Lecture Note 8. Optimization

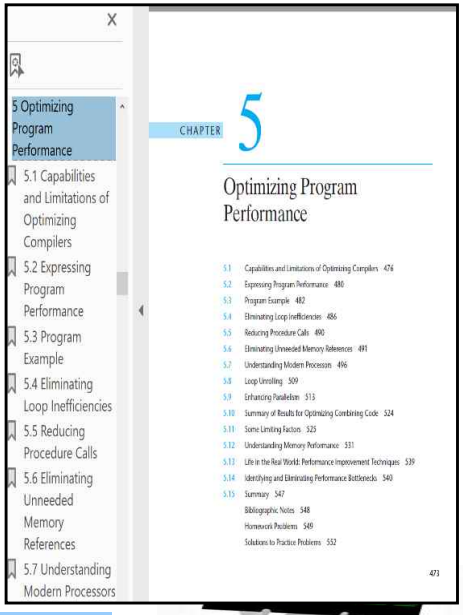
November 17, 2021

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

- 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

- 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

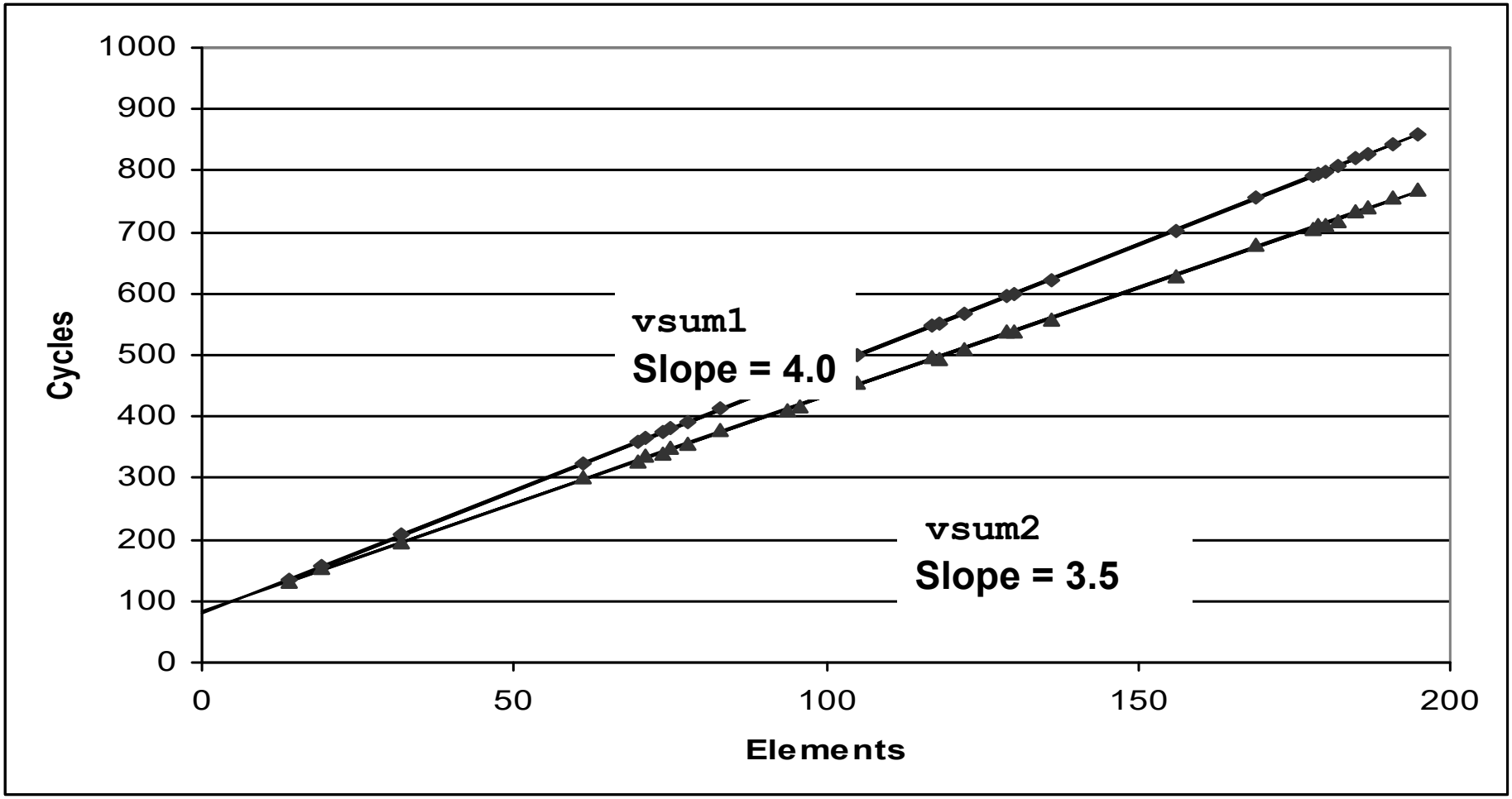
- Example program

```
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 (1st Ed.), Figure 5.1)

Expressing Program Performance

- CPE (Cycles Per Element)

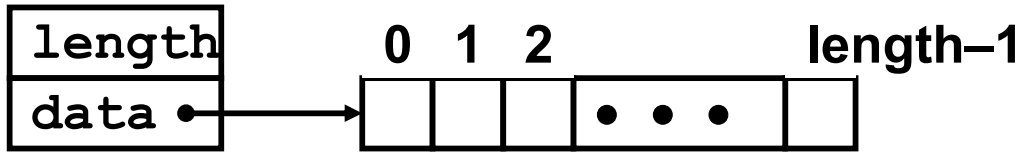


- ✓ Using a least square fit mechanism
  - vsum1 = 80 + 4.0*n vs vsum2 = 83.5 + 3.5*n

Optimization Practice

- Basic Program
  - ✓ Data type

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

// data_t는 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

- 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

- 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

- 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

- 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;
        }
}
```

```
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;
        }
}
```

Optimization Practice

- 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

Optimization Practice

- 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];
        }
}
```

```
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

- 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**

Optimization Practice

- 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;
}
```

```
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

- 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

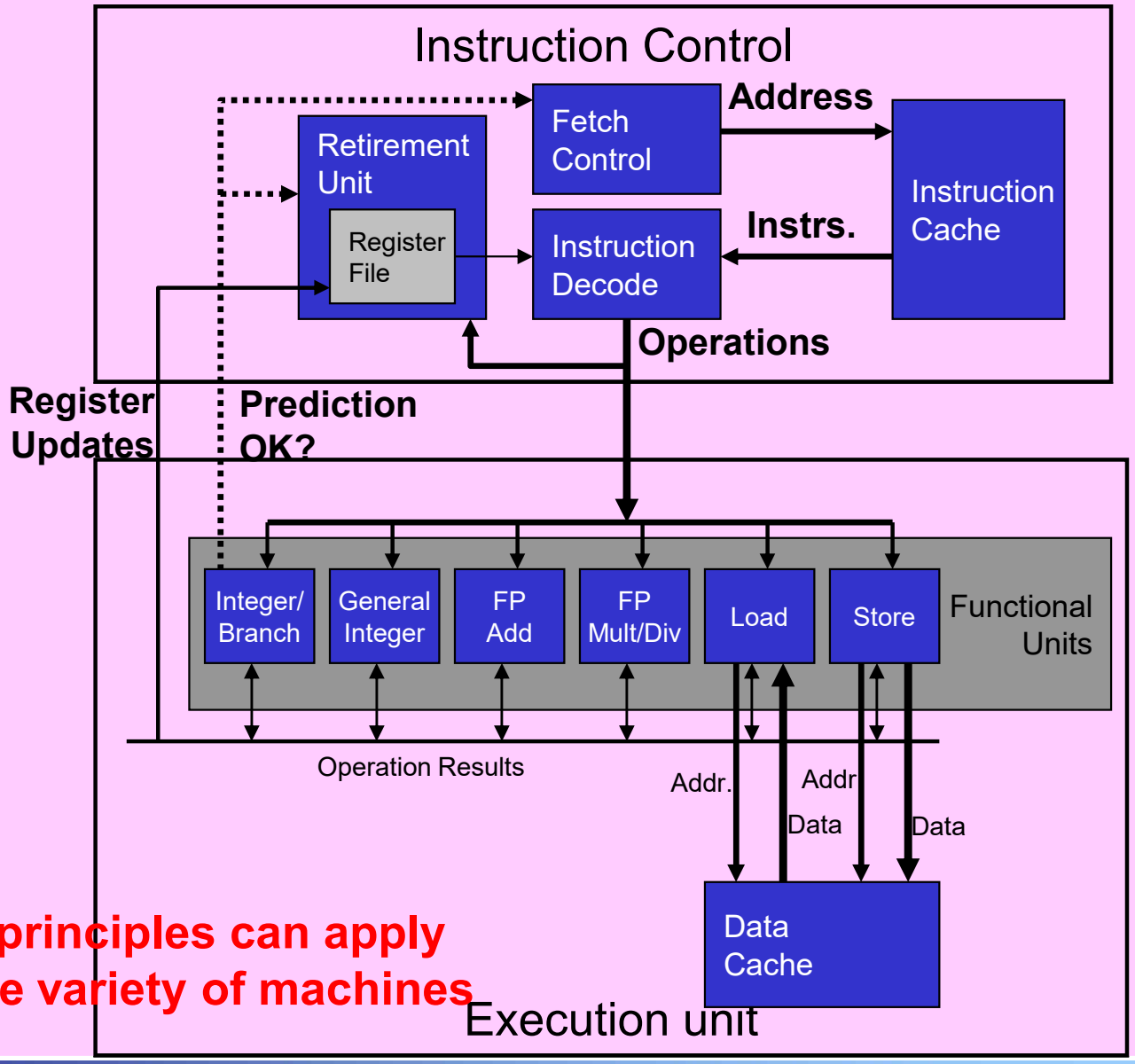
Optimization Practice

- 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
        jl      .L24
```

☞ **4 CPE ??**



☞ **general principles can apply to a wide variety of machines**

Understanding Modern Processor

- 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

- 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
    - values passed directly from one functional unit to another

Understanding Modern Processor

- 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

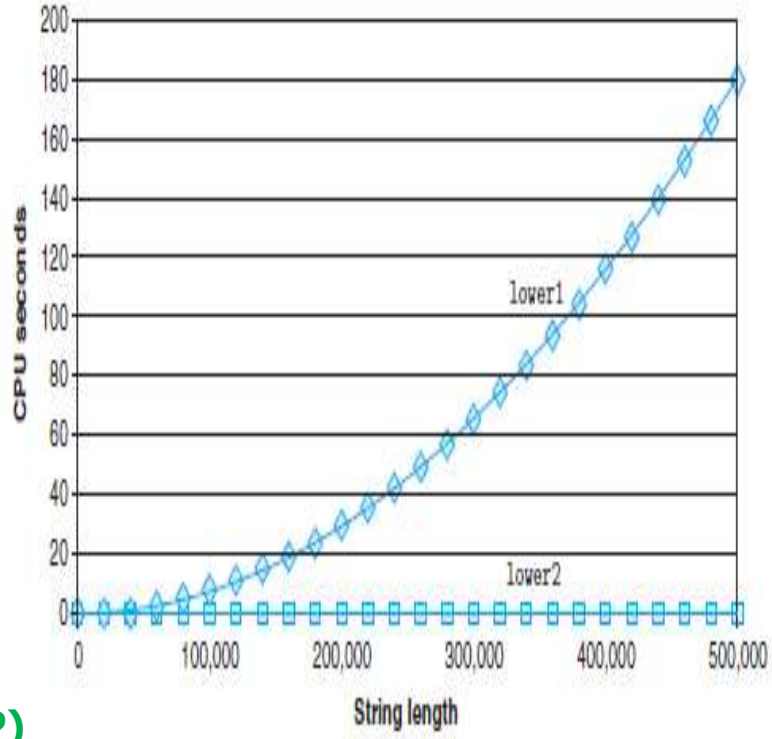
- ☞ **Integer Multiply, FP add, load, ... can be pipelined**
- ☞ **The latency and issue time are improved in Intel Core i7 (eg. multiply latency = 3, add issue time = 0.33 (3 pipeline engines), …)**

Quiz for 13th-Week 1st-Lesson

- Quiz
  - ✓ 1. What are the differences between the lower1() and lower2() in the below codes?
  - ✓ 2. Discuss the differences between combine3() and combine4() using Intel assembly language
  - ✓ Due: until 6 PM Friday of this week (3th, December)

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

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



(Source: CSAPP)

Understanding Modern Processor

- 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
```

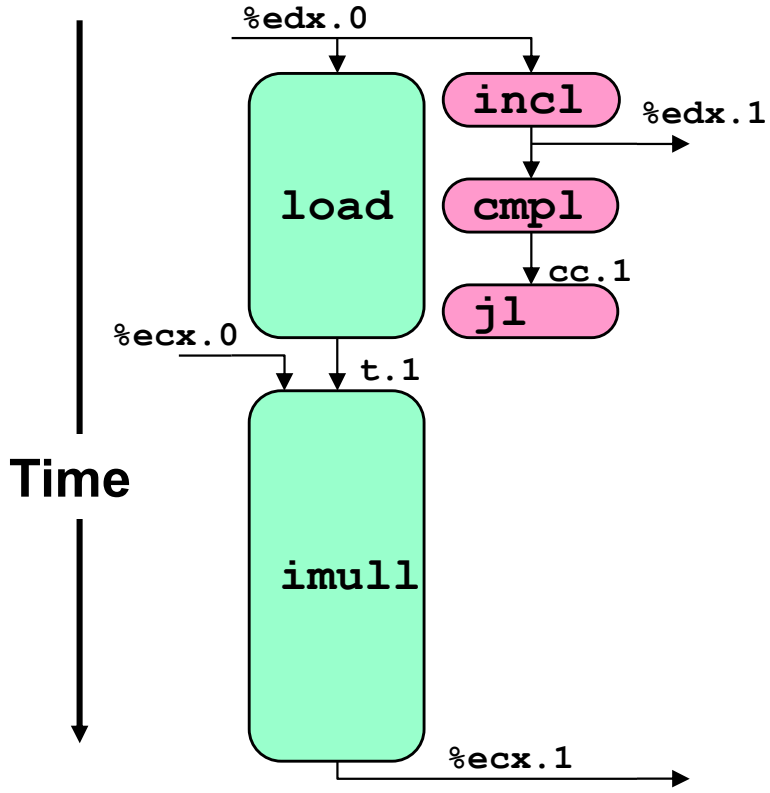
Assembly Instructions	Execution unit operations
.L24:	
imull (%eax,%edx, 4), %ecx	load (%eax, %edx.0, 4) → t.1
	imull t.1, %ecx.0 → %ecx.1
incl %edx	incl %edx.0 → %edx.1
cmpl %esi, %edx	cmpl %esi, %edx.1 → cc.1
jl .L24	jl-taken cc.1

☞ **branch prediction**

Understanding Modern Processor

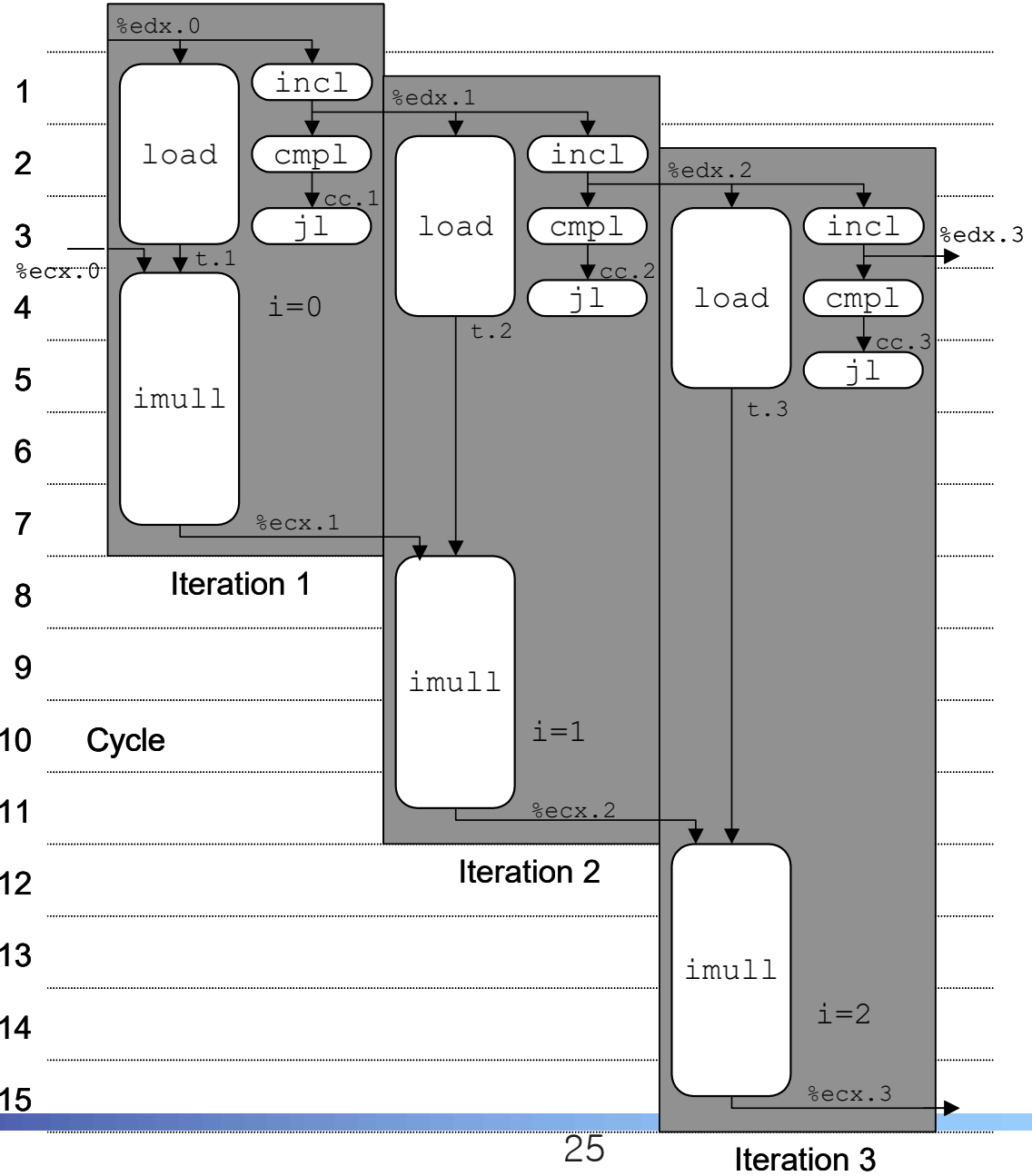
- timing in execution unit: multiplication case

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

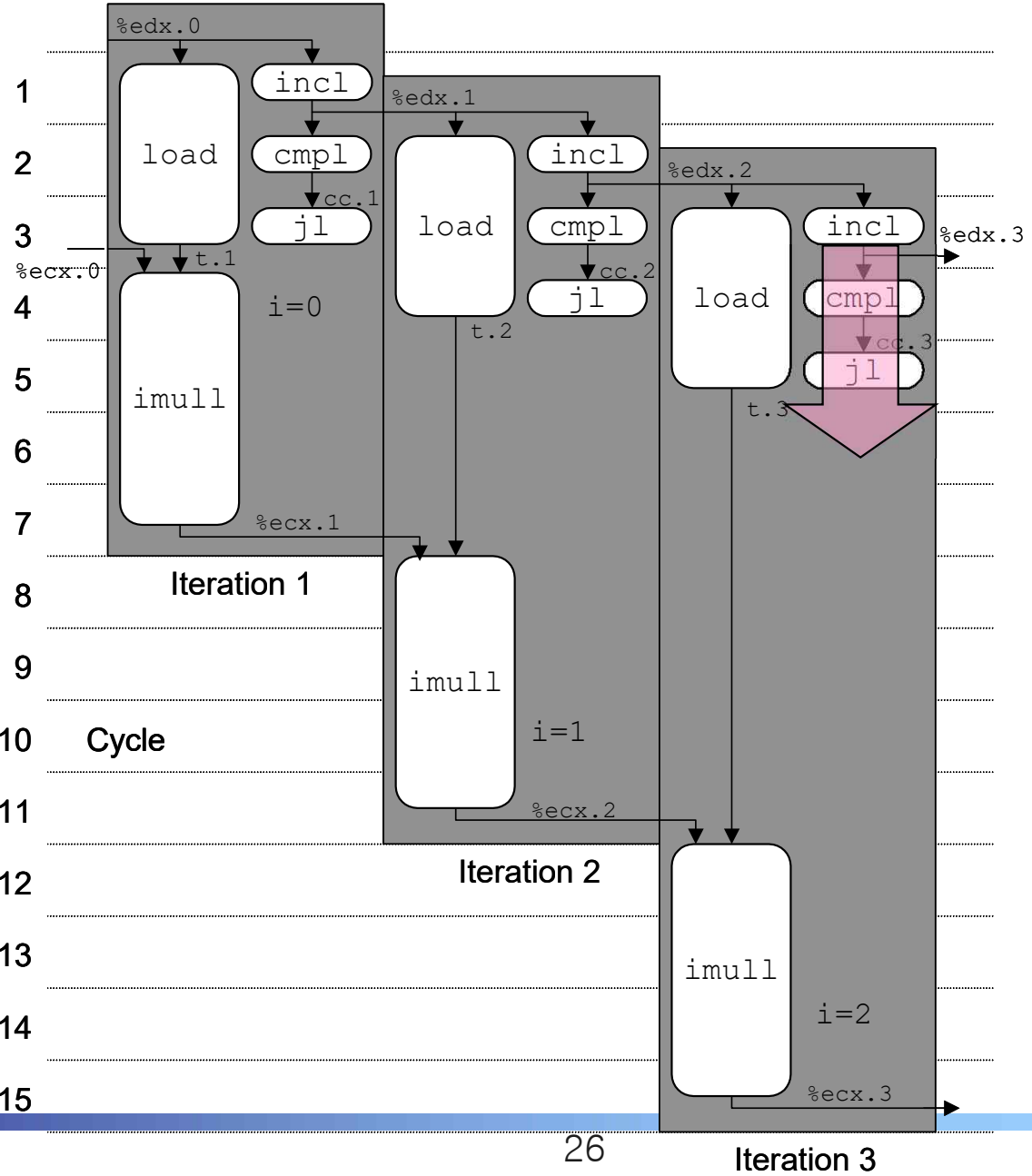
- Scheduling of Operations with Unlimited Resources



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

Understanding Modern Processor

- Scheduling of Operations with Resources Constraints

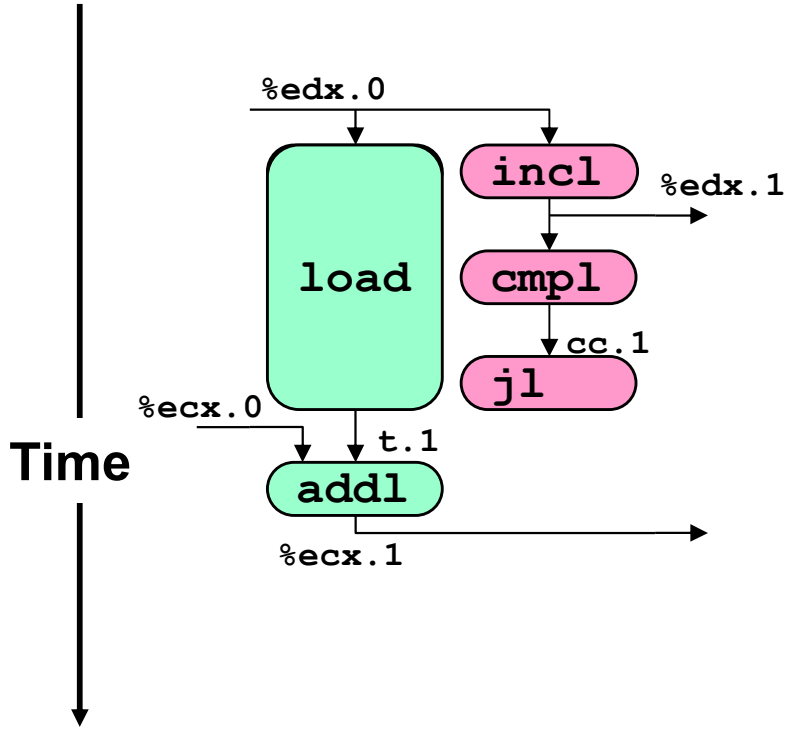


☞ **CPE = 4.0**

Understanding Modern Processor

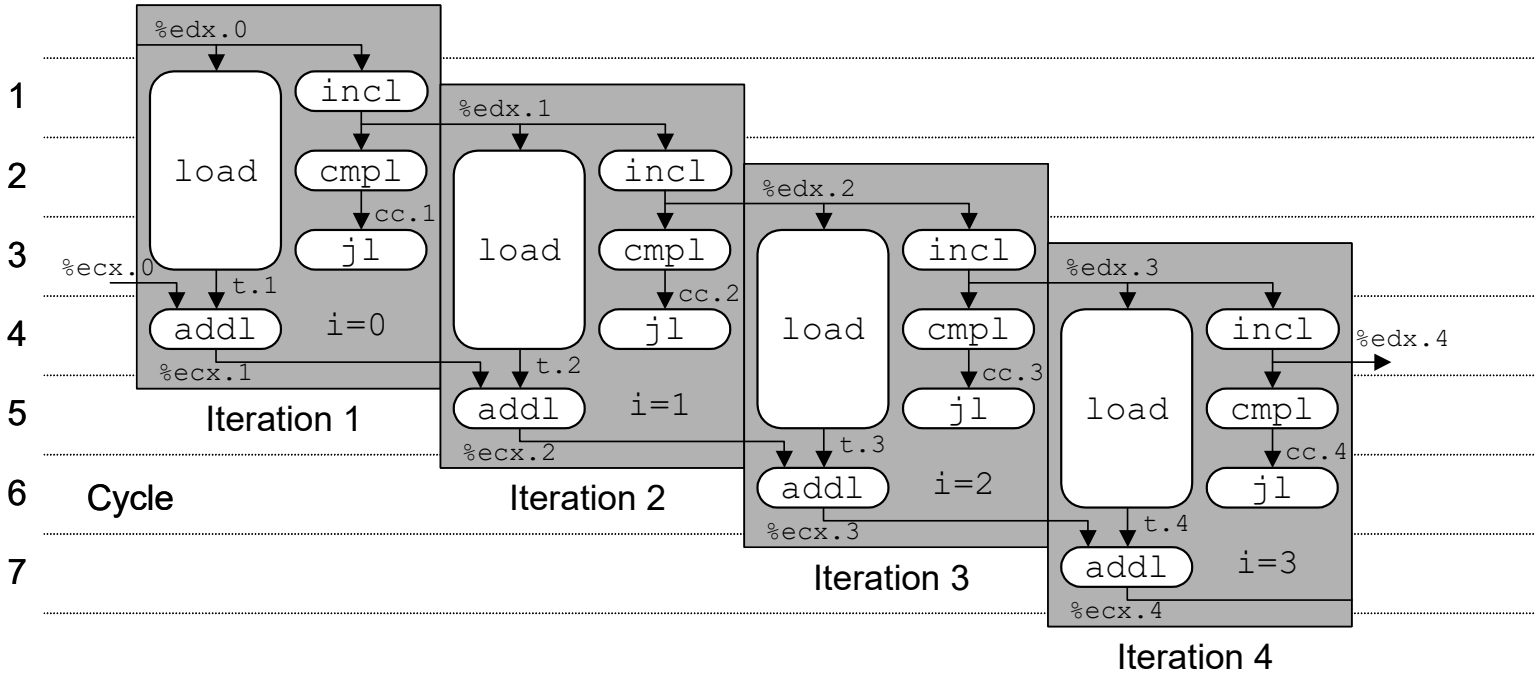
- timing in execution unit: add case

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

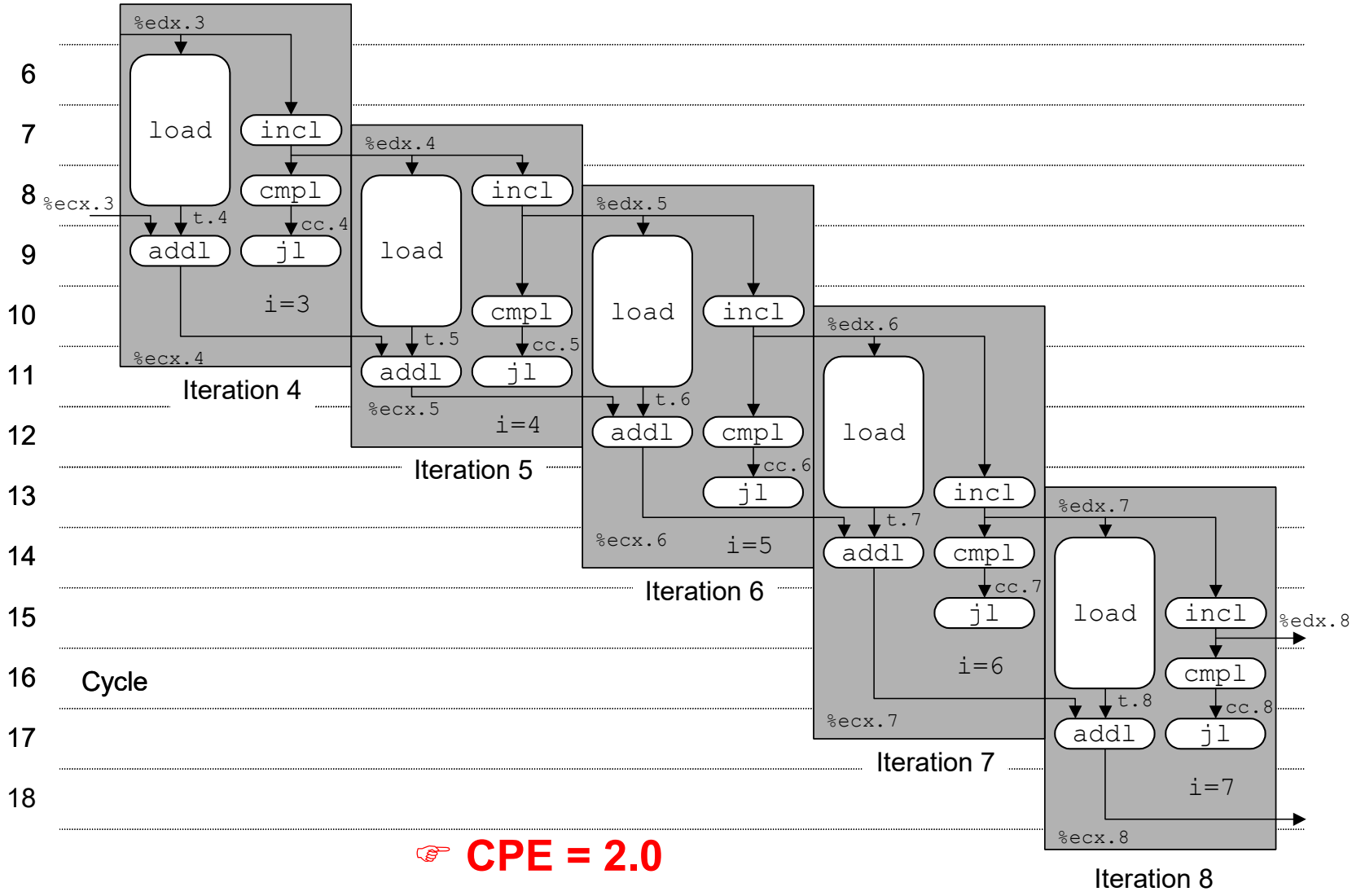
- Scheduling of Operations with Unlimited Resources



☞ **CPE = 1.0**

Understanding Modern Processor

- Scheduling of Operations with Resources Constraints



☞ **CPE = 2.0**

Understanding Modern Processor

- 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

- Loop unrolling
  - ✓ Original vs. New

```
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;
}
```

```
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;
}
```

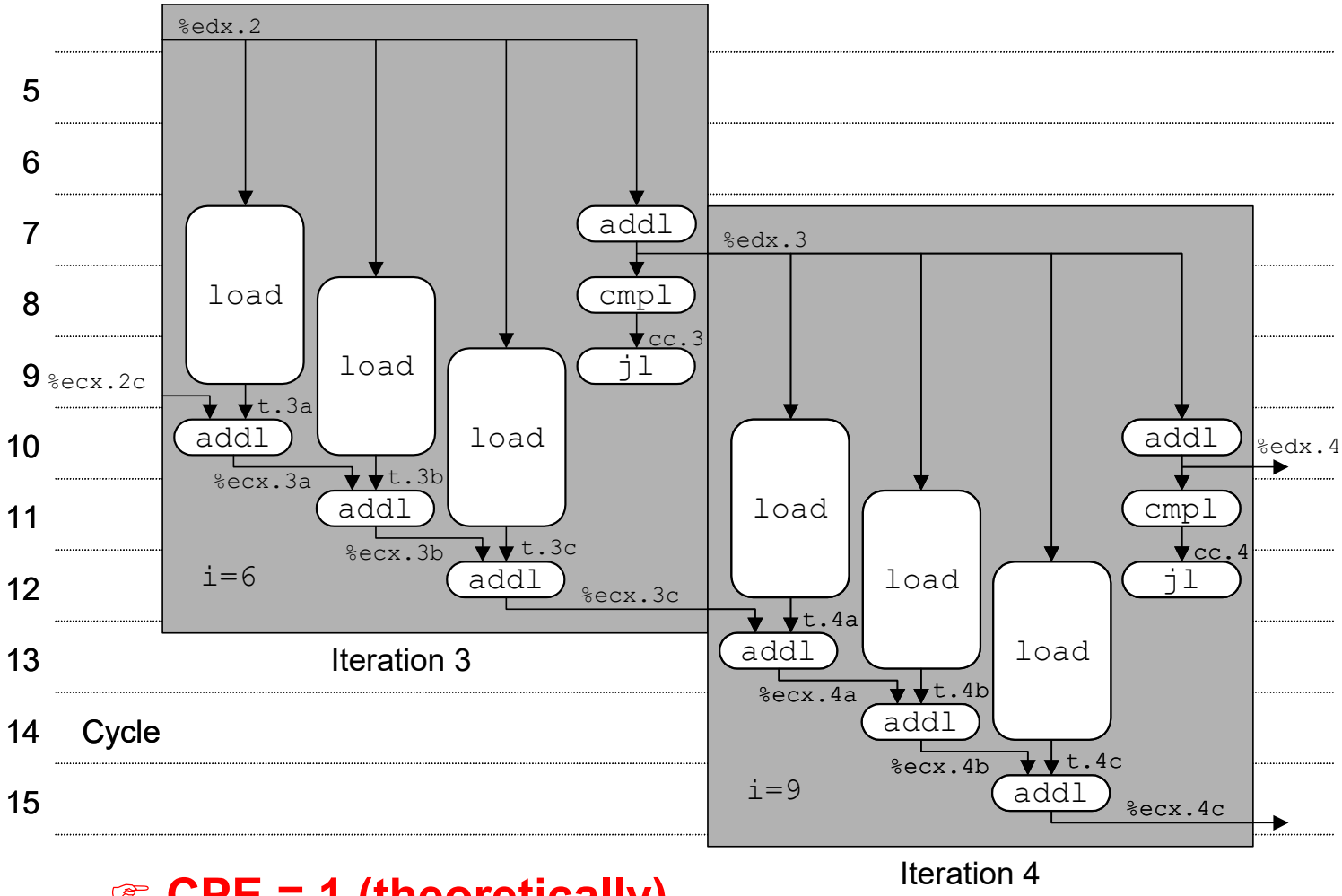
Enhanced Optimization

- Loop unrolling
  - ✓ instruction and operations

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

Enhanced Optimization

- 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

- 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)**

Enhanced Optimization

- 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;
}
```

```
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;
}
```

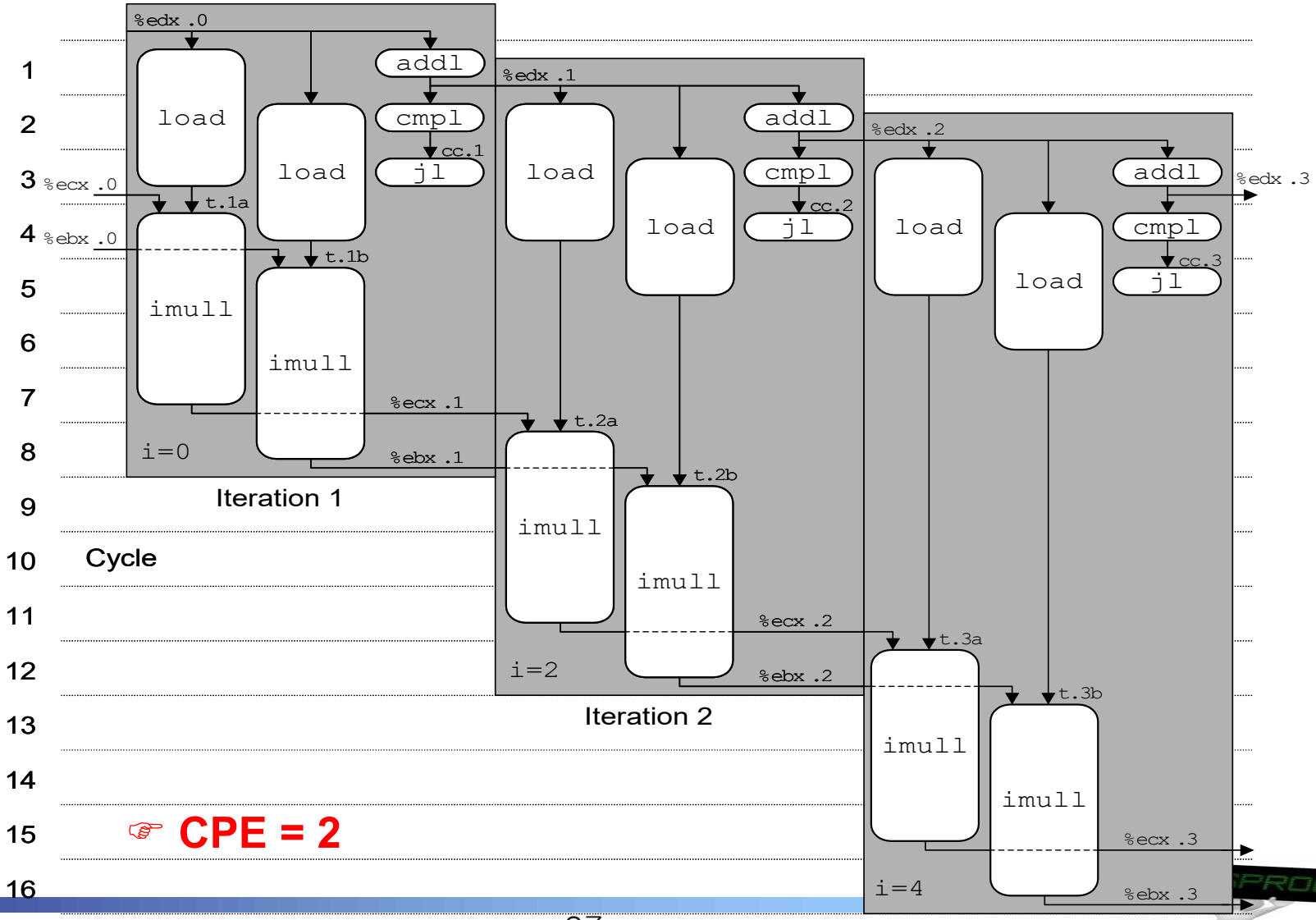
Enhanced Optimization

- Loop splitting
  - ✓ instruction and operations

Assembly Instructions	Execution unit operations
.L151:	
imull (%eax,%edx, 4), %ecx	load (%eax,%edx.0,4) ➔ t.1a
	imull t.1a, %ecx.0 ➔ %ecx.1
imull 4(%eax,%edx, 4), %ebx	load 4(%eax,%edx.0,4) ➔ t.1b
	imull t.1b, %ebx.0 ➔ %ebx.1
addl %edx, 2	iaddl $2,%edx.0 ➔ %edx.1
cmpl %esi, %edx	cmpl %esi, %edx.1 ➔ cc.1
jl .L151	jl-taken cc.1

Enhanced Optimization

- Loop splitting
  - ✓ scheduling of operations



Enhanced Optimization

- 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, Parallelism * 2	1.50	2.00	2.00	2.50

Enhanced Optimization

- 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

- 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

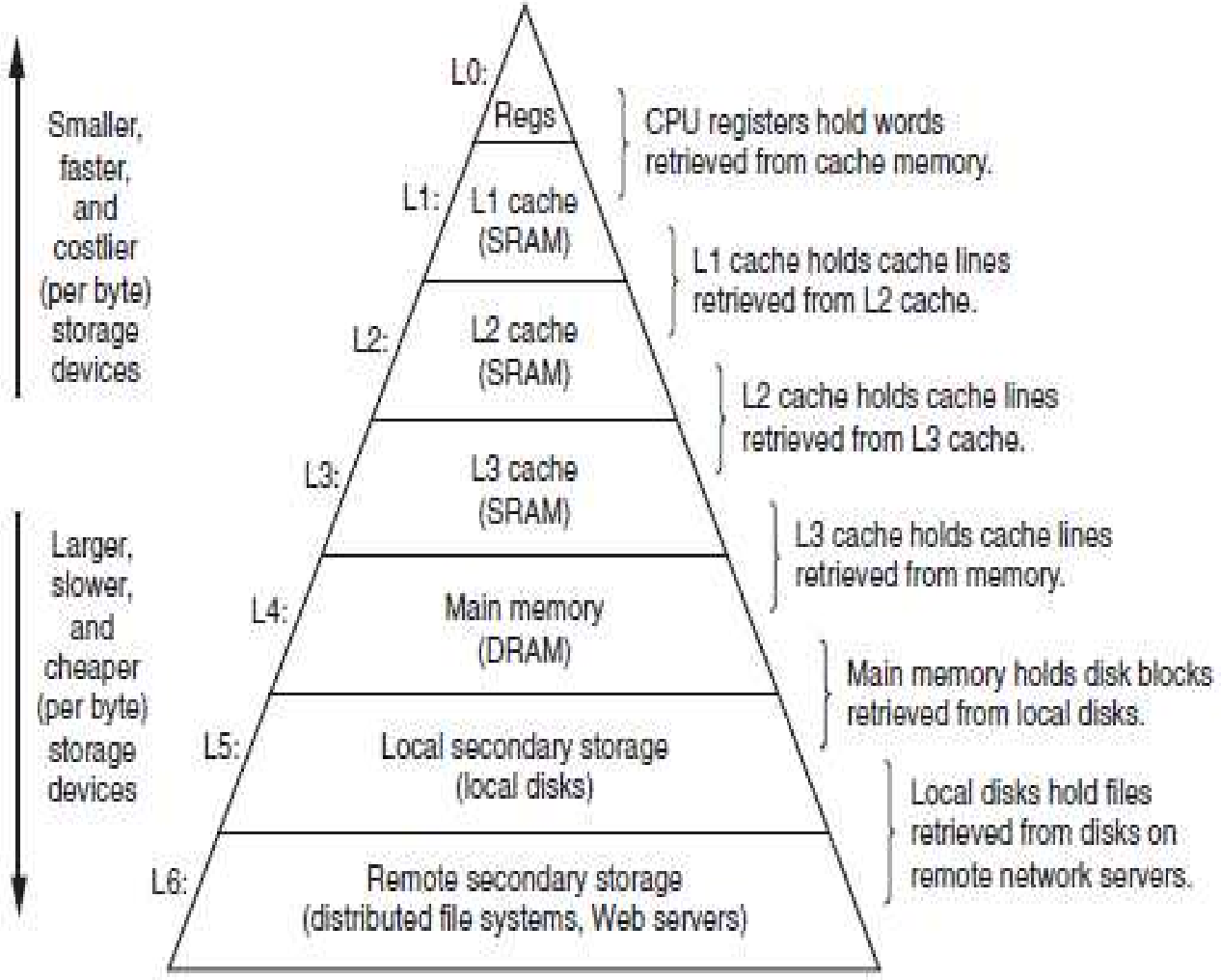
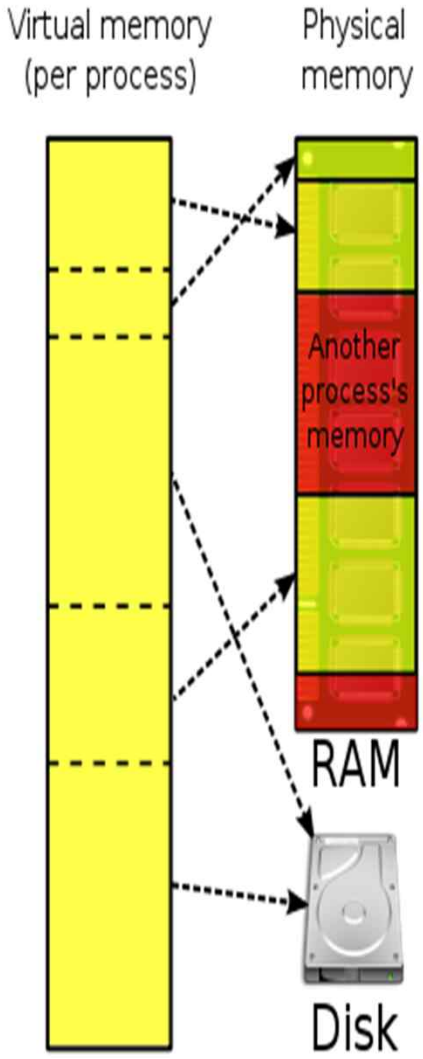


Figure 6.23 The memory hierarchy.



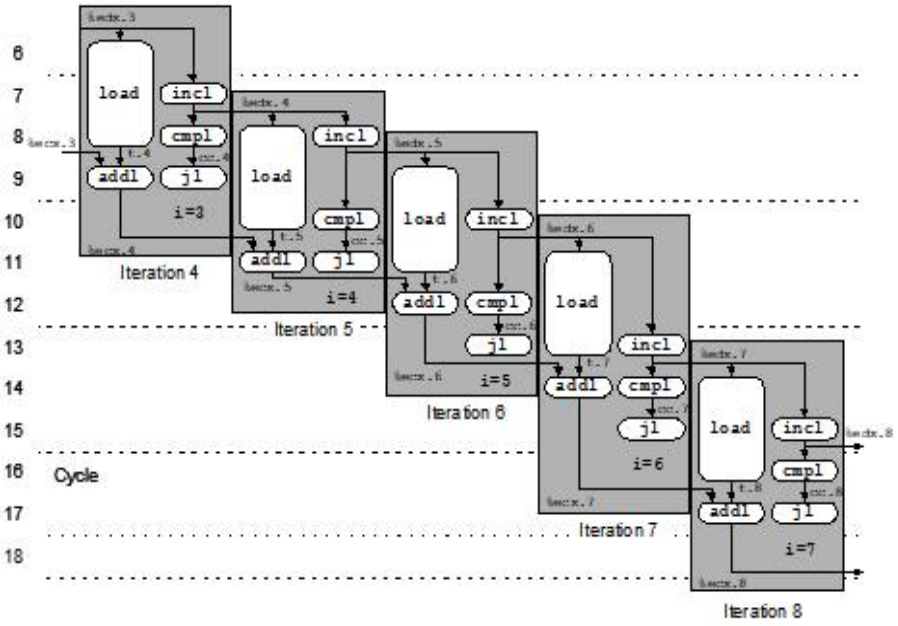
(Source: CSAPP)

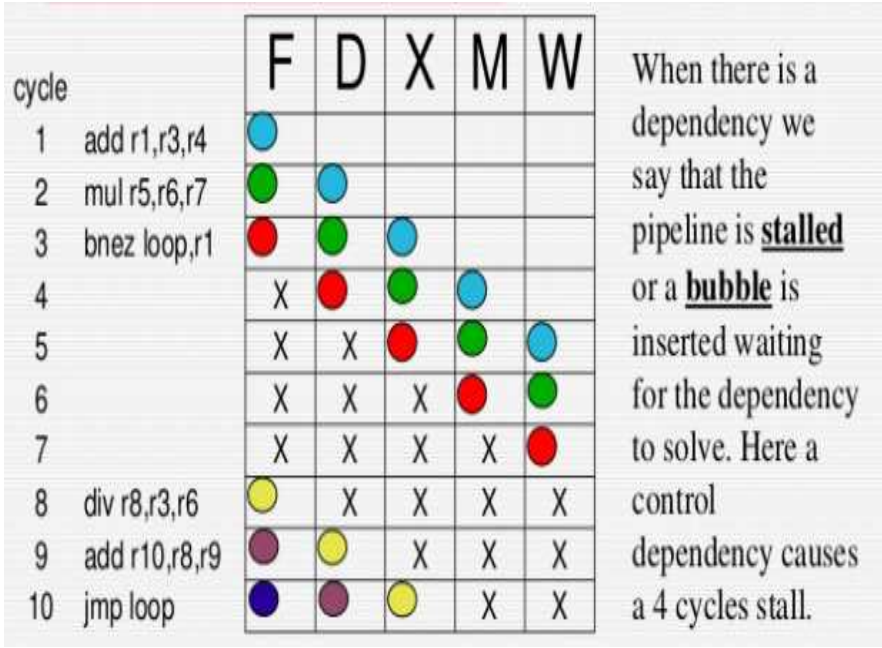
Memory Hierarchy and Performance (2/4)

- 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} x T_M) + (P_{miss} x T_M)
      - 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= 100ns, T_D= 10ms (10,000,000ns)
      - P_{hit}= 50% ➔ AMAT = 0.5 x 100 + 0.5 x 10,000,000 = 5,000,050 = 5ms
      - P_{hit}= 90% ➔ AMAT = 0.9 x 100 + 0.1 x 10,000,000 = 1,000,090 = 1ms
      - P_{hit}= 99% ➔ AMAT = 0.99 x 100 + 0.01 x 10,000,000 = 100,099 = 0.1ms
    - 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

Quiz for 13th-Week 2nd-Lesson

- Quiz
  - ✓ 1. Explain the Intel techniques you can observe in the below left figure (at least 5 techniques).
  - ✓ 2. Discuss how to make a program that can mitigate the performance degradation due to the control hazard (at least 2 ways)?
  - ✓ Due: until 6 PM Friday of this week (3th, December)



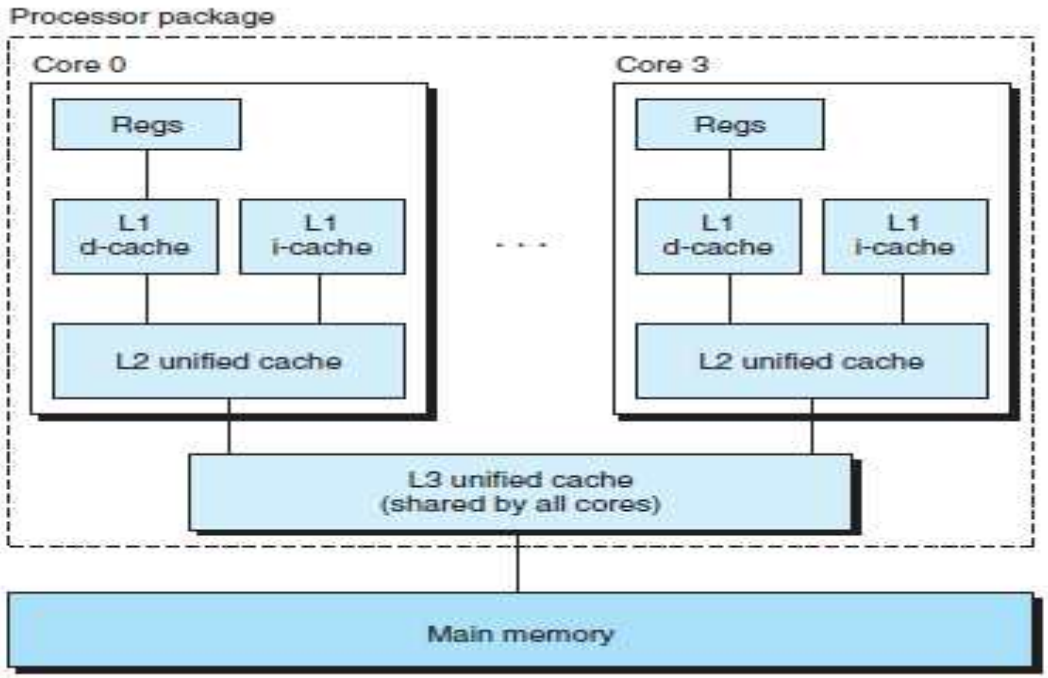


(Source: https://www.slideshare.net/rinnocente/computer-architecture-branch-prediction)

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      }
```

code/mem/matmult/mm.c

(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      }
```

code/mem/matmult/mm.c

(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      }
```

code/mem/matmult/mm.c

(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      }
```

code/mem/matmult/mm.c

(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      }
```

code/mem/matmult/mm.c

(f) Version *ikj*

code/mem/matmult/mm.c

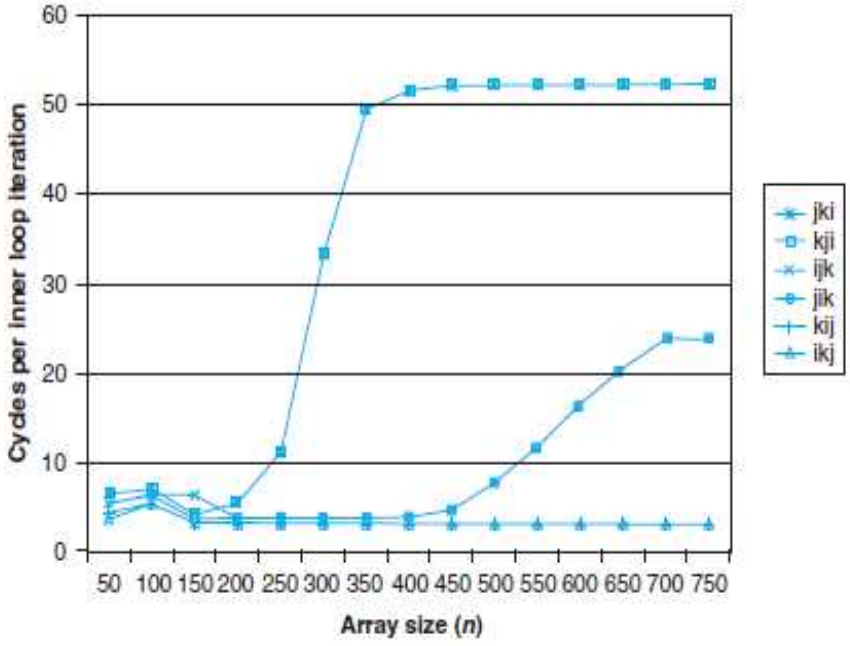
```
1  for (i = 0; i < n; i++)
2      for (k = 0; k < n; k++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += r*B[k][j];
6      }
```

code/mem/matmult/mm.c

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

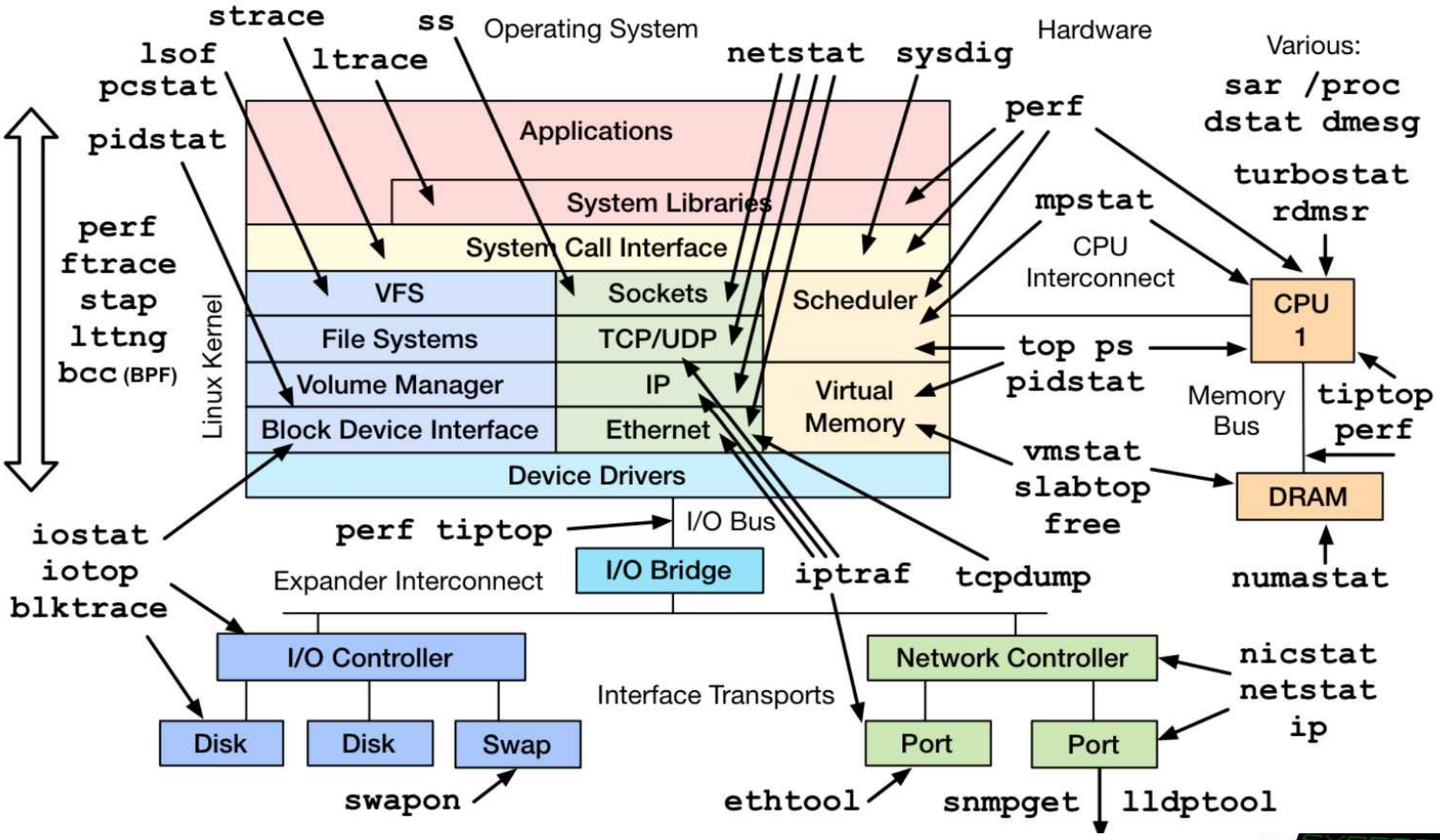
Matrix multiply version (class)	Loads per iter.	Stores per iter.	A misses per iter.	B misses per iter.	C misses per iter.	Total misses per iter.
ijk & *jik* (*AB*)	2	0	0.25	1.00	0.00	1.25
jki & *kji* (*AC*)	2	1	1.00	0.00	1.00	2.00
kij & *ikj* (*BC*)	2	1	0.00	0.25	0.25	0.50

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.



Linux Performance Monitoring Tools

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



Linux Performance Monitoring Tools

- 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

- 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

- 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

- gettimeofday()

```
choijm@embedded: ~/syspro
choijm@embedded:~/syspro$ vi elapsed_time.c
choijm@embedded:~/syspro$ gcc -o elapsed_time elapsed_time.c
choijm@embedded:~/syspro$
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

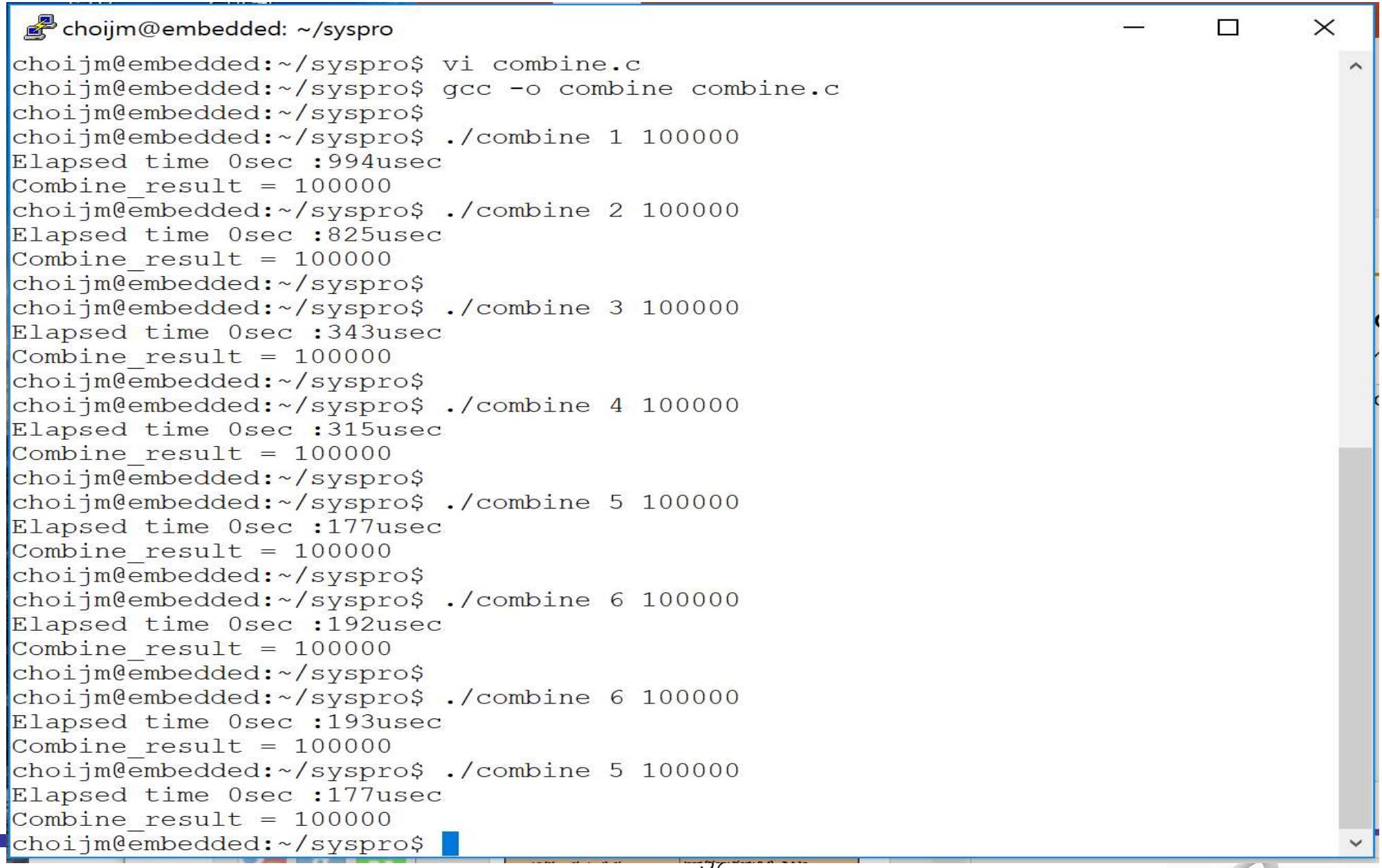
- Revisit combine(): code

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 = combine1;
        else if (choice == 2)
            func_ptr = combine2;
        else if (choice == 3)
            func_ptr = combine3;
"combine.c" [Modified] 231 lines --11%--                    27,34-37      Top
```

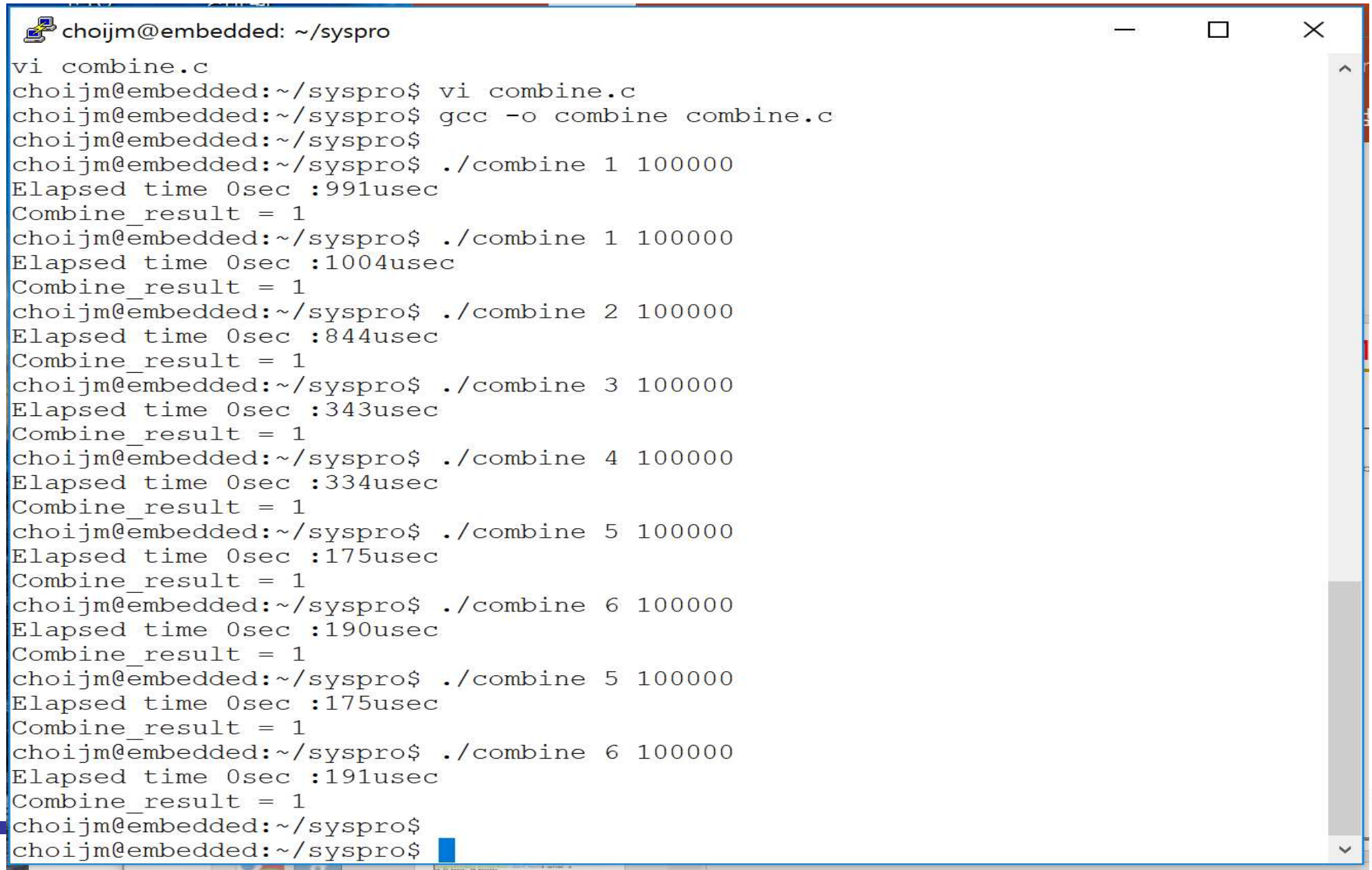
Linux Performance Monitoring Tools

- Revisit combine(): add case



Linux Performance Monitoring Tools

- Revisit combine(): multiply case



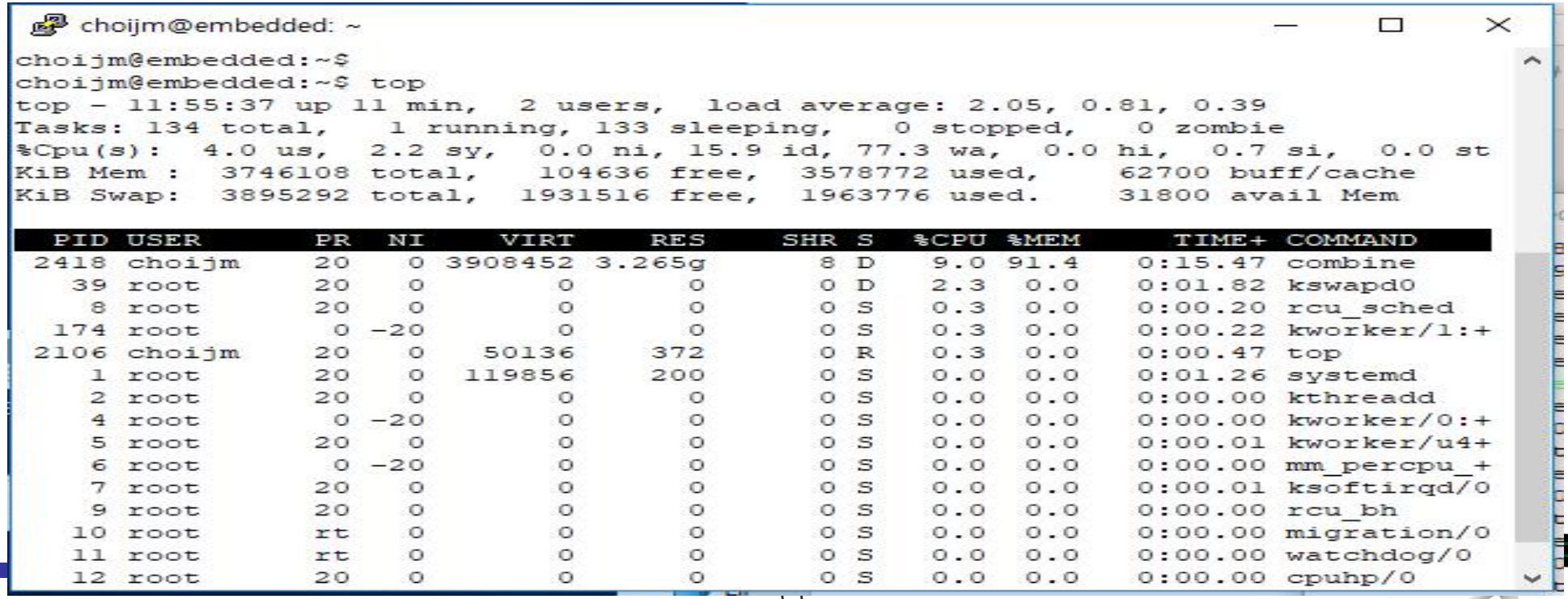
Linux Performance Monitoring Tools

- 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, "₩177ELF₩1₩1₩1₩3₩0₩0₩0₩0₩0₩0₩0₩0₩3₩0₩3₩0₩1₩0₩0₩0₩320₩207₩1₩0004₩0₩0₩0"..., 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

- 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



Linux Performance Monitoring Tools

- 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

- 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

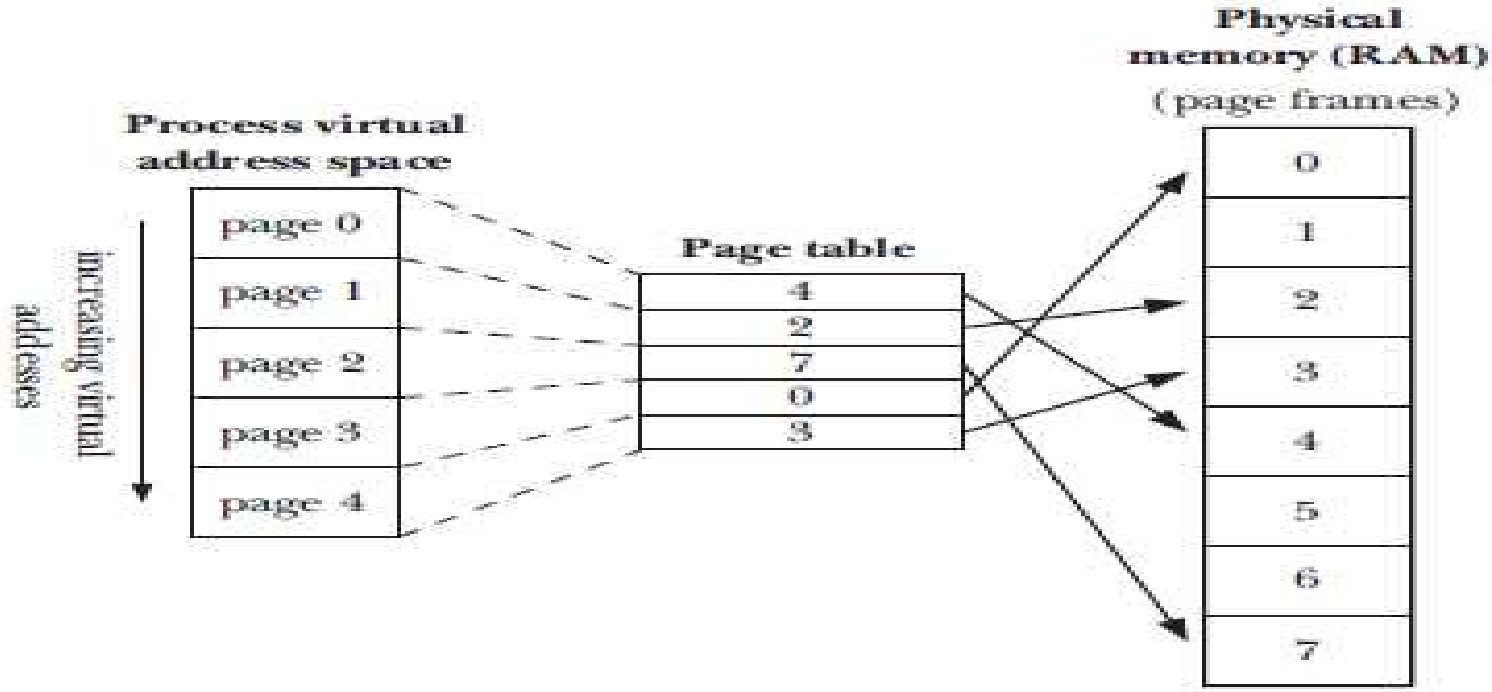
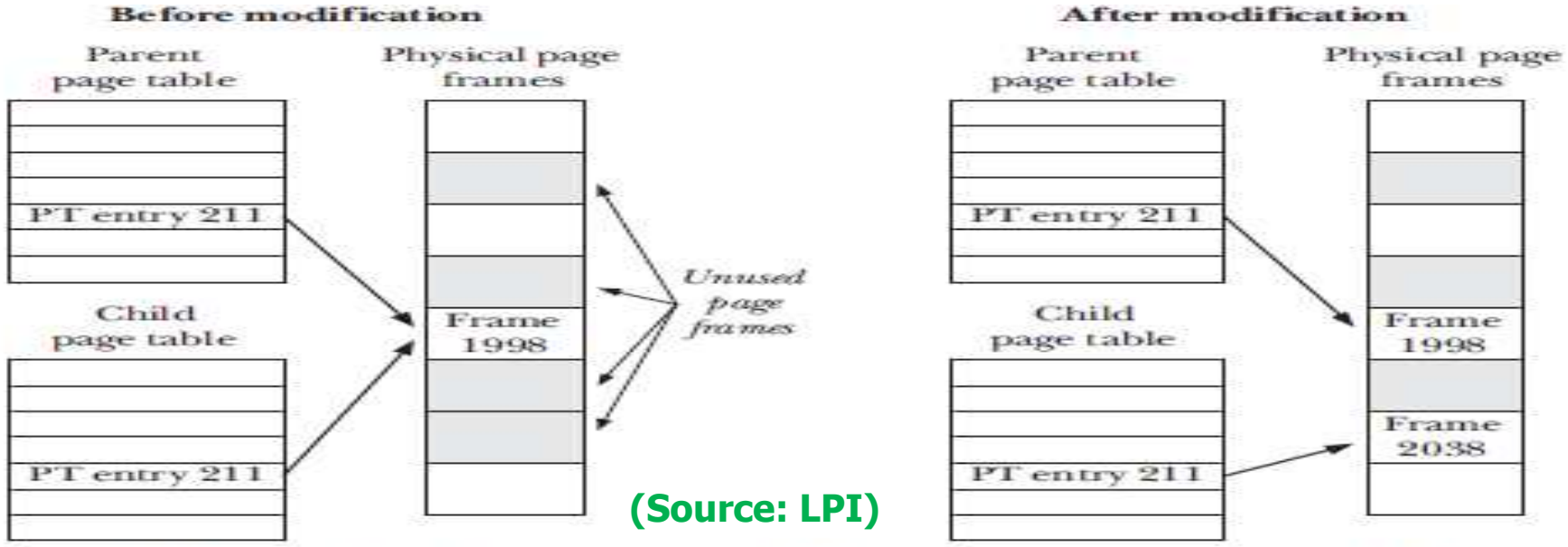


Figure 6-2: Overview of virtual memory **(Source: LPI)**

COW: memory copy optimization

- 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)



(Source: LPI)

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

Amdahl's Law

- How much can we enhance a program?
  - ✓ Execution time of an enhanced program

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

    - α : enhanced fraction
    - k : enhanced factor
  - ✓ Speed up

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

  - ✓ Implication
    - Which one is better?
      - α = 60% and k = 2 or α = 30% and k = 10
    - α = 60% and k = 2 ➔ S = 1/(0.4 + 0.3) = 1/0.7 =1.43
    - α = 30% and k = 10 ➔ 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

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 6: Performance measure and discussion**

✓ Requirements:

- Measure performance differences using 2 or more programs

- Program examples: 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)

- Measure performance using gettimeofday() or other tools

- Make a report that includes a snapshot and "discussion".

1) Upload the report to the e-Campus (pdf format!!, 17th December)

2) Send the report and source code to TA (이제연: 2reenact@naver.com)

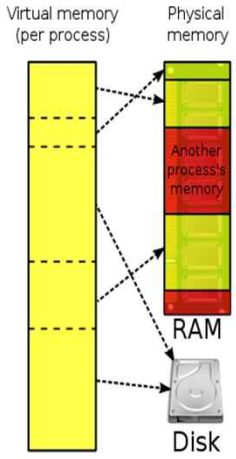
Quiz for 14th-Week 1st-Lesson

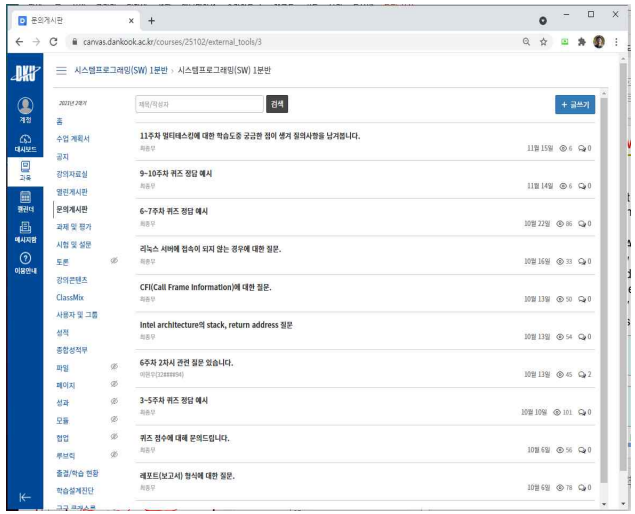
- Quiz
  - ✓ 1. The below codes are what we have discussed in LN 1. Explain the performance differences of the codes using memory hierarchy and hit ratio.
  - ✓ 2. There are many good Q&As in the 문의 게시판 of our e-learning campus. 1) Either “add a new Q&A (or opinion) that you are interested in” or “choose one existing question and add your opinion” in the 문의 게시판. 2) Then, explain your activity as the answer of this quiz.
  - ✓ Due: until 6 PM Friday of this week (10th, December)

```
/* program A */
int a[1000][1000];
int i, j;
....

for (i=0; i<1000; i++)
   for (j=0; j<1000; j++)
      a[i][j] ++;
```

VS

```
/* program B */
int a[1000][1000];
int i, j;
....

for (i=0; i<1000; i++)
   for (j=0; j<1000; j++)
      a[j][i] ++;
```





(Source: LN 1 What is System programming?)

Appendix: Memory Hierarchy and Performance

- Cache performance

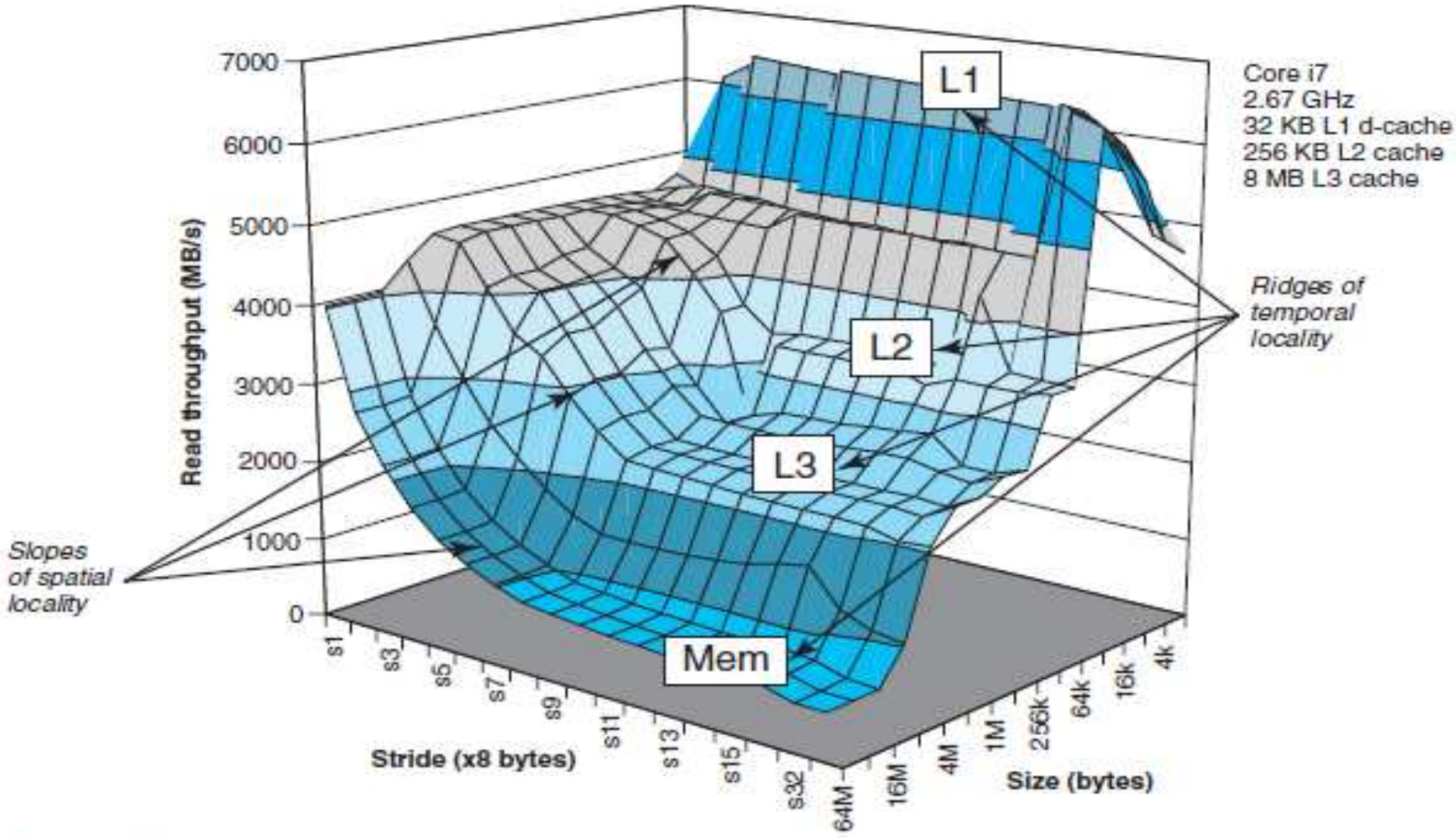


Figure 6.43 **The memory mountain.**

Appendix: SIMD effect

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

	Integer		Floating point		
Method	+	*	+	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.