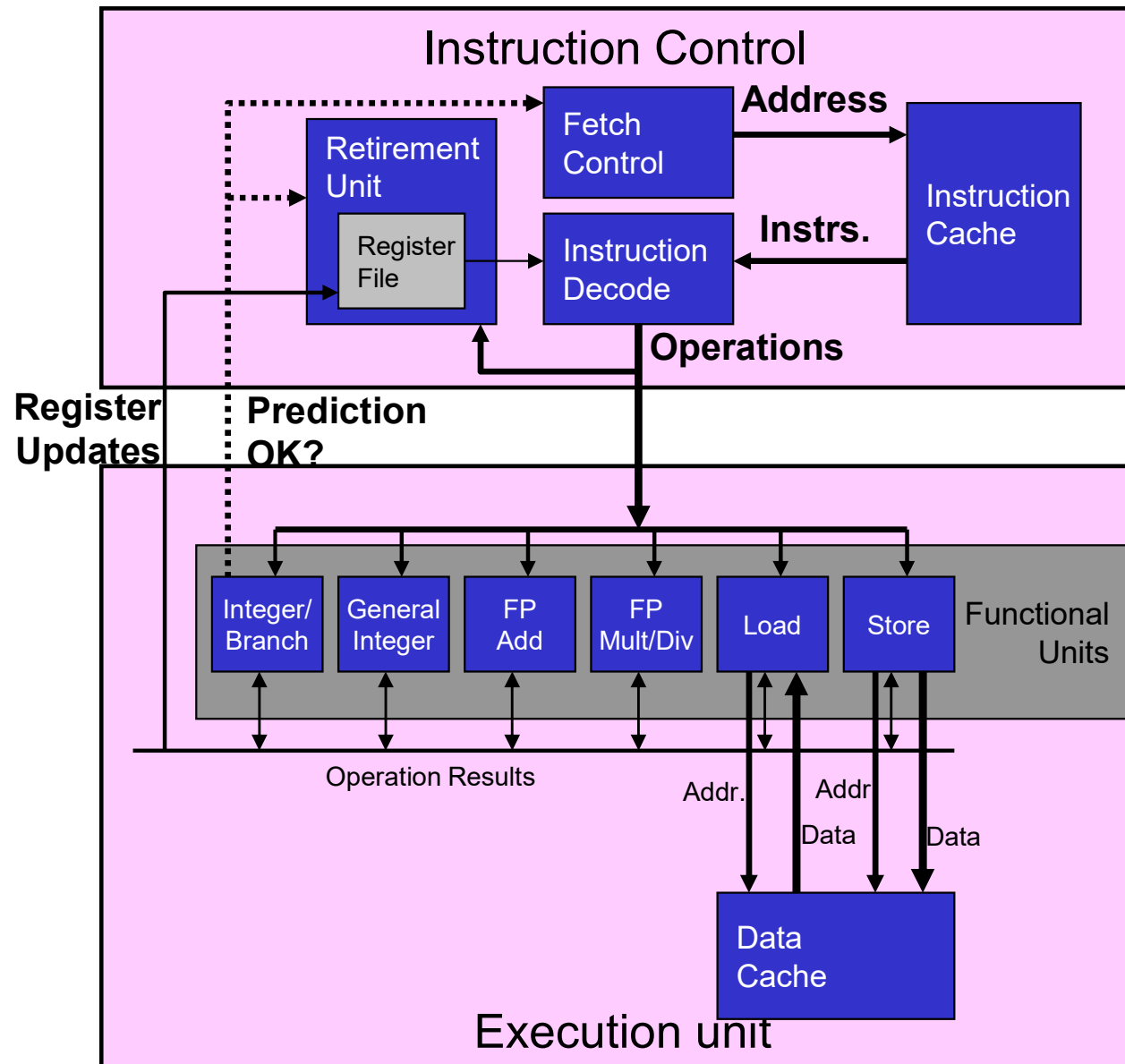
```
/* combine3
   dest in %ecx, data in %eax, i in %edx, length in %esi
*/
.L18:
    movl    (%ecx), %ebx           # read *dest
    imull   (%eax, %edx, 4), %ebx  # multiply by data[i]
    movl    %ebx, (%ecx)          # write *dest
    incl    %edx                  # i++
    cmpl    %esi, %edx            # compare
    jl      .L18                  # if <, goto loop
```

```
/* combine4
   x in %ecx, data in %eax, i in %edx, length in %esi
*/
.L24:
    imull   (%eax, %edx, 4), %ecx  # multiply by data[i]
    incl    %edx
    cmpl    %esi, %edx              4 CPE ??
    jl      .L24
```



# Understanding Modern Processor (1/12)

## ■ Intel P6 microarchitecture



# Understanding Modern Processor (2/12)

## ■ Overall features

### ✓ two main units

- instruction control: fetch and decode (generates operations)
- execution unit: execute generated operations

```
addl    %eax, %edx
```

☞ **one instruction, one operation**

```
addl    %eax, 4(%edx)
```

☞ **one instruction, three operations**

### ✓ functional units

- Integer/Branch: simple integer (add, test, compare, logical) and branch
- General Integer: all integer operations, including multiplication
- Floating-Point Add: floating-point operations (addition, format, conversion)
- Floating-Point Multiplication/Division: floating-point multiplication and division
- Load: read data from memory
- Store: write data to memory



# Understanding Modern Processor (3/12)

---

## ■ Overall features

- ✓ Superscalar
  - perform multiple operations on every clock cycle
- ✓ Out-of-order
  - the order in which instructions execute need not correspond to their ordering in the assembly program
- ✓ Branch prediction
  - guess whether or not a branch will be taken
- ✓ Speculative execution
  - commit execution when the prediction is proven as correct.
  - otherwise, discard the results of the incorrect prediction (retirement unit)
- ✓ Register renaming
  - maps the logical registers (used by programmers) to a larger pool of physical registers to eliminate false data dependencies
  - tagging so that values passed directly from one functional unit to another



# Understanding Modern Processor (4/12)

## ■ Functional unit performance

- ✓ Latency: total number of cycle for a single operation
- ✓ Issue time: number of cycles between successive operations

| Operation               | Latency | Issue time |
|-------------------------|---------|------------|
| Integer add             | 1       | 1          |
| Integer Multiply        | 4       | 1          |
| Integer Divide          | 36      | 36         |
| Floating-point add      | 3       | 1          |
| Floating-point multiply | 5       | 2          |
| Floating-point divide   | 38      | 38         |
| Load (cache hit)        | 3       | 1          |
| Store (cache hit)       | 3       | 1          |

☞ Integer Multiply, FP add, load, ... can be pipelined

☞ The latency and issue time are improved in Intel Core i7 (eg. multiply latency = 3, add issue time = 0.33 (3 pipeline engines), ...)



# Understanding Modern Processor (5/12)

## ■ instruction and operations

```
/* combine4
   data in %eax, x in %ecx, i in %edx, length in %esi
*/
.L24:
    imull    (%eax,%edx, 4), %ecx    # multiply by data[i]
    incl     %edx                   # i++
    cmpl     %esi, %edx              # compare
    jl       .L24                   # if <, goto loop
```

| Assembly Instructions                                                                               | Execution unit operations                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .L24:<br>imull    (%eax,%edx, 4), %ecx<br><br>incl     %edx<br>cmpl     %esi, %edx<br>jl       .L24 | load (%eax, %edx.0, 4)    → t.1<br>imull t.1, %ecx.0        → %ecx.1<br>incl %edx.0              → %edx.1<br>cmpl %esi, %edx.1        → cc.1<br>jl-taken cc.1 |

👉 **branch prediction and register renaming**

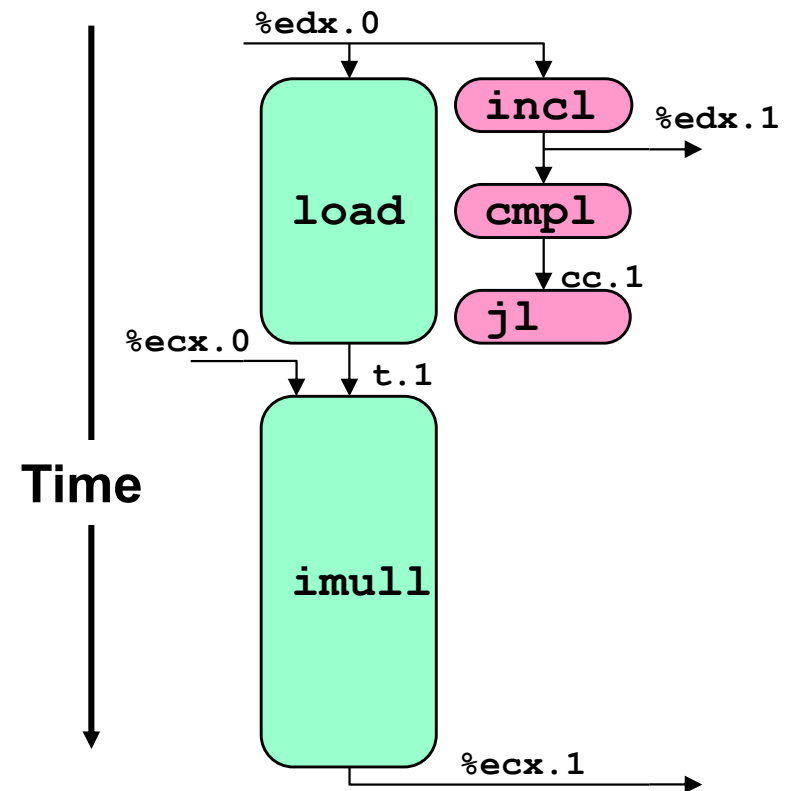


# Understanding Modern Processor (6/12)

## ■ timing in execution unit: multiplication case

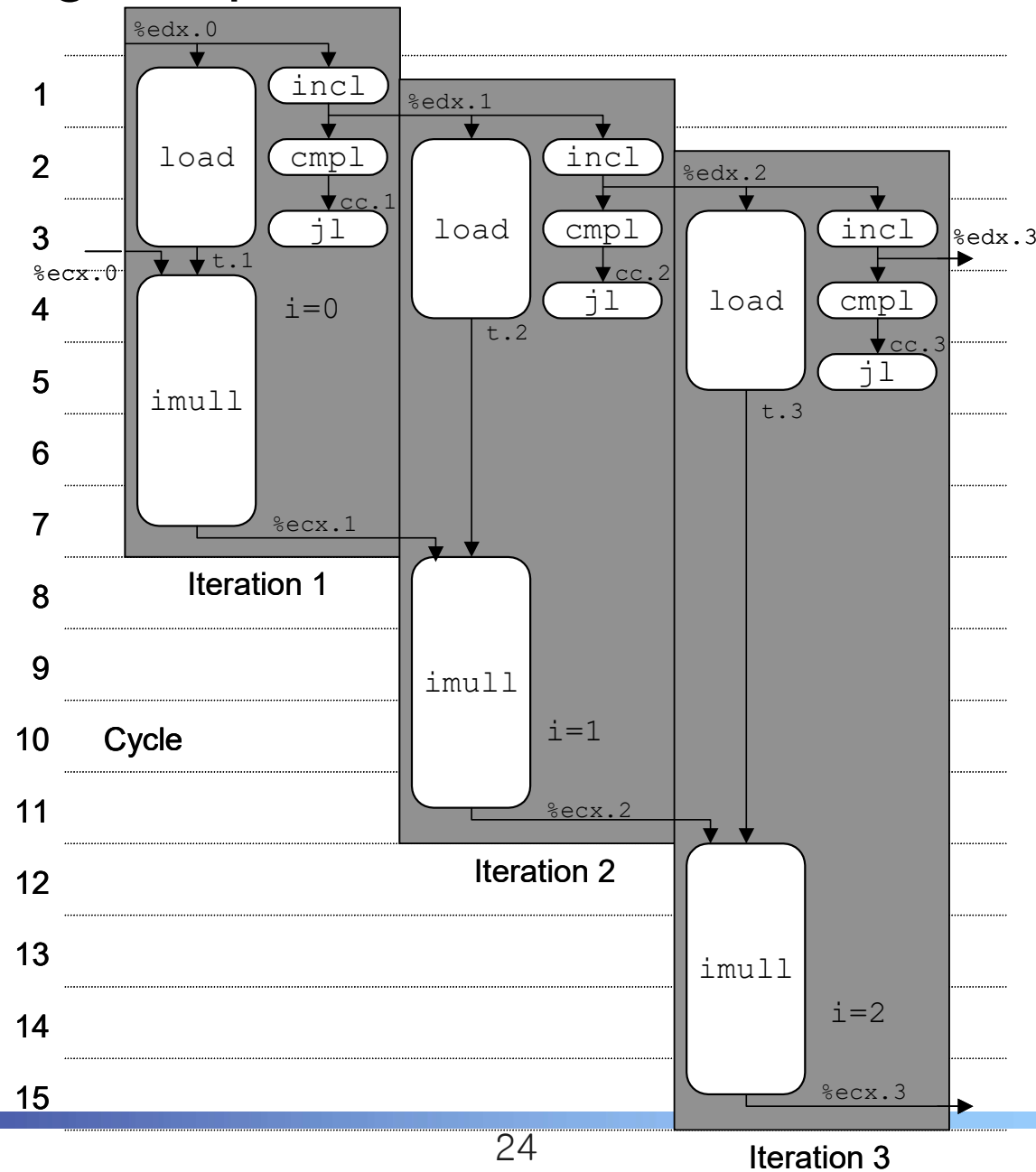
### Execution unit operations

|                        |          |
|------------------------|----------|
| load (%eax, %edx.0, 4) | → t.1    |
| imull t.1, %ecx.0      | → %ecx.1 |
| incl %edx.0            | → %edx.1 |
| cmpl %esi, %edx.1      | → cc.1   |
| jl-taken cc.1          |          |



# Understanding Modern Processor (7/12)

## ■ Scheduling of Operations with Unlimited Resources

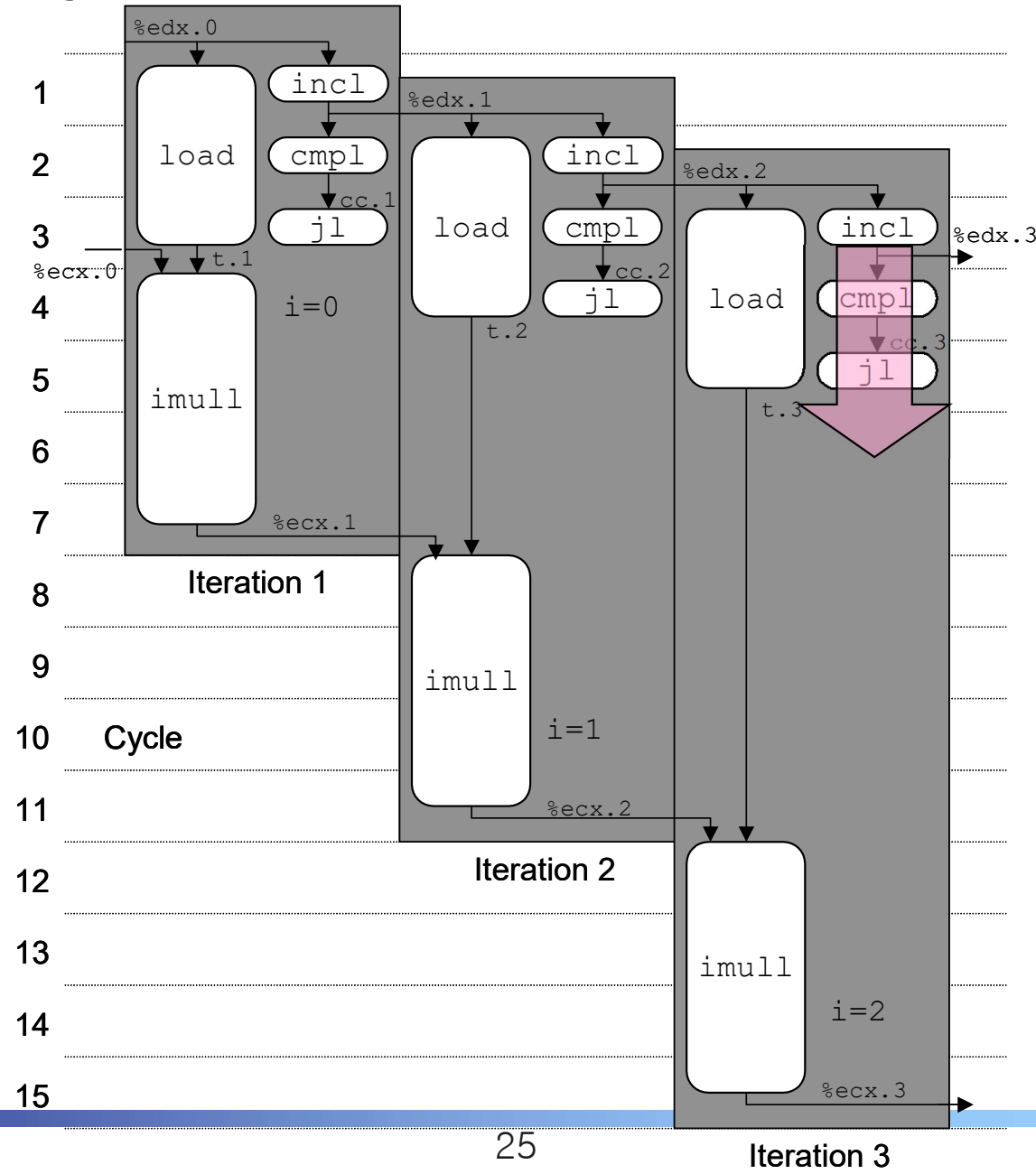


- 👉 superscalar
- 👉 pipeline
- 👉 data dependency (flow analysis)
- 👉 out-of order execution
- 👉 branch prediction
- 👉 CPE = 4.0



# Understanding Modern Processor (8/12)

## ■ Scheduling of Operations with Resources Constraints



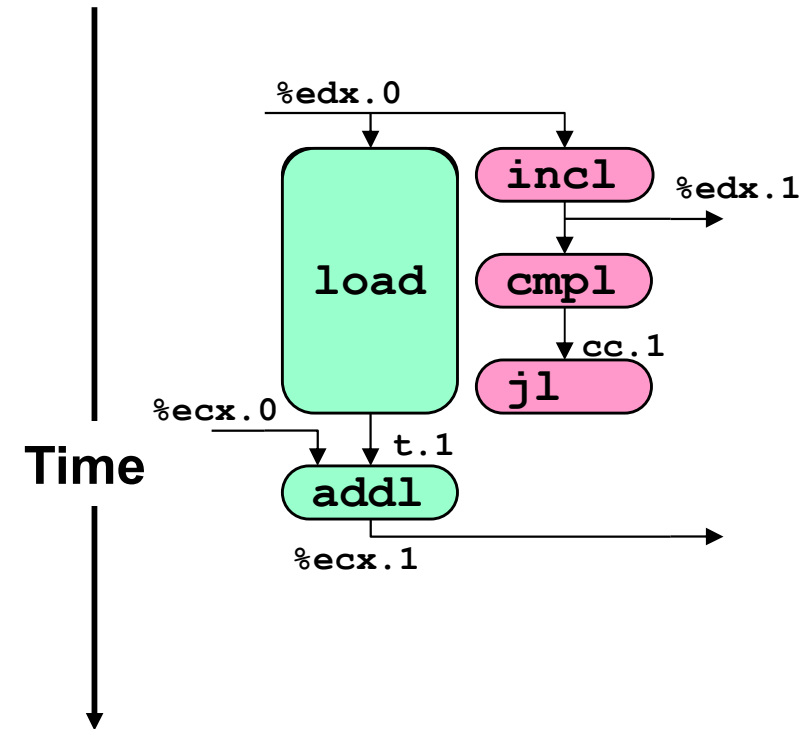
👉 CPE = 4.0



# Understanding Modern Processor (9/12)

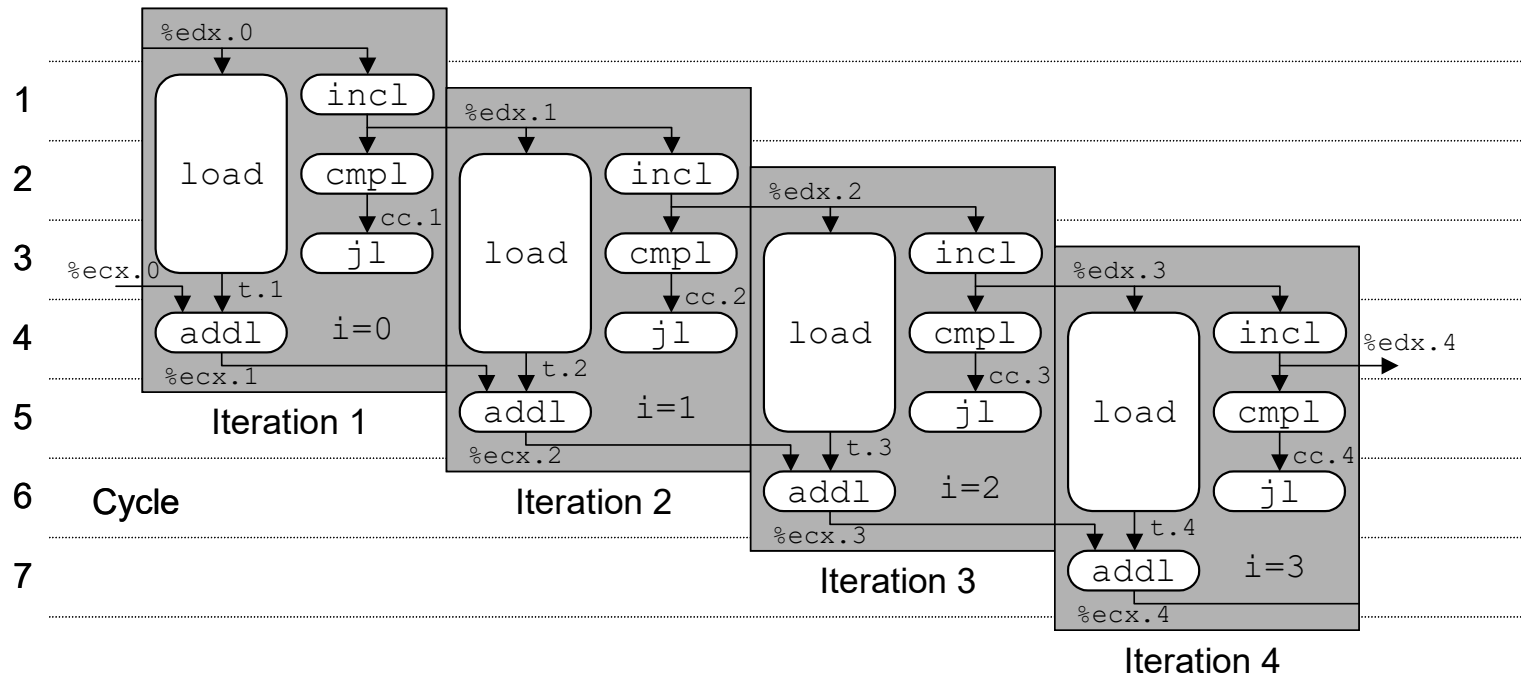
## ■ timing in execution unit: add case

| Execution unit operations |          |
|---------------------------|----------|
| load (%eax, %edx.0, 4)    | → t.1    |
| addl t.1, %ecx.0          | → %ecx.1 |
| incl %edx.0               | → %edx.1 |
| cmpl %esi, %edx.1         | → cc.1   |
| jl-taken cc.1             |          |



# Understanding Modern Processor (10/12)

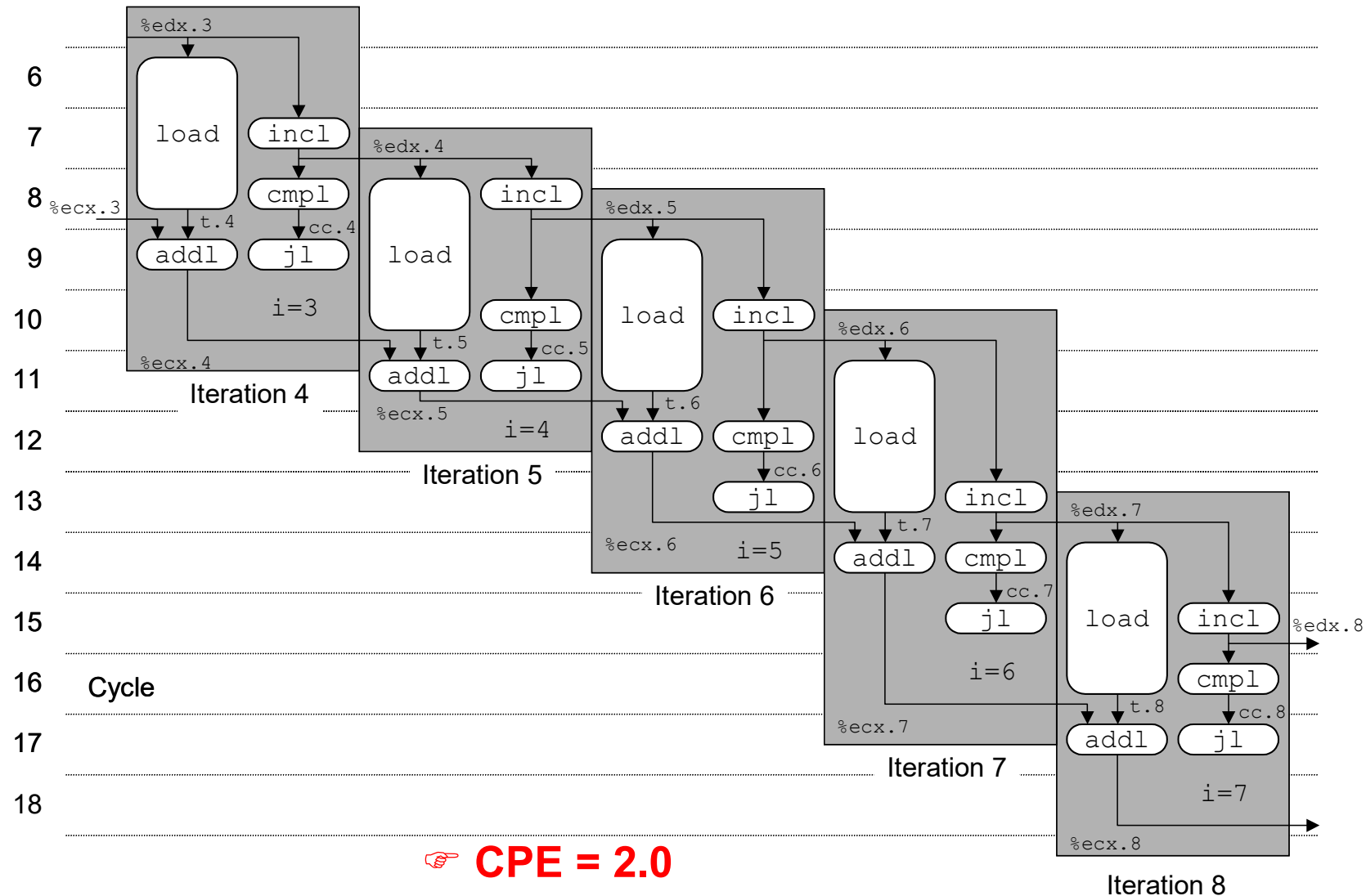
## ■ Scheduling of Operations with Unlimited Resources



👉 CPE = 1.0

# Understanding Modern Processor (11/12)

## ■ Scheduling of Operations with Resources Constraints



# Understanding Modern Processor (12/12)

- Results of combine4 revisit
  - ✓ eliminating unneeded memory references

| Functions | Method                  | Integer |       | Float Pointing |        |
|-----------|-------------------------|---------|-------|----------------|--------|
|           |                         | +       | ×     | +              | ×      |
| combine1  | O2                      | 31.25   | 33.25 | 31.25          | 143.00 |
| combine2  | Code motion             | 20.66   | 21.25 | 21.25          | 135.00 |
| combine3  | Direct data access      | 6.00    | 9.00  | 8.00           | 117.00 |
| combine4  | Accumulate in temporary | 2.00    | 4.00  | 3.00           | 5.00   |

- 👉 Integer add: can be 1 but becoming 2.0 due to the resource constraints
- 👉 Integer multiplication: there is an idle room due to the dependency between iterations



# Enhanced Optimization (1/9)

## ■ Loop unrolling

- ✓ exploit superscalar and decrease the total # of iterations (the frequency of loop maintenance instructions)

```
void combine5(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t x = IDENT;
    int limit = length - 2;

    for (i = 0; i < limit; i += 3) {
        x = x OPER data[i] OPER data[i+1] OPER data[i+2];
    }

    for (; i < length; i++) {
        x = x OPER data[i];
    }
    *dest = x;
}
```



# Enhanced Optimization (2/9)

## ■ Loop unrolling

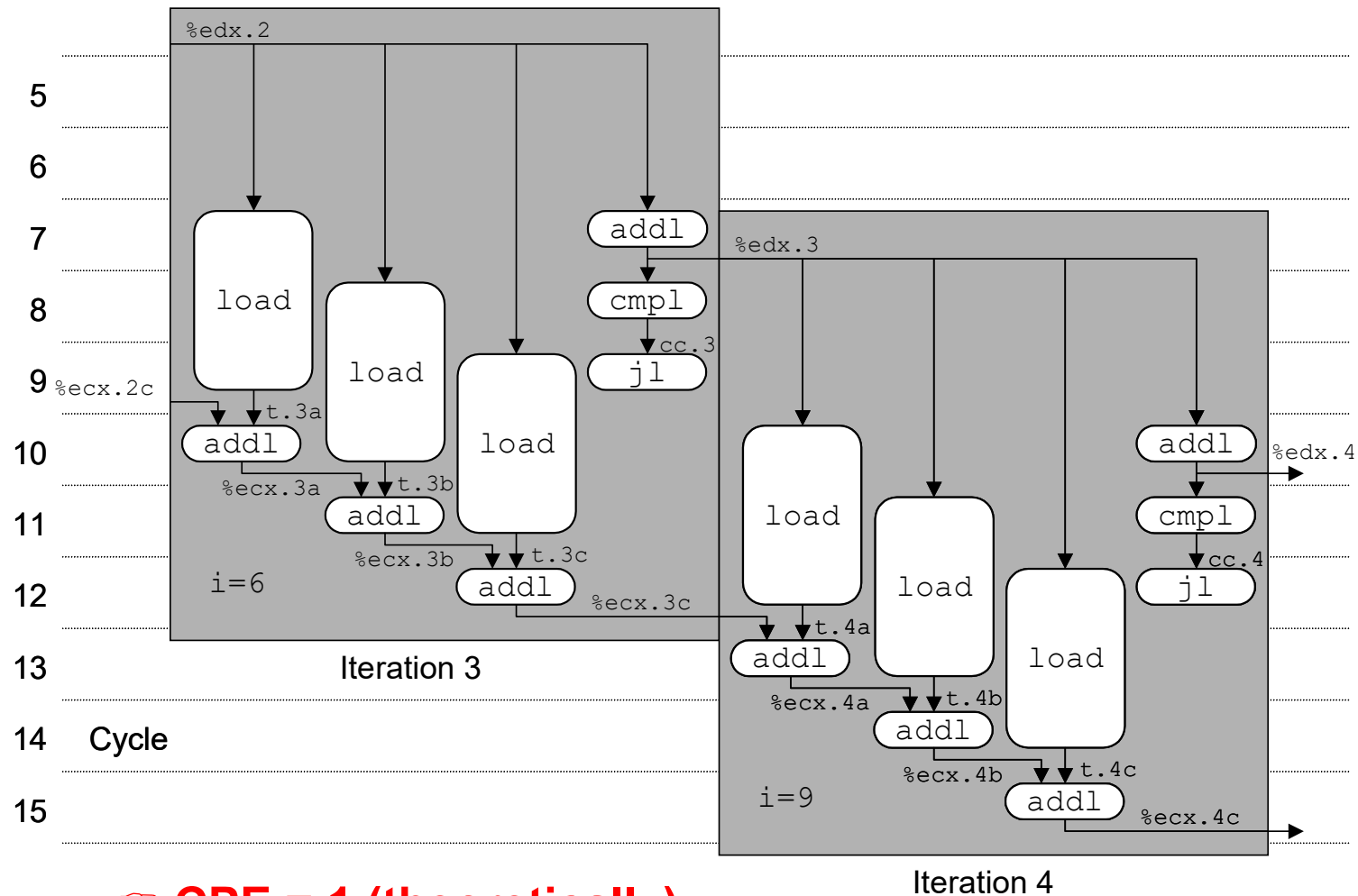
- ✓ instruction and operations

| Assembly Instructions                                                                                                                                              | Execution unit operations                                                                                                                                                                                                                               |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>.L49:     addl (%eax,%edx, 4), %ecx      addl 4(%eax,%edx, 4), %ecx      addl 8(%eax,%edx, 4), %ecx      addl %edx, 3     cmpl %esi, %edx     jl   .L49</pre> | <pre>load (%eax,%edx.0,4) → t.1a addl t.1a, %ecx.0c → %ecx.1a load 4(%eax,%edx.0,4) → t.1b addl t.1b, %ecx.1a → %ecx.1b load 8(%eax,%edx.0,4) → t.1c addl t.1c, %ecx.1b → %ecx.1c addl \$3,%edx.0 → %edx.1 cmpl %esi, %edx.1 → cc.1 jl-taken cc.1</pre> |



# Enhanced Optimization (3/9)

- Loop unrolling
- ✓ scheduling of operations



☞ **CPE = 1 (theoretically)**

☞ **CPE = 1.33 (empirically, not account for the reason, may be register spilling or resource contention)**



# Enhanced Optimization (4/9)

## ■ Loop unrolling

- ✓ degree of unrolling

| Vector length | Degree of unrolling |      |      |      |      |      |
|---------------|---------------------|------|------|------|------|------|
|               | 1                   | 2    | 3    | 4    | 8    | 16   |
| CPE           | 2.00                | 1.50 | 1.33 | 1.50 | 1.25 | 1.06 |

- not monotonic
- depends on various factors such as scheduling, constraints, registers, ...

👉 **loop unrolling increases the code size (and reduce code readability)**



- Loop splitting (parallelism)
  - ✓ unrolling by 2 and 2-way parallelism

```
void combine6(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    int limit = length - 1;
    data_t *data = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;

    for (i = 0; i < limit; i+=2) {
        x0 = x0 OPER data[i];
        x1 = x1 OPER data[i+1];
    }
    for (; i < length; i++) {
        x0 = x0 OPER data[i];
    }
    *dest = x0 OPER x1;
}
```



# Enhanced Optimization (6/9)

## ■ Loop splitting

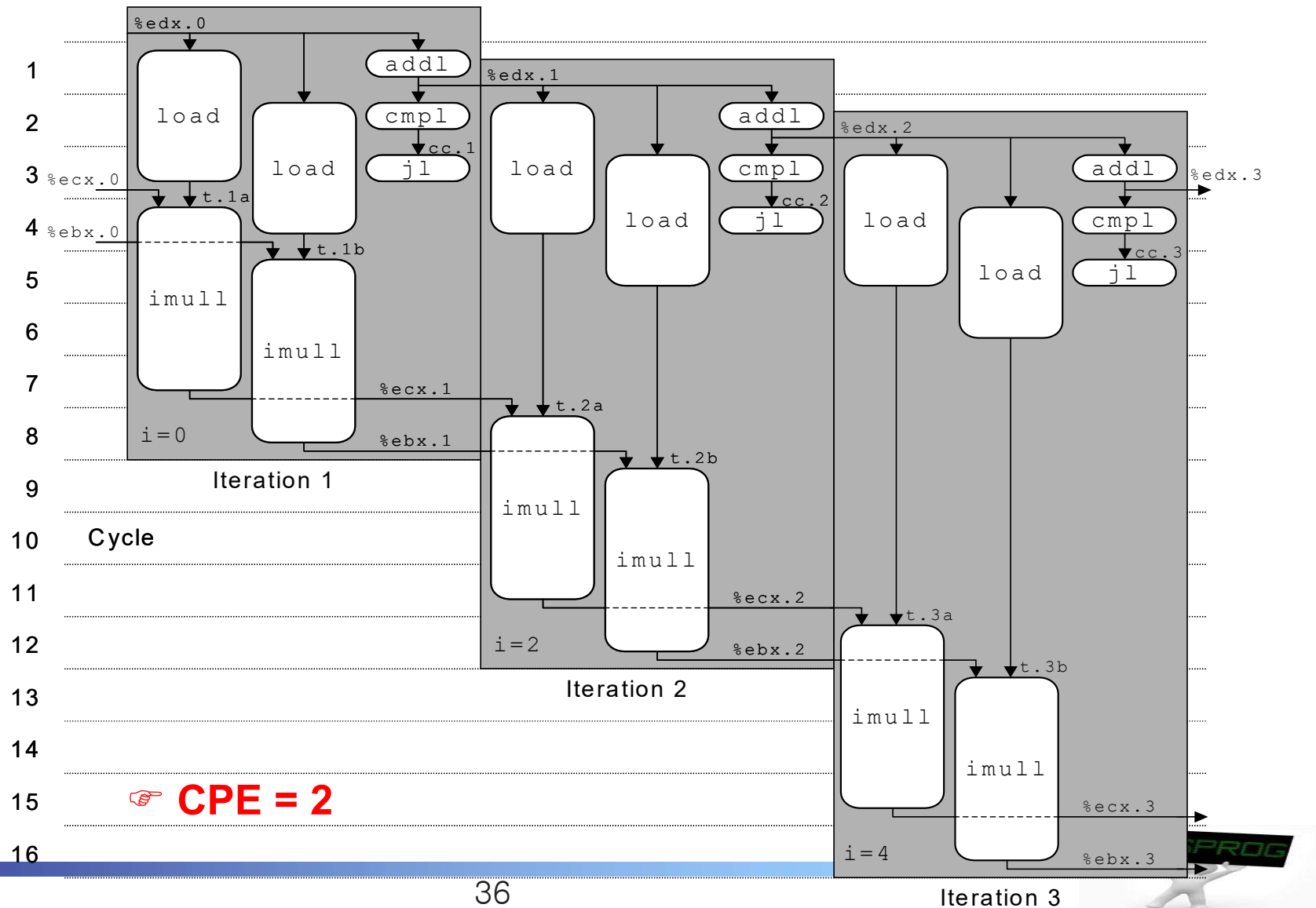
- ✓ instruction and operations

| Assembly Instructions                                                                                                                                                                                    | Execution unit operations                                                                                                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>.L151:</code><br><code>imull (%eax,%edx, 4), %ecx</code><br><br><code>imull 4(%eax,%edx, 4), %ebx</code><br><br><code>addl %edx, 2</code><br><code>cmpl %esi, %edx</code><br><code>jl .L151</code> | <code>load (%eax,%edx.0,4) → t.1a</code><br><code>imull t.1a, %ecx.0 → %ecx.1</code><br><code>load 4(%eax,%edx.0,4) → t.1b</code><br><code>imull t.1b, %ebx.0 → %ebx.1</code><br><code>iaddl \$2,%edx.0 → %edx.1</code><br><code>cmpl %esi, %edx.1 → cc.1</code><br><code>jl-taken cc.1</code> |



# Enhanced Optimization (7/9)

- Loop splitting
  - ✓ scheduling of operations



# Enhanced Optimization (8/9)

## ■ Loop unrolling and splitting

| Functions | Method                         | Integer |      | Float Pointing |      |
|-----------|--------------------------------|---------|------|----------------|------|
|           |                                | +       | ×    | +              | ×    |
| combine4  | Accumulate in temporary        | 2.00    | 4.00 | 3.00           | 5.00 |
| combine5  | Unroll * 2                     | 1.50    | 4.00 | 3.00           | 5.00 |
| combine6  | Unroll * 2,<br>Parallelism * 2 | 1.50    | 2.00 | 2.00           | 2.50 |



# Enhanced Optimization (9/9)

## ■ Increasing Parallelism

| Method                      | Integer |      | Float Pointing |      |
|-----------------------------|---------|------|----------------|------|
|                             | +       | ×    | +              | ×    |
| Unroll * 2                  | 1.50    | 4.00 | 3.00           | 5.00 |
| Unroll * 2, Parallelism * 2 | 1.50    | 2.00 | 2.00           | 2.50 |
| Unroll * 4                  | 1.50    | 4.00 | 3.00           | 5.00 |
| Unroll * 4, Parallelism * 2 | 1.50    | 2.00 | 1.50           | 2.50 |
| Unroll * 8                  | 1.25    | 4.00 | 3.00           | 5.00 |
| Unroll * 8, Parallelism * 2 | 1.25    | 2.00 | 1.50           | 2.50 |
| Unroll * 8, Parallelism * 4 | 1.25    | 1.25 | 1.61           | 2.00 |
| Unroll * 8, Parallelism * 8 | 1.75    | 1.87 | 1.87           | 2.07 |
| Unroll * 9, Parallelism * 3 | 1.22    | 1.33 | 1.66           | 2.00 |

👉 **Non-linear → one possible reason is the register spill**



# Branch Misprediction Penalties (1/1)

## ■ Branch Prediction

```
int absval(int val)
{
    return (val < 0) ? - val : val;
}
```

```
absval:
    pushl    %ebp
    movl     %esp, %ebp
    movl     8(%ebp), %eax
    testl    %eax, %eax
    jge      .L3
    negl     %eax
.L3:
    movl     %ebp, %esp
    popl     %ebp
    ret
```

- ✓ if correct, takes 13 cycles
- ✓ otherwise, takes 27 cycles (the penalty is 14 cycles)
- ✓ regular patterns: 13.01 ~ 13.41 cycles
- ✓ random patterns: average 20.32 cycles (13 ~ 27 cycles)



# Memory Hierarchy and Performance (1/4)

## ■ Memory hierarchy

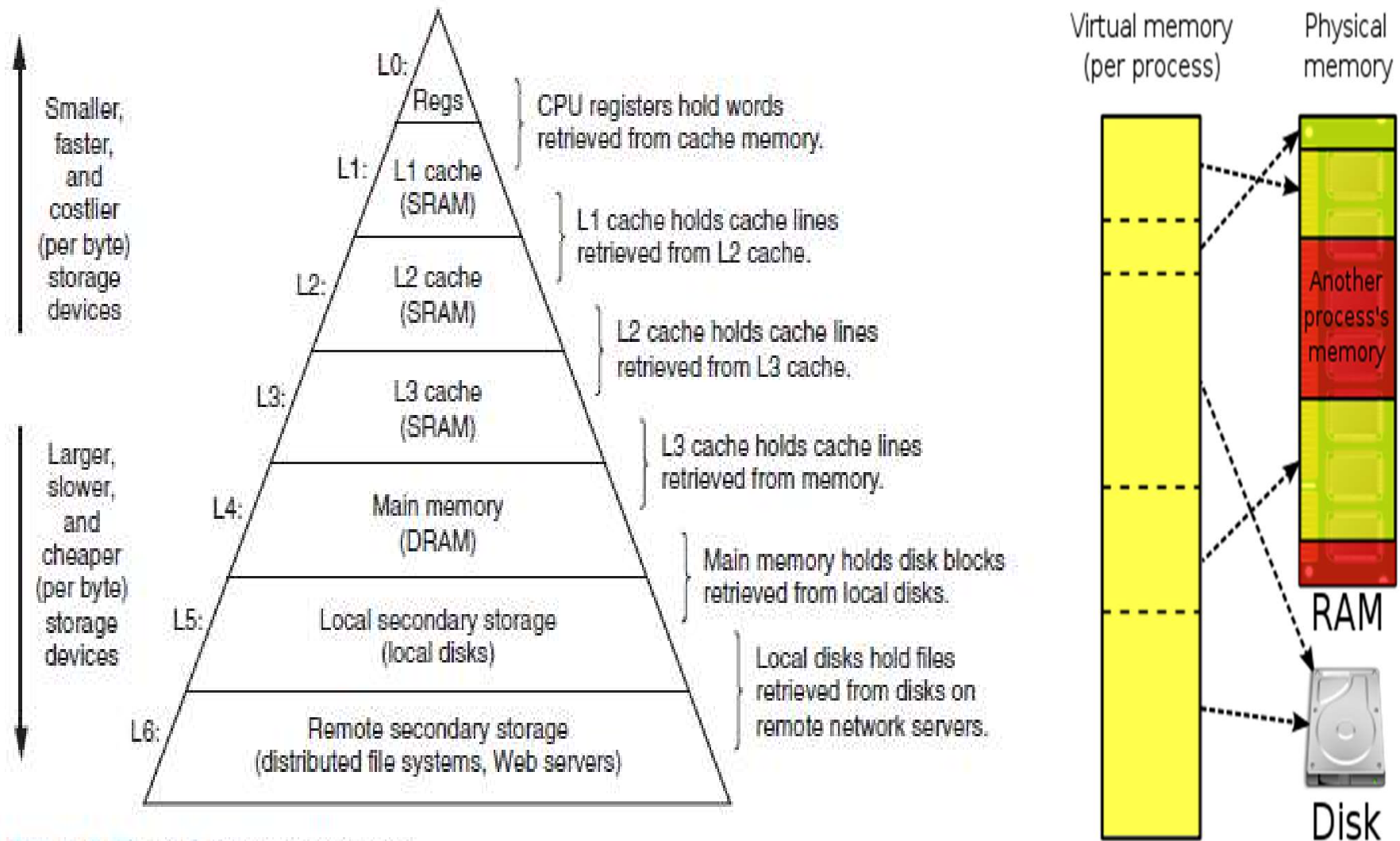


Figure 6.23 The memory hierarchy.

(Source: CSAPP)



# Memory Hierarchy and Performance (2/4)

## ■ Cache effect

- ✓ Consider DRAM and Disk case (DRAM as a Disk's cache)
- ✓ Goal: Maximize **cache hit** (or minimize cache miss)
- ✓ Model

### ■ Average memory access time (AMAT)

- $AMAT = (P_{hit} \times T_M) + (P_{miss} \times T_D)$  (or  $P_{miss} \times (T_M + T_D)$ )
- Where
  - $T_M$ : memory access latency
  - $T_D$ : disk access latency
  - $P_{hit}$ : probability of finding data in the cache ( $P_{miss} = 1 - P_{hit}$ )

### ■ Example

- Assume that  $T_M = 100\text{ns}$ ,  $T_D = 10\text{ms}$  (10,000,000ns)
- $P_{hit} = 50\% \rightarrow AMAT = 0.5 \times 100 + 0.5 \times 10,000,000 = 5,000,050 = 5\text{ms}$
- $P_{hit} = 90\% \rightarrow AMAT = 0.9 \times 100 + 0.1 \times 10,000,000 = 1,000,090 = 1\text{ms}$
- $P_{hit} = 99\% \rightarrow AMAT = 0.99 \times 100 + 0.01 \times 10,000,000 = 100,099 = 0.1\text{ms}$

### ■ Hit ratio is quite important

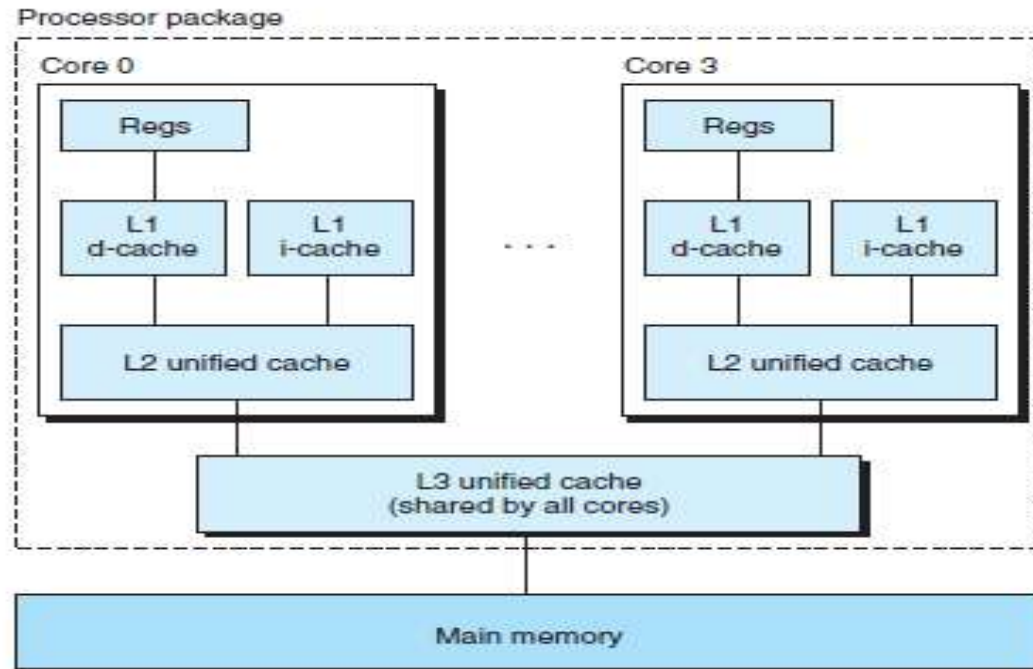
- Expected hit ratio =  $S_M/S_D$  if an access pattern is the uniform distribution
- Remember **locality** which makes it feasible to obtain high hit ratio



# Memory Hierarchy and Performance (3/4)

## ■ How about CPU cache?

Figure 6.40  
Intel Core i7 cache  
hierarchy.



| Cache type       | Access time (cycles) | Cache size ( $C$ ) | Assoc. ( $E$ ) | Block size ( $B$ ) | Sets ( $S$ ) |
|------------------|----------------------|--------------------|----------------|--------------------|--------------|
| L1 i-cache       | 4                    | 32 KB              | 8              | 64 B               | 64           |
| L1 d-cache       | 4                    | 32 KB              | 8              | 64 B               | 64           |
| L2 unified cache | 11                   | 256 KB             | 8              | 64 B               | 512          |
| L3 unified cache | 30–40                | 8 MB               | 16             | 64 B               | 8192         |

Figure 6.41 Characteristics of the Intel Core i7 cache hierarchy.

# Memory Hierarchy and Performance (4/4)

## Cache performance

- ✓ Using matrix multiplication

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

(a) Version *ijk*

```
code/mem/matmult/mm.c
1 for (i = 0; i < n; i++)
2   for (j = 0; j < n; j++) {
3     sum = 0.0;
4     for (k = 0; k < n; k++)
5       sum += A[i][k]*B[k][j];
6     C[i][j] += sum;
7   }
```

(b) Version *jik*

```
code/mem/matmult/mm.c
1 for (j = 0; j < n; j++)
2   for (i = 0; i < n; i++) {
3     sum = 0.0;
4     for (k = 0; k < n; k++)
5       sum += A[i][k]*B[k][j];
6     C[i][j] += sum;
7   }
```

(c) Version *jki*

```
code/mem/matmult/mm.c
1 for (j = 0; j < n; j++)
2   for (k = 0; k < n; k++) {
3     r = B[k][j];
4     for (i = 0; i < n; i++)
5       C[i][j] += A[i][k]*r;
6   }
```

(d) Version *kji*

```
code/mem/matmult/mm.c
1 for (k = 0; k < n; k++)
2   for (j = 0; j < n; j++) {
3     r = B[k][j];
4     for (i = 0; i < n; i++)
5       C[i][j] += A[i][k]*r;
6   }
```

(e) Version *kij*

```
code/mem/matmult/mm.c
1 for (k = 0; k < n; k++)
2   for (i = 0; i < n; i++) {
3     r = A[i][k];
4     for (j = 0; j < n; j++)
5       C[i][j] += r*B[k][j];
6   }
```

(f) Version *ikj*

```
code/mem/matmult/mm.c
1 for (i = 0; i < n; i++)
2   for (k = 0; k < n; k++) {
3     r = A[i][k];
4     for (j = 0; j < n; j++)
5       C[i][j] += r*B[k][j];
6   }
```

| Matrix multiply version (class) | Loads per iter. | Stores per iter. | A misses per iter. | B misses per iter. | C misses per iter. | Total misses per iter. |
|---------------------------------|-----------------|------------------|--------------------|--------------------|--------------------|------------------------|
| <i>ijk</i> & <i>jik</i> (AB)    | 2               | 0                | 0.25               | 1.00               | 0.00               | 1.25                   |
| <i>jki</i> & <i>kji</i> (AC)    | 2               | 1                | 1.00               | 0.00               | 1.00               | 2.00                   |
| <i>kij</i> & <i>ikj</i> (BC)    | 2               | 1                | 0.00               | 0.25               | 0.25               | 0.50                   |

Figure 6.47 Analysis of matrix multiply inner loops. The six versions partition into three equivalence classes, denoted by the pair of arrays that are accessed in the inner loop.

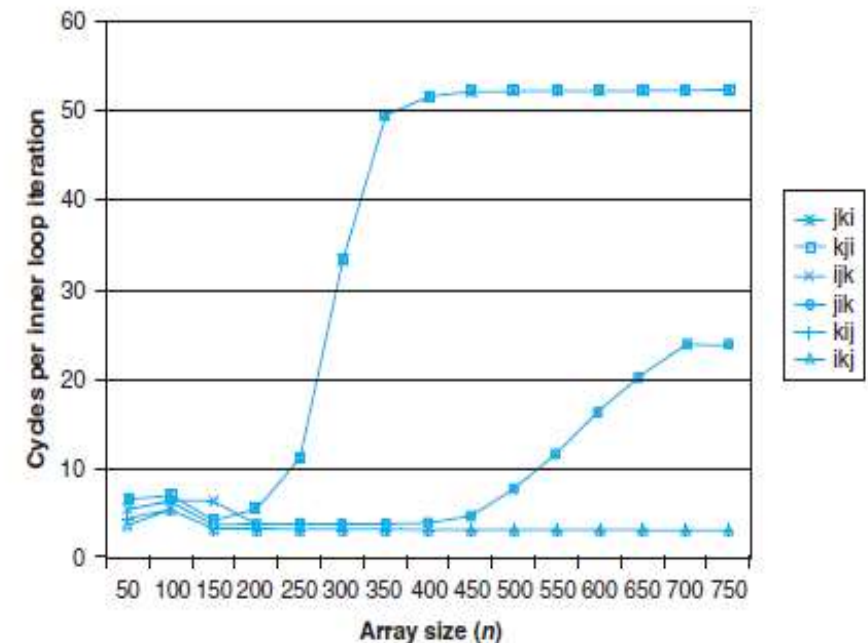
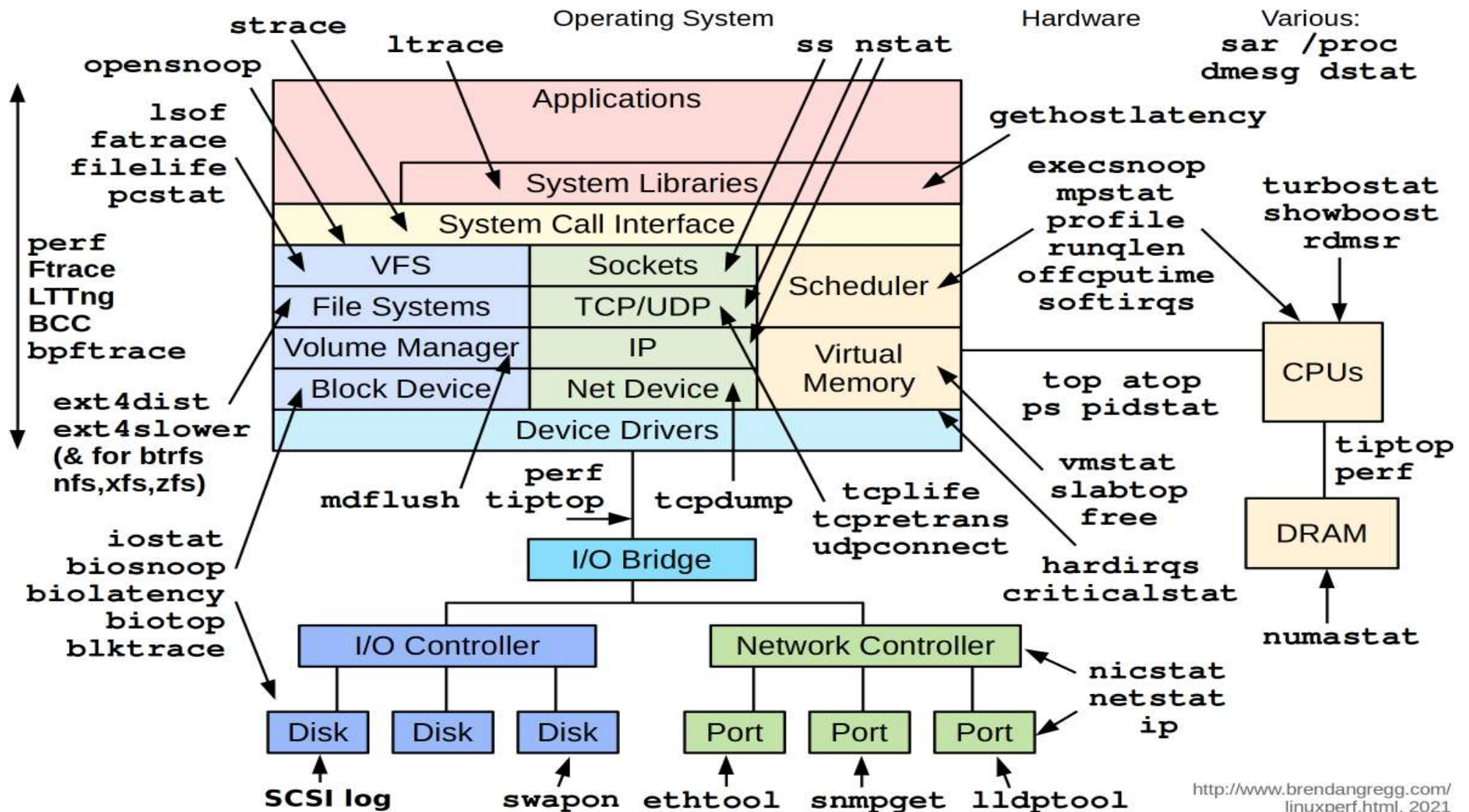


Figure 6.46 Six versions of matrix multiply. Each version is uniquely identified by the ordering of its loops.

- From <http://www.brendangregg.com/linuxperf.html>



# Linux Performance Monitoring Tools (2/11)

---

## ■ Basic

- ✓ ps: show information about active processes
- ✓ time: run programs and summarize timing statistics (user/system)
- ✓ gettimeofday: obtain the current time, expressed as seconds and microseconds since the Epoch (Jan. 1, 1970)
- ✓ uptime: show how long the system has been running and the system load averages for the past 1, 5 and 15 minutes
- ✓ top: display system summary information as well as a list of processes
- ✓ vmstat: report virtual memory statistics
- ✓ iostat: report input/output device loading
- ✓ mpstat: report the utilization of multiple processors
- ✓ pcstat: report page cache stats
- ✓ pidstat: report information for monitoring individual tasks
- ✓ dstat: report all aspects of system performance



# Linux Performance Monitoring Tools (3/11)

---

## ■ Intermediate

- ✓ strace: trace system calls
- ✓ blktrace: trace I/Os, a block layer IO tracing mechanism
- ✓ tcpdump: print out the contents of packets on a network interface
- ✓ netstat: show network status information.
- ✓ nicstat: show statistics for all network interface cards (NICs)
- ✓ swapon: to specify devices on which paging and swapping are to take place
- ✓ lsof: list open files
- ✓ sar: system activity reporter

## ■ Advanced

- ✓ perf\_event: performance events such as CPU cache hit, # of memory access, cycle per instruction, ...
- ✓ ltrace: a debugging utility in Linux, used to display the calls a user space application makes to shared libraries
- ✓ ftrace: function tracer, a tracing framework for the Linux kernel
- ✓ lxc-monitor: monitor the state of Linux containers
- ✓ ...



# Linux Performance Monitoring Tools (4/11)

## ■ gettimeofday()

```
/* For measuring elapsed time of the target function by choijm@dankook.ac.kr */
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>

void target_job(int loopsize)
{
    int i;
    for (i=0; i < loopsize; i++);
}

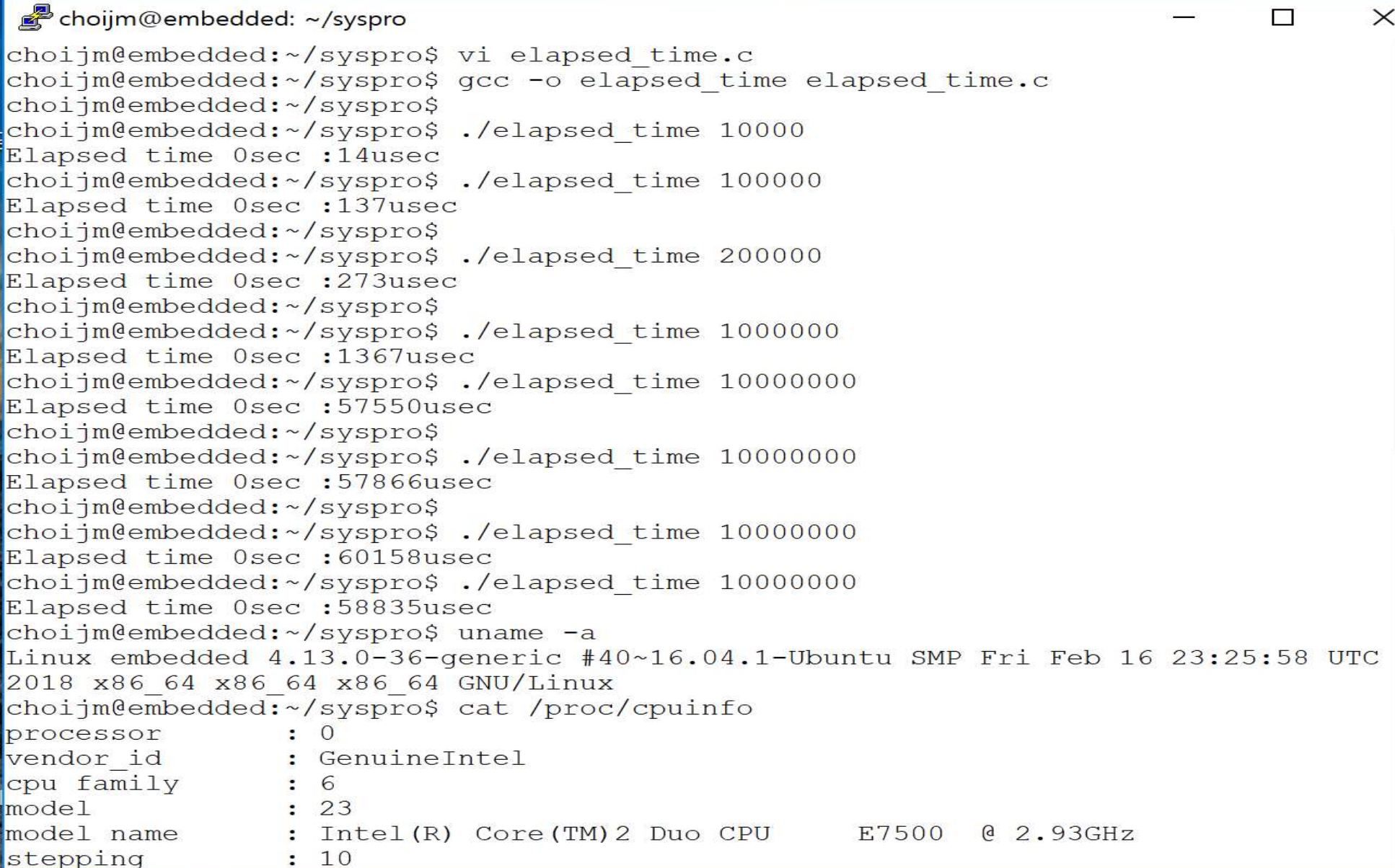
int main(int argc, char *argv[])
{
    int i, loop = 0;  struct timeval stime, etime, gap;

    if (argc == 2) {
        loop = atoi(argv[1]);
        gettimeofday(&stime, NULL);
        target_job(loop);
        gettimeofday(&etime, NULL);
        gap.tv_sec = etime.tv_sec - stime.tv_sec; gap.tv_usec = etime.tv_usec - stime.tv_usec;
        if (gap.tv_usec < 0) {
            gap.tv_sec = gap.tv_sec - 1; gap.tv_usec = gap.tv_usec + 1000000;
        }
        printf("Elapsed time %ldsec :%ldusec\n", gap.tv_sec, gap.tv_usec);
    }
    else
        printf("Usage: command loop_size\n");
}
```



# Linux Performance Monitoring Tools (5/11)

## ■ gettimeofday()



```
choijm@embedded: ~/syspro
choijm@embedded:~/syspro$ vi elapsed_time.c
choijm@embedded:~/syspro$ gcc -o elapsed_time elapsed_time.c
choijm@embedded:~/syspro$ ./elapsed_time 10000
Elapsed time 0sec :14usec
choijm@embedded:~/syspro$ ./elapsed_time 100000
Elapsed time 0sec :137usec
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./elapsed_time 200000
Elapsed time 0sec :273usec
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./elapsed_time 1000000
Elapsed time 0sec :1367usec
choijm@embedded:~/syspro$ ./elapsed_time 10000000
Elapsed time 0sec :57550usec
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./elapsed_time 10000000
Elapsed time 0sec :57866usec
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./elapsed_time 10000000
Elapsed time 0sec :60158usec
choijm@embedded:~/syspro$ ./elapsed_time 10000000
Elapsed time 0sec :58835usec
choijm@embedded:~/syspro$ uname -a
Linux embedded 4.13.0-36-generic #40~16.04.1-Ubuntu SMP Fri Feb 16 23:25:58 UTC
2018 x86_64 x86_64 x86_64 GNU/Linux
choijm@embedded:~/syspro$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 23
model name     : Intel(R) Core(TM)2 Duo CPU      E7500   @ 2.93GHz
stepping       : 10
```

# Linux Performance Monitoring Tools (6/11)

## ■ Revisit combine(): code

```
choijm@embedded: ~/syspro
/* To measure the elapsed time of the combines, by choijm@dankook.ac.kr */
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>

typedef long data_t;
// typedef float data_t;

#define CONFIG_FOR_ADD
#ifdef CONFIG_FOR_ADD
#define OPER +
#define IDENT 0
#else
#define OPER *
#define IDENT 1
#endif

typedef struct {
    int len;
    data_t *data;
} vec_rec, *vec_ptr;

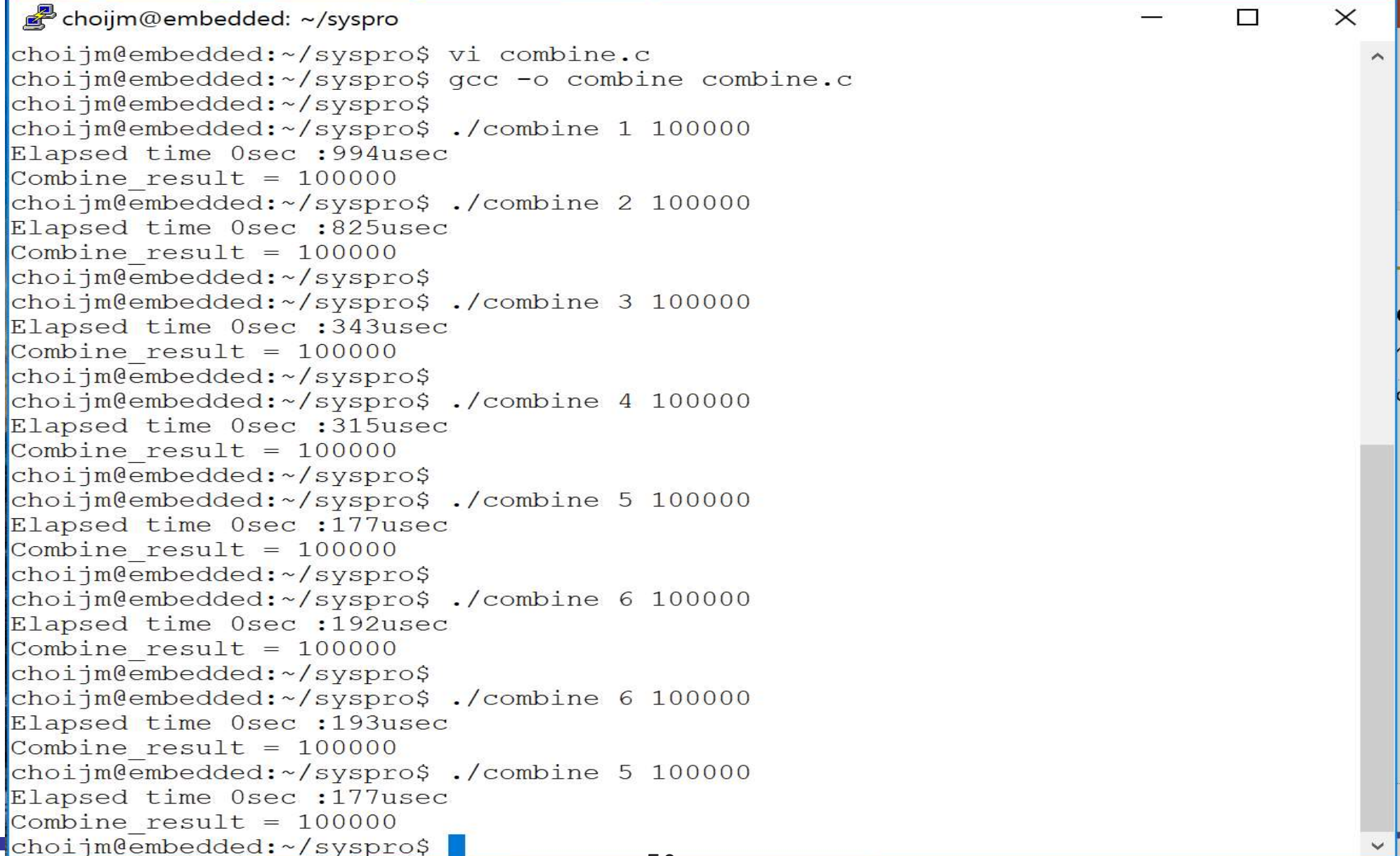
int main(int argc, char *argv[])
{
    int i, choice = 1, loop = 0;
    struct timeval stime, etime, gap;
    vec_ptr instance; data_t com_result = 0;
    void (*func_ptr)(vec_ptr, data_t *);

    if (argc == 3) {
        choice = atoi(argv[1]);
        loop = atoi(argv[2]);
        if (choice == 1)
            func_ptr = combinel;
        else if (choice == 2)
            func_ptr = combine2;
        else if (choice == 3)
            func_ptr = combine3;
    }

    "combine.c" [Modified] 231 lines --11%--
```

# Linux Performance Monitoring Tools (7/11)

## ■ Revisit combine(): add case



```
choijm@embedded: ~/syspro
choijm@embedded:~/syspro$ vi combine.c
choijm@embedded:~/syspro$ gcc -o combine combine.c
choijm@embedded:~/syspro$ ./combine 1 100000
Elapsed time 0sec :994usec
Combine_result = 100000
choijm@embedded:~/syspro$ ./combine 2 100000
Elapsed time 0sec :825usec
Combine_result = 100000
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./combine 3 100000
Elapsed time 0sec :343usec
Combine_result = 100000
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./combine 4 100000
Elapsed time 0sec :315usec
Combine_result = 100000
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./combine 5 100000
Elapsed time 0sec :177usec
Combine_result = 100000
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./combine 6 100000
Elapsed time 0sec :192usec
Combine_result = 100000
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./combine 6 100000
Elapsed time 0sec :193usec
Combine_result = 100000
choijm@embedded:~/syspro$ ./combine 5 100000
Elapsed time 0sec :177usec
Combine_result = 100000
choijm@embedded:~/syspro$
```

# Linux Performance Monitoring Tools (8/11)

## ■ Revisit combine(): multiply case

```
choijm@embedded: ~/syspro
vi combine.c
choijm@embedded:~/syspro$ vi combine.c
choijm@embedded:~/syspro$ gcc -o combine combine.c
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$ ./combine 1 100000
Elapsed time 0sec :991usec
Combine_result = 1
choijm@embedded:~/syspro$ ./combine 1 100000
Elapsed time 0sec :1004usec
Combine_result = 1
choijm@embedded:~/syspro$ ./combine 2 100000
Elapsed time 0sec :844usec
Combine_result = 1
choijm@embedded:~/syspro$ ./combine 3 100000
Elapsed time 0sec :343usec
Combine_result = 1
choijm@embedded:~/syspro$ ./combine 4 100000
Elapsed time 0sec :334usec
Combine_result = 1
choijm@embedded:~/syspro$ ./combine 5 100000
Elapsed time 0sec :175usec
Combine_result = 1
choijm@embedded:~/syspro$ ./combine 6 100000
Elapsed time 0sec :190usec
Combine_result = 1
choijm@embedded:~/syspro$ ./combine 5 100000
Elapsed time 0sec :175usec
Combine_result = 1
choijm@embedded:~/syspro$ ./combine 6 100000
Elapsed time 0sec :191usec
Combine_result = 1
choijm@embedded:~/syspro$
choijm@embedded:~/syspro$
```

☞ Can be different in your system

# Linux Performance Monitoring Tools (9/11)

## ■ strace

```
choijm@embedded:~/syspro$ strace -f ./mysh
execve("./mysh", ["/mysh"], [/* 31 vars */]) = 0
strace: [ Process PID=3538 runs in 32 bit mode. ]
brk(NULL)                               = 0x9c30000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7f32000
....
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "W177ELFW1W1W1W3W0W0W0W0W0W0W0W0W3W0W3W0W1W0W0W0W320W207W1W0004W0W0W0"..., 512)
fstat64(3, {st_mode=S_IFREG|0755, st_size=1786484, ...}) = 0
mmap2(NULL, 1792540, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xf7d65000
...
write(1, "/home/choijm/syspro$ ", 21/home/choijm/syspro$ ) = 21
read(0, cat test.c
"cat test.c\n", 1024)      = 11
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0xf7d64768) =
3541
strace: Process 3541 attached
[pid 3538] waitpid(3541, <unfinished ...>
[pid 3541] dup2(-7336184, 1)      = -1 EBADF (Bad file descriptor)
[pid 3541] execve("/bin/cat", ["cat", "test.c"], [/* 31 vars */]) = 0
strace: [ Process PID=3541 runs in 64 bit mode. ]
[pid 3541] brk(NULL)              = 0x1e3b000
...
[pid 3541] fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 8), ...}) = 0
[pid 3541] open("test.c", O_RDONLY) = 3
[pid 3541] fstat(3, {st_mode=S_IFREG|0664, st_size=196, ...}) = 0
[pid 3541] mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f79cac71000
[pid 3541] read(3, "#include <stdio.h>\n\nint main(int"... , 131072) = 196
[pid 3541] write(1, "#include <stdio.h>\n\nint main(int"... , 196#include <stdio.h>

int main(int argc, char *argv[])
{
    int i, loop = 0;
    ...
```



# Linux Performance Monitoring Tools (10/11)

## ■ top

- ✓ To display the system tasks (refreshes every 3 seconds)
- ✓ Three areas
  - Summary Area: 1) Uptime and load average, 2) Task state, 3) CPU state, 4) Memory usage, 5) Swap
  - Fields/Columns Header: pid, user name, priority, nice value, Virtual memory size, Resident memory size, Shared memory size, Status, %CPU, %Mem, Time+, command
  - Task Area: task information

```
choijm@embedded: ~
choijm@embedded:~$ top
top - 11:55:37 up 11 min,  2 users,  load average: 2.05, 0.81, 0.39
Tasks: 134 total,   1 running, 133 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4.0 us,   2.2 sy,   0.0 ni, 15.9 id, 77.3 wa,   0.0 hi,   0.7 si,   0.0 st
KiB Mem : 3746108 total,  104636 free, 3578772 used,   62700 buff/cache
KiB Swap: 3895292 total, 1931516 free, 1963776 used.   31800 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
 2418 choijm    20   0 3908452  3.265g    8 D   9.0  91.4   0:15.47 combine
    39 root      20   0         0        0    0 D    2.3   0.0   0:01.82 kswapd0
     8 root      20   0         0        0    0 S    0.3   0.0   0:00.20 rcu_sched
  174 root       0 -20         0        0    0 S    0.3   0.0   0:00.22 kworker/1:+
 2106 choijm    20   0   50136    372    0 R    0.3   0.0   0:00.47 top
     1 root      20   0  119856    200    0 S    0.0   0.0   0:01.26 systemd
     2 root      20   0         0        0    0 S    0.0   0.0   0:00.00 kthreadd
     4 root       0 -20         0        0    0 S    0.0   0.0   0:00.00 kworker/0:~
     5 root      20   0         0        0    0 S    0.0   0.0   0:00.01 kworker/u4+
     6 root       0 -20         0        0    0 S    0.0   0.0   0:00.00 mm_percpu_+
     7 root      20   0         0        0    0 S    0.0   0.0   0:00.01 ksoftirqd/0
     9 root      20   0         0        0    0 S    0.0   0.0   0:00.00 rcu_bh
    10 root      rt    0         0        0    0 S    0.0   0.0   0:00.00 migration/0
    11 root      rt    0         0        0    0 S    0.0   0.0   0:00.00 watchdog/0
    12 root      20   0         0        0    0 S    0.0   0.0   0:00.00 cpuhp/0
```

# Linux Performance Monitoring Tools (11/11)

## ■ Perf event

```
# perf list
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                [Hardware event]
  instructions                        [Hardware event]
  cache-references                     [Hardware event]
  cache-misses                        [Hardware event]
  branch-instructions OR branches     [Hardware event]
  branch-misses                       [Hardware event]
  bus-cycles                           [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend [Hardware event]
[...]
```

|                              |                  |
|------------------------------|------------------|
| cpu-clock                    | [Software event] |
| task-clock                   | [Software event] |
| page-faults OR faults        | [Software event] |
| context-switches OR cs       | [Software event] |
| cpu-migrations OR migrations | [Software event] |

```
[...]
```

|                       |                        |
|-----------------------|------------------------|
| L1-dcache-loads       | [Hardware cache event] |
| L1-dcache-load-misses | [Hardware cache event] |
| L1-dcache-stores      | [Hardware cache event] |

```
[...]
```

|                             |                    |
|-----------------------------|--------------------|
| skb:kfree_skb               | [Tracepoint event] |
| skb:consume_skb             | [Tracepoint event] |
| skb:skb_copy_datagram_iovec | [Tracepoint event] |
| net:net_dev_xmit            | [Tracepoint event] |
| net:net_dev_queue           | [Tracepoint event] |
| net:netif_receive_skb       | [Tracepoint event] |
| net:netif_rx                | [Tracepoint event] |



# COW: memory copy optimization (1/2)

## ■ What is COW (Copy On Write)?

### ✓ Paging system

- Virtual memory: **divide in pages**
  - Address space → text + data + stack → 2 pages + 1 page + 2 pages
- Physical memory: **divide into page frames**
- Address translation: using PT (page tables)
  - From virtual address to physical address

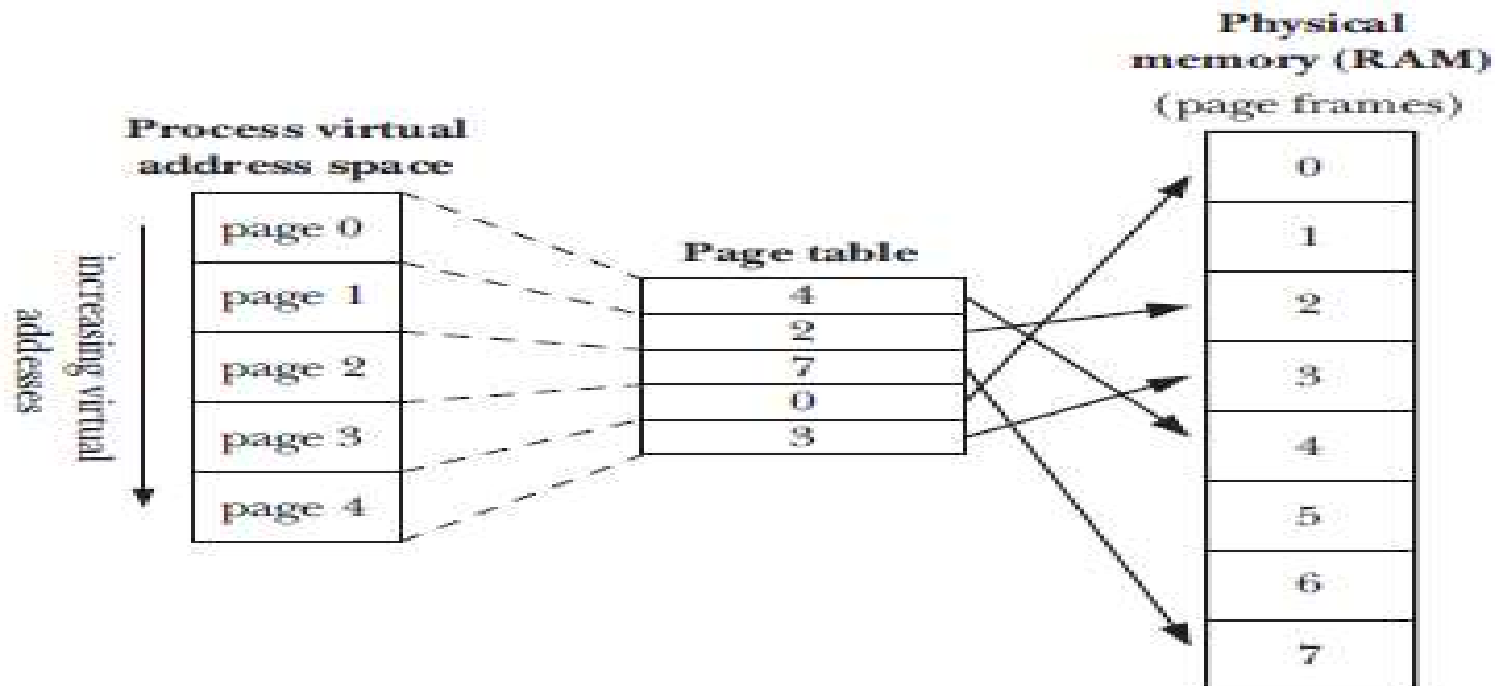


Figure 6-2: Overview of virtual memory (Source: LPI)



# COW: memory copy optimization (2/2)

## ■ What is COW (Copy On Write)?

- ✓ fork(): create a new process that has the same image of the parent's address space
  - Text: can share since it is read-only
  - Data, Stack, ..: need to copy for supporting independence
- ✓ Issue for copying
  - 1) time-consuming job, 2) often not use (e.g. fork and execve)
- ✓ Solutions
  - Copy when a write actually occurs (COW) → **Lazy evaluation**

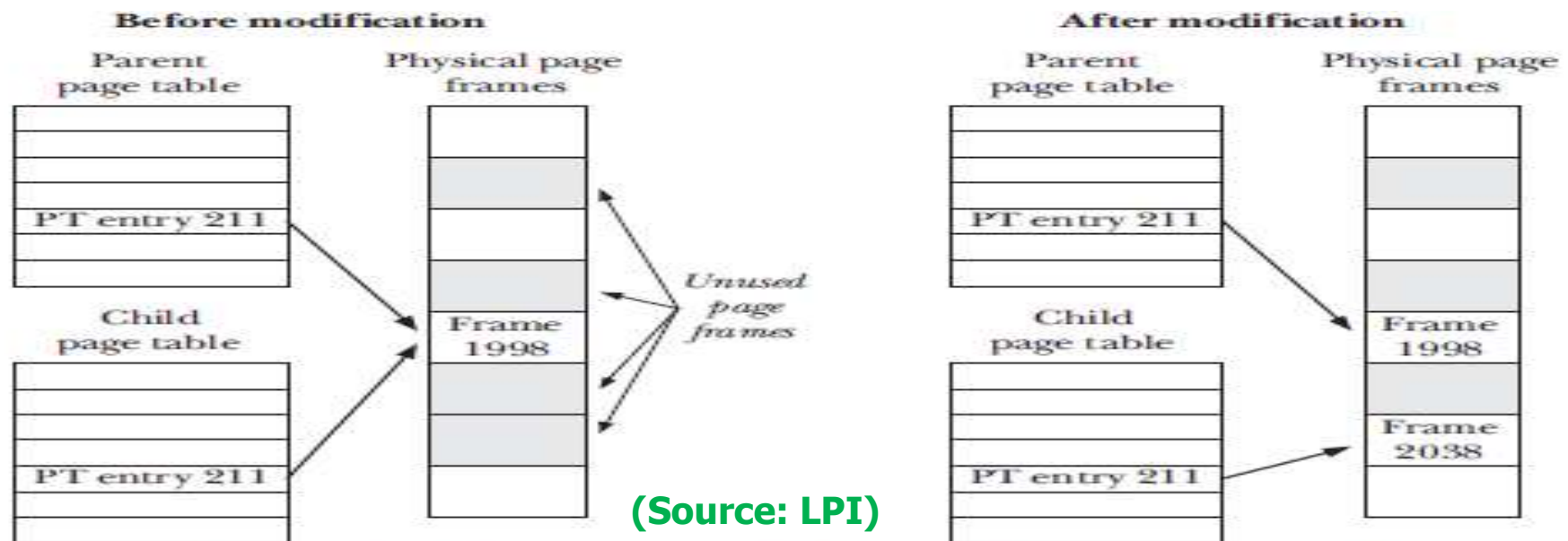


Figure 24-3: Page tables before and after modification of a shared copy-on-write page

# Amdahl's Law (1/1)

## ■ How much can we enhance a program?

- ✓ Execution time of an enhanced program

$$\begin{aligned}T_{new} &= (1 - \alpha)T_{old} + (\alpha T_{old})/k \\ &= T_{old}[(1 - \alpha) + \alpha/k]\end{aligned}$$

- $\alpha$  : enhanced fraction
- $k$  : enhanced factor

- ✓ Speed up

$$S = T_{old}/T_{new} = \frac{1}{(1 - \alpha) + \alpha/k}$$

- ✓ Implication

- Which one is better?
  - $\alpha = 60\%$  and  $k = 2$  or  $\alpha = 30\%$  and  $k = 10$
- $\alpha = 60\%$  and  $k = 2 \rightarrow S = 1/(0.4 + 0.3) = 1/0.7 = 1.43$
- $\alpha = 30\%$  and  $k = 10 \rightarrow S = 1/(0.7 + 0.03) = 1/0.73 = 1.37$
- We must improve the speed of a very large fraction such as loop body and bottleneck portion (e.g. combine())



# Conclusion

---

## ■ Performance Improvement Techniques

- ✓ High-level Design: choose appropriate algorithms and data structure
- ✓ Basic coding principles: eliminating excessive function calls and unnecessary memory references
- ✓ Low-level optimization: loop unrolling and exploit parallelism (based on data dependency, resource constraint, and branch prediction)
- ✓ Consider Amdahl's law

## ☞ Homework 7: Performance measure and discussion

### 1.1 Requirements

- Measure performance differences using 2 or more programs
- Program examples (later is better)
  - 1) combine 1~6, 2) cache performance,
  - 3) branch prediction, 4) Amdahl's law, 5) various algorithms,
  - 6) anything you want (even a game program or DB program)
- Using gettimeofday() or other tools

### 1.2 Write a report

- 1) Introduction (why, what, how, ...), 2) Test program/Source code, 3) Snapshots, 4) Discussion
- "Discussion" is important in this homework

### 1.3 How to submit? Upload a report to google form

### 1.4 Deadline: a week later (same time)





# Quiz for this Lecture

## ■ Quiz

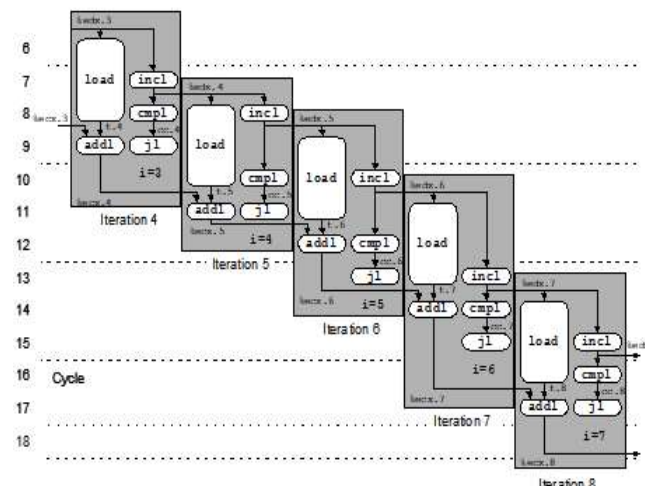
1. What are the differences between the lower1() and lower2() in the below left figure?
2. Explain the Intel techniques you can observe in the below center figure (at least 5 techniques).
3. Describe what is the bubble in the right figure. Discuss how to make a program that can mitigate the performance degradation due to the control hazard (at least 2 ways)?
4. Discuss the effect of Amdahl's law using your own programs.

```

1  /* Convert string to lowercase: slow */
2  void lower1(char *s)
3  {
4      int i;
5      for (i = 0; i < strlen(s); i++)
6          if (s[i] >= 'A' && s[i] <= 'Z')
7              s[i] -= ('A' - 'a');
8  }
9
10 /* Convert string to lowercase: faster */
11 void lower2(char *s)
12 {
13     int i;
14     int len = strlen(s);
15     for (i = 0; i < len; i++)
16         if (s[i] >= 'A' && s[i] <= 'Z')
17             s[i] -= ('A' - 'a');
18 }
19
20 /* Sample implementation of library function strlen */
21 /* Compute length of string */
22 size_t strlen(const char *s)
23 {
24     int length = 0;
25     while (*s != '\0') {
26         s++;
27         length++;
28     }
29     return length;
30 }
31
32

```

Figure 5.7 Lowercase conversion routines. The two procedures have radically different performance.



| cycle | F             | D | X | M | W |  |
|-------|---------------|---|---|---|---|--|
| 1     | add r1,r3,r4  |   |   |   |   |  |
| 2     | mul r5,r6,r7  |   |   |   |   |  |
| 3     | bnez loop,r1  |   |   |   |   |  |
| 4     | X             |   |   |   |   |  |
| 5     | X             | X |   |   |   |  |
| 6     | X             | X | X |   |   |  |
| 7     | X             | X | X | X |   |  |
| 8     | div r8,r3,r6  | X | X | X | X |  |
| 9     | add r10,r8,r9 |   | X | X | X |  |
| 10    | jmp loop      |   |   | X | X |  |

When there is a dependency we say that the pipeline is **stalled** or a **bubble** is inserted waiting for the dependency to solve. Here a control dependency causes a 4 cycles stall.

(Source: CSAPP)



# Appendix: Memory Hierarchy and Performance

## ■ Cache performance

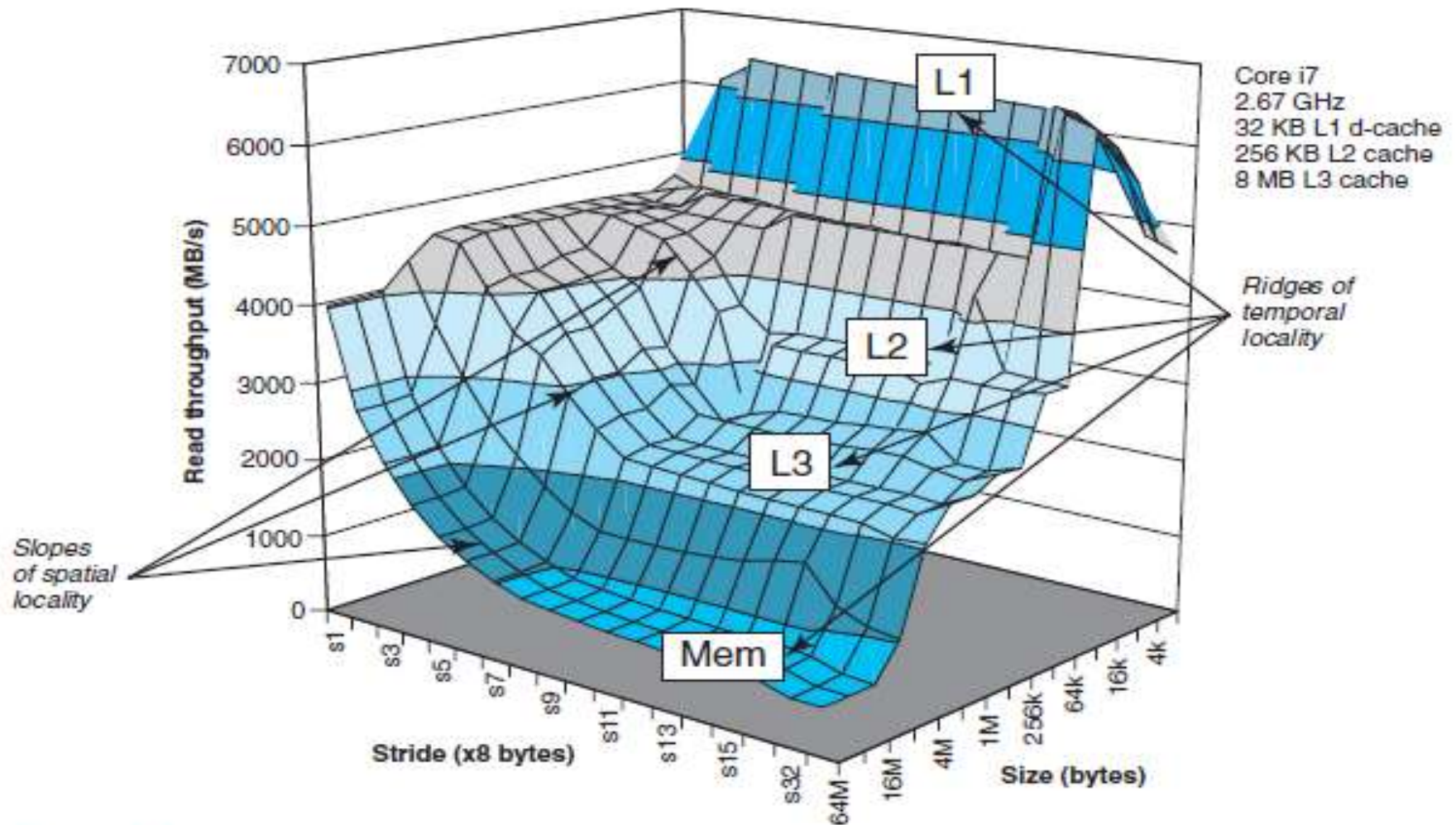


Figure 6.43 The memory mountain.

# Appendix: SIMD effect

## ■ Using SIMD

- ✓ We can achieve the following results from the combine example

| Method                | Integer |      | Floating point |      |      |
|-----------------------|---------|------|----------------|------|------|
|                       | +       | *    | +              | F *  | D *  |
| SSE + 8-way unrolling | 0.25    | 0.55 | 0.25           | 0.24 | 0.58 |
| Throughput bound      | 0.25    | 0.50 | 0.25           | 0.25 | 0.50 |

- Four operations in parallel for integer and single precision float, while two for double precision float.



- 본 교재는 2025년도 과학기술정보통신부 및 정보통신기획평가원의 ‘SW중심대학사업’ 지원을 받아 제작 되었습니다.
- 본 결과물의 내용을 전재할 수 없으며, 인용(재사용)할 때에는 반드시 과학기술정보통신부와 정보통신기획평가원이 지원한 ‘SW중심대학’의 결과물이라는 출처를 밝혀야 합니다.

**IITP** 정보통신기획평가원  
디지털인재양성단 SW인재팀

