

The Memory Hierarchy & Cache Memories & Program optimization

李天宇 段棋怀 许珈铭

2023.11.1

Memory (CS:APP Ch. 6)

李天宇

Contents

- Storage Technologies
- Locality
- The Memory Hierarchy
- Cache Memories
- Writing Cache-Friendly Code
- Putting It Together: The Impact of Caches on Program Performance

Memory & Storage

- e.g. 对于32G + 2T的计算机，32G是其内存(memory)，而2T指硬盘的存储(storage)
- 所谓“运行内存”是由对内存的错误解读造出的词汇，实际上是一种错误的说法
- 不过如果说自己电脑2T内存的时候先不要急着笑，万一人家真的有呢



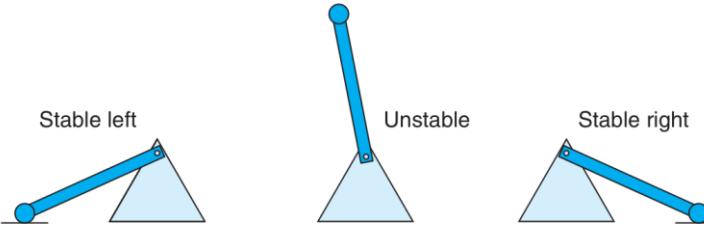
Storage Technologies

- Random Access Memory (RAM)
- Disk Storage
- Solid State Disks (SSD)
- Storage Technology Trends

RAM

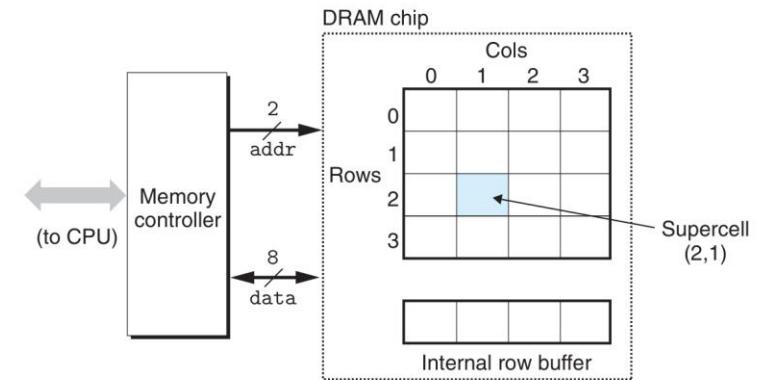
- Static RAM (SRAM)

Figure 6.1
Inverted pendulum.
Like an SRAM cell, the
pendulum has only two
stable configurations, or
states.



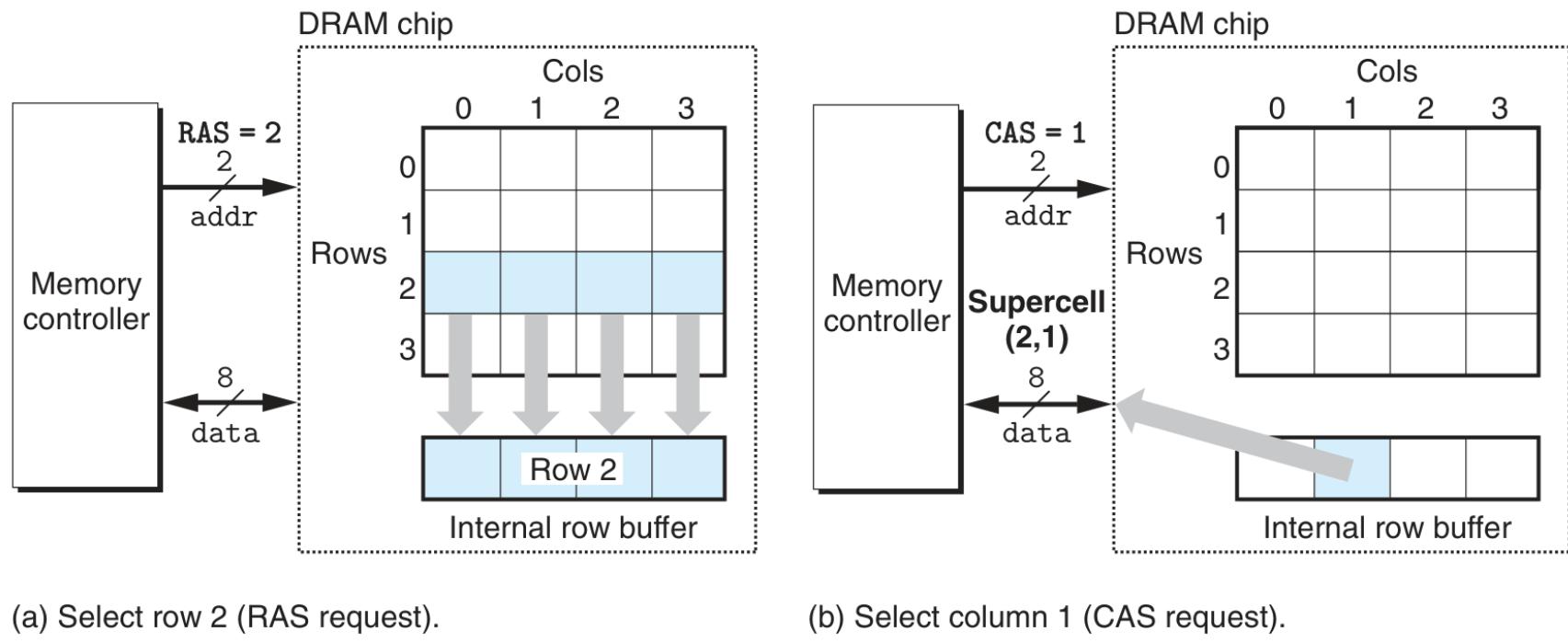
- Dynamic RAM (DRAM)

Figure 6.3
High-level view of a
128-bit 16×8 DRAM
chip.



	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative cost	Applications
SRAM	6	1×	Yes	No	1,000×	Cache memory
DRAM	1	10×	No	Yes	1×	Main memory, frame buffers

Conventional DRAM



(a) Select row 2 (RAS request).

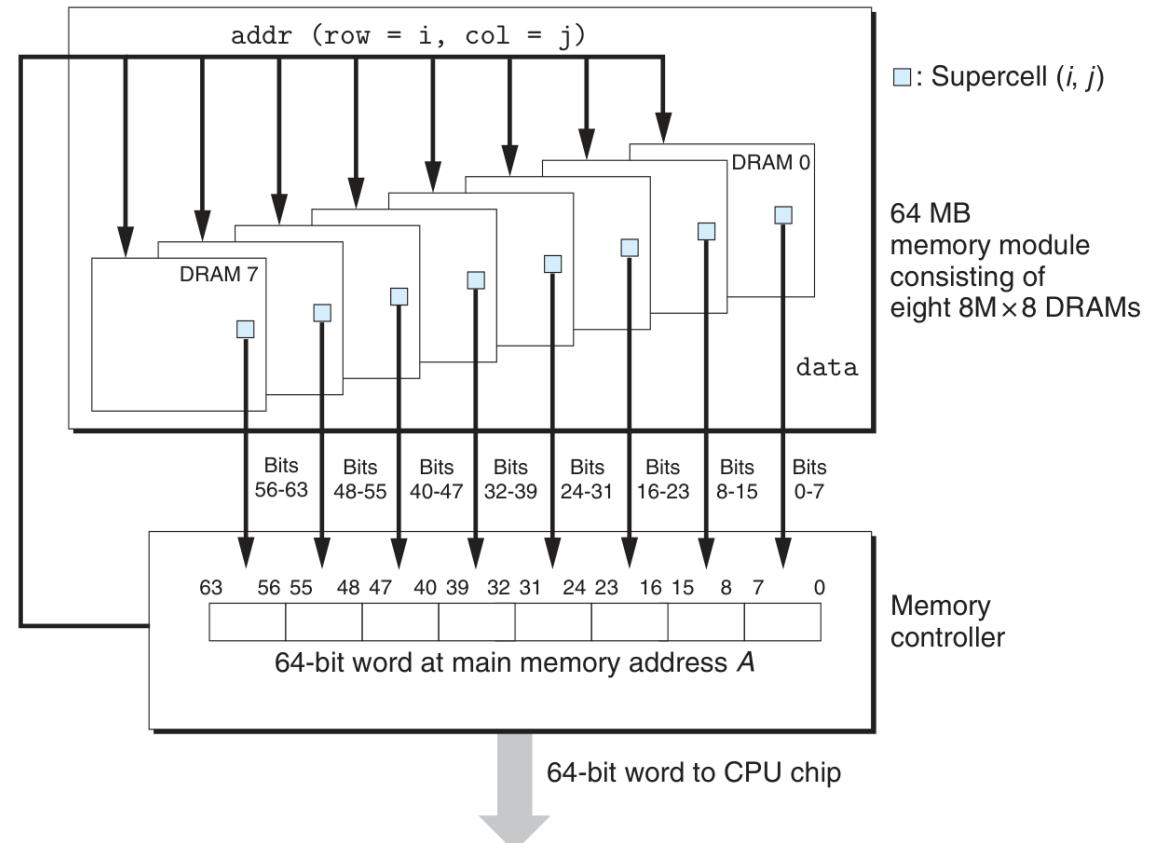
(b) Select column 1 (CAS request).

Figure 6.4 Reading the contents of a DRAM supercell.

Reading the Contents of A Memory Module

Figure 6.5

Reading the contents of a memory module.



Enhance DRAMs

- Fast Page Mode DRAM (FPM DRAM)
- Extended Data Out DRAM (EDO DRAM)
- Synchronous DRAM (SDRAM)
- Double Data-Rate Synchronous DRAM (DDR SDRAM): DDR, DDR2, DDR3
- Video RAM (VRAM)

Nonvolatile Memory: Read-Only Memory (ROM)

- Programmable ROM (PROM)

can be programmed exactly once (Real ROM)

- Erasable Programmable ROM (EPROM)

can be erased and reprogrammed on the order of 1,000 times

- Electrically Erasable PROM (EEPROM)

can be reprogrammed on the order of 10^5 times

- Flash Memory

A new form of flash-based disk drive: Solid State Disk

- Firmware: Programs stored in ROM, e.g. a PC's BIOS

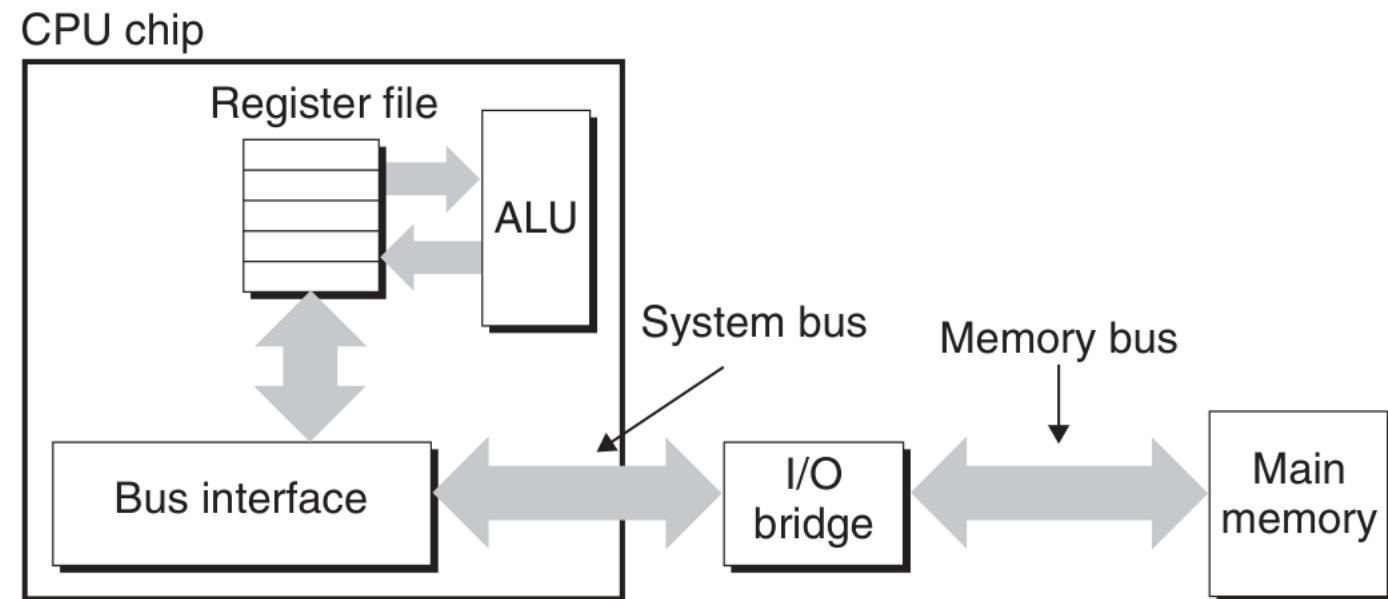
(固件升级原来是这个意思



Accessing Main Memory

- Bus & Bus Transaction

Figure 6.6
Example bus structure
that connects the CPU
and main memory.

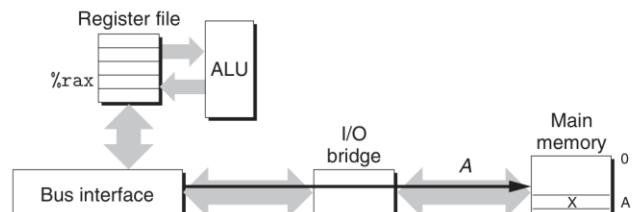


Accessing Main Memory

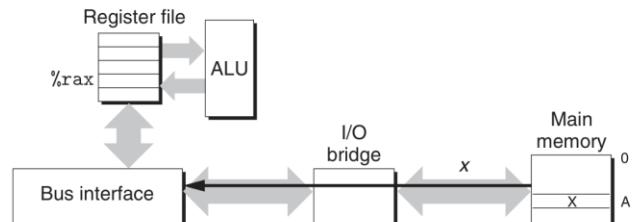
- Read Transaction & Write Transaction

Figure 6.7

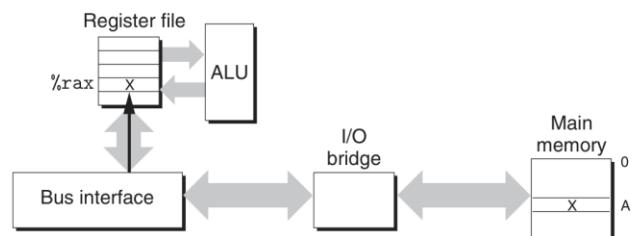
Memory read transaction for a load operation: `movq A, %rax`.



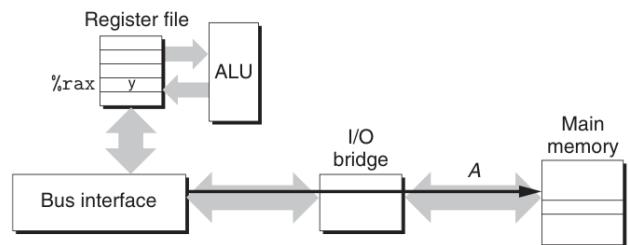
(a) CPU places address `A` on the memory bus.



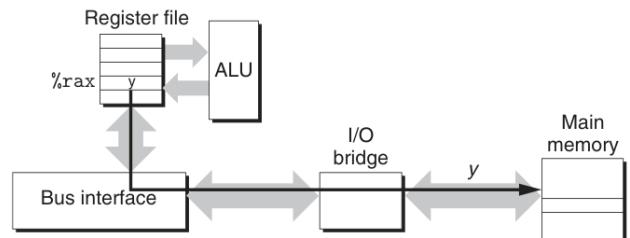
(b) Main memory reads `A` from the bus, retrieves word `x`, and places it on the bus.



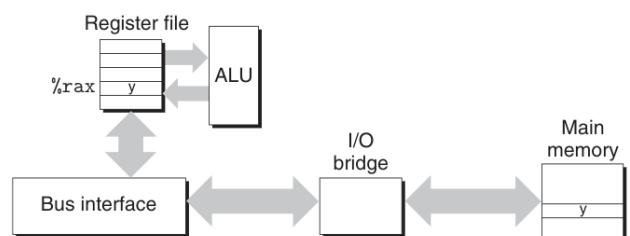
(c) CPU reads word `x` from the bus, and copies it into register `%rax`.



(a) CPU places address `A` on the memory bus. Main memory reads it and waits for the data word.



(b) CPU places data word `y` on the bus.



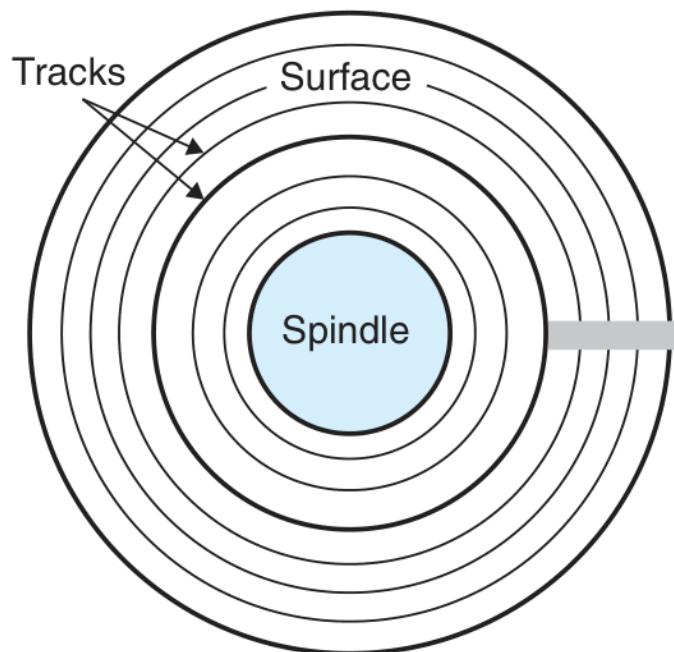
(c) Main memory reads data word `y` from the bus and stores it at address `A`.

Figure 6.8 Memory write transaction for a store operation: `movq %rax, A`.

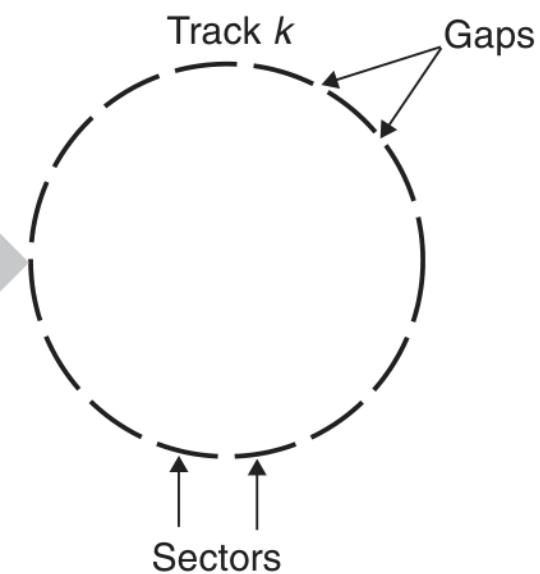
Disk Storage

- Disk Geometry
- Disk Capacity
- Disk Operation
- Logical Disk Blocks
- Connecting I/O Devices
- Accessing Disks

Disk Geometry



(a) Single-platter view



(b) Multiple-platter view

Figure 6.9 Disk geometry.

Disk Capacity

- Recording density (*bits/in*)

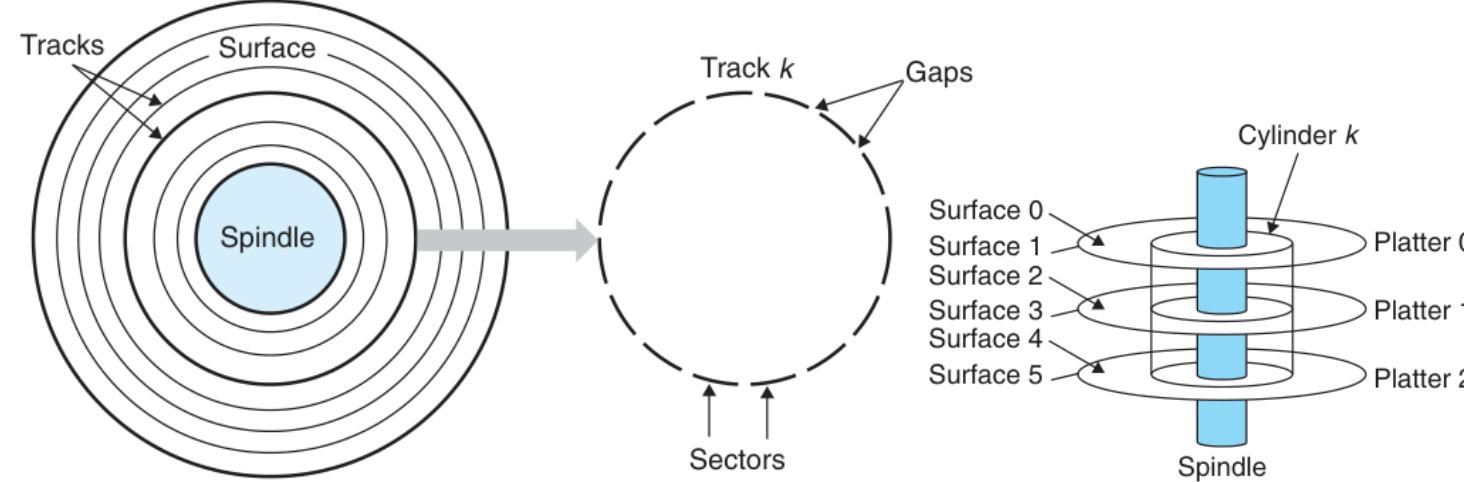
The number of bits that can be squeezed into a 1-inch segment of a track.

- Track density (*tracks/in*)

The number of tracks that can be squeezed into a 1-inch segment of the radius extending from the center of the platter.

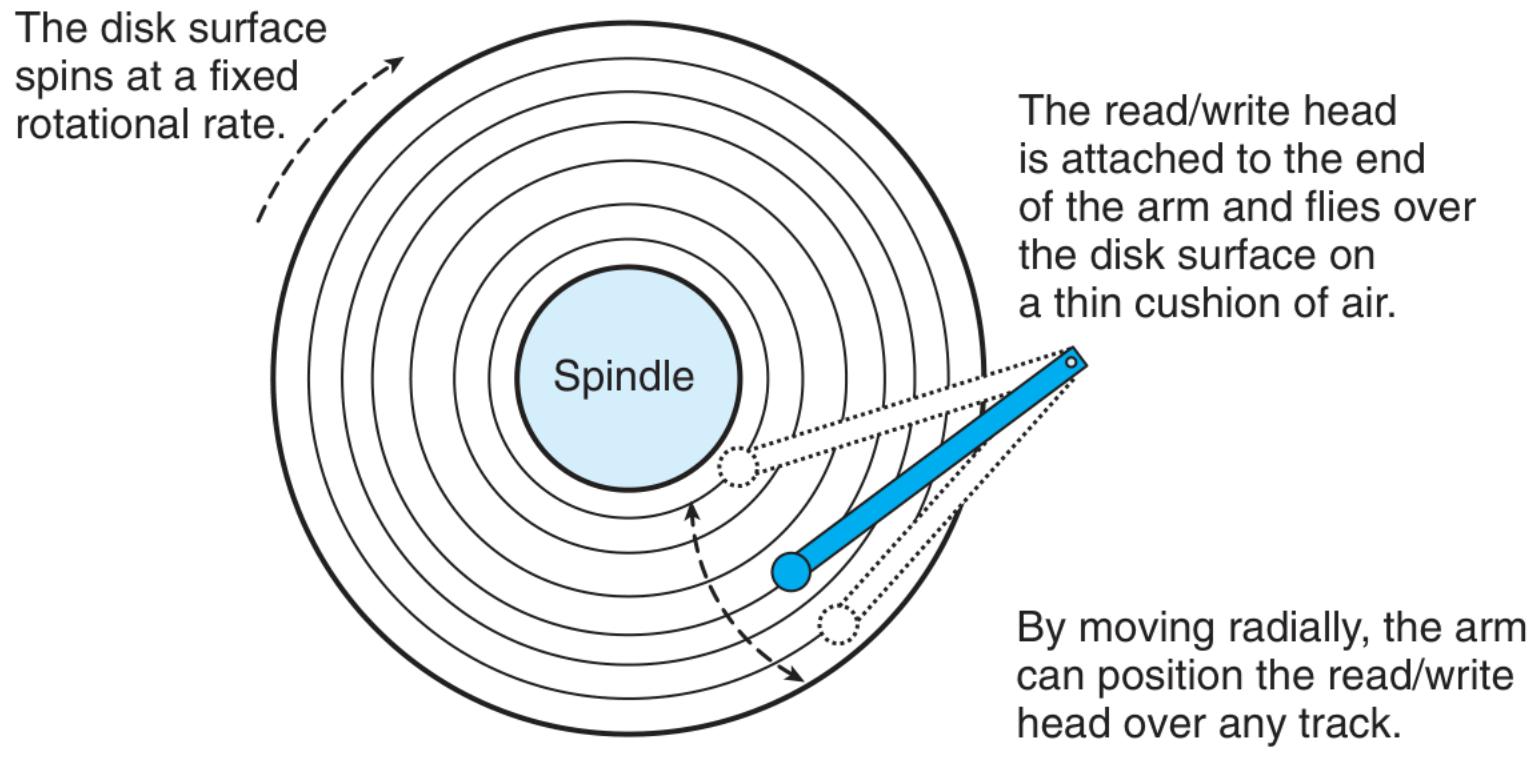
- Areal density (*bits/in²*).

The product of the recording density and the track density.

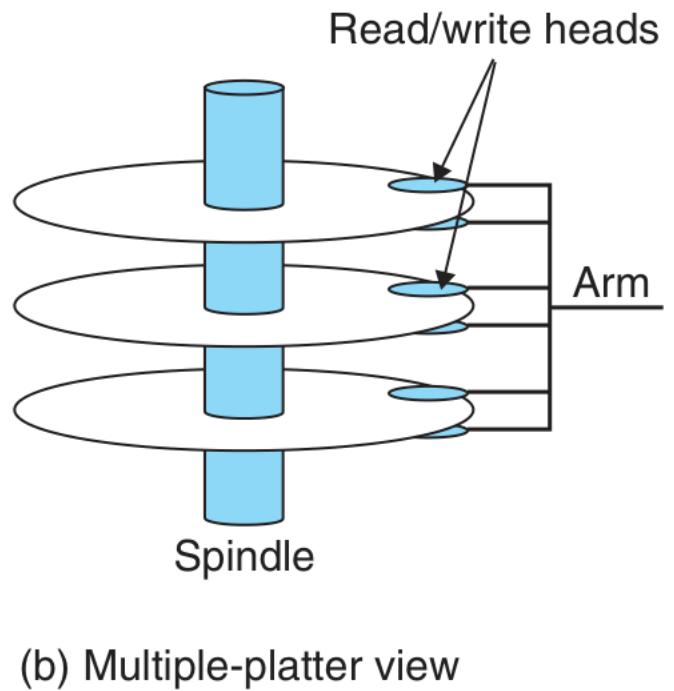


$$\text{Capacity} = \frac{\# \text{ bytes}}{\text{sector}} \times \frac{\text{average } \# \text{ sectors}}{\text{track}} \times \frac{\# \text{ tracks}}{\text{surface}} \times \frac{\# \text{ surfaces}}{\text{platter}} \times \frac{\# \text{ platters}}{\text{disk}}$$

Disk Operation



(a) Single-platter view



(b) Multiple-platter view

Figure 6.10 Disk dynamics.

Disk Operation: Access Time

- Seek Time (T_{seek})

$$T_{avg\ rotation} \approx T_{seek} \gg T_{avg\ transfer}$$

$$T_{access\ time} \approx 2 \times T_{seek}$$

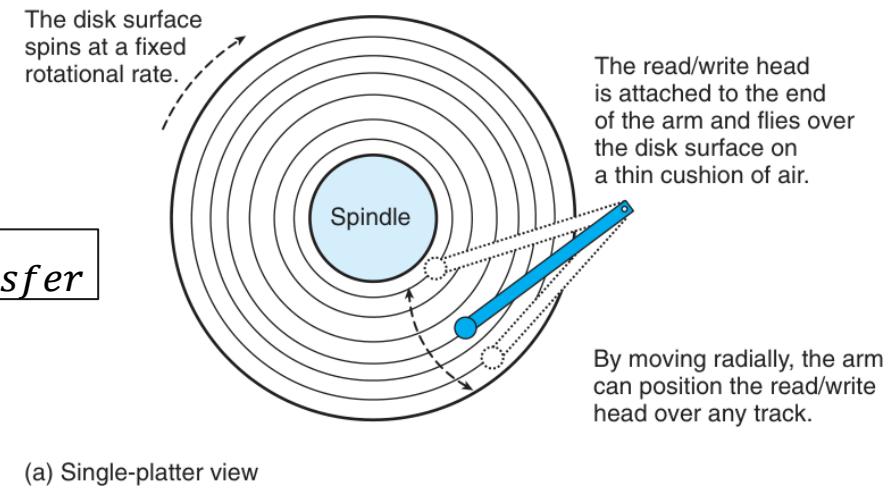
- Rotation Latency ($T_{rotation}$)

$$T_{max\ rotation} = \frac{1}{RPM} \times \frac{60s}{1min}, T_{avg\ rotation} = \frac{1}{2} T_{max\ rotation}$$

- Transfer Time (T_{access})

$$T_{avg\ transfer} = \frac{1}{RPM} \times \frac{1}{(average\ #\ sectors/track)} \times \frac{60secs}{1\ min}$$

RPM: Rotations Per Minute



(a) Single-platter view

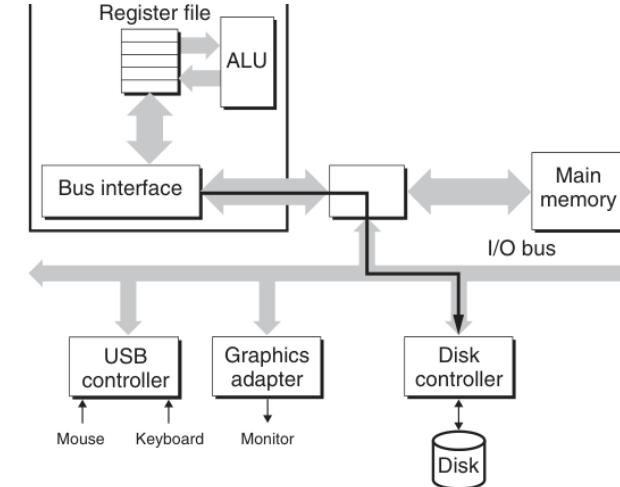
Connecting I/O Devices

Devices attached to the bus:

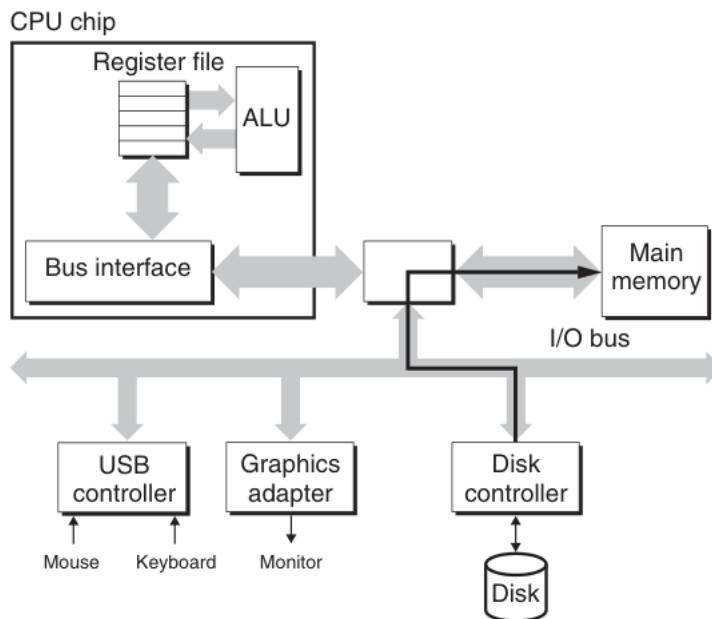
- Universal Serial Bus (USB) Controller
- Graphics Card (or Adapter)
- Host Bus Adapter: Connects one or more disks to the I/O Bus
SCSI/SATA

Accessing Disks

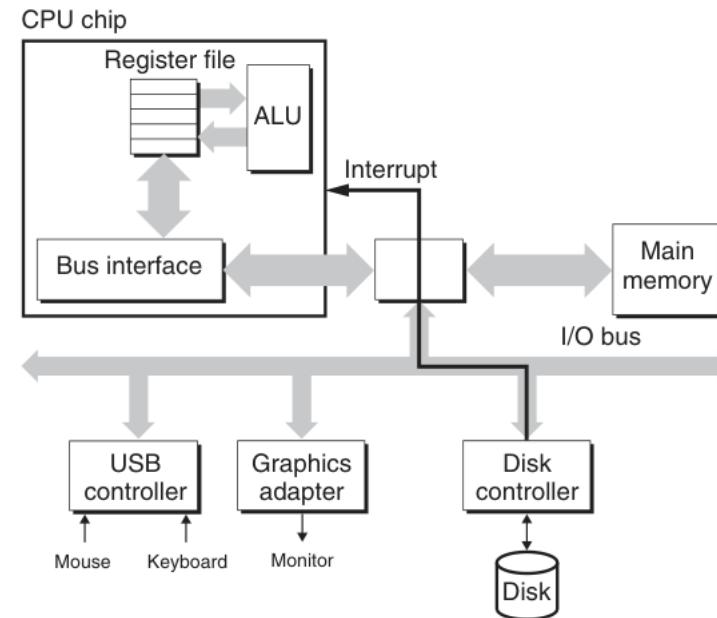
Reading a disk sector.



(a) The CPU initiates a disk read by writing a command, logical block number, and destination memory address to the memory-mapped address associated with the disk.



(b) The disk controller reads the sector and performs a DMA transfer into main memory.



(c) When the DMA transfer is complete, the disk controller notifies the CPU with an interrupt.

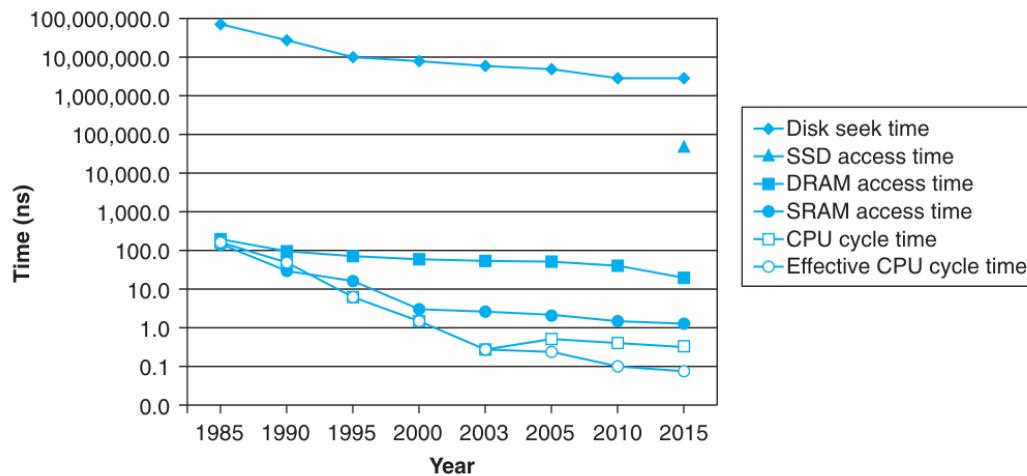
Solid State Disk (SSD)

- 读SSD比写要快：数据是以页为单位进行读写的，只有在一页所属的块整个被擦除之后，才可以写这一页（通常是所有位置1）



Storage Technology Trends

- 不同的存储技术有不同的价格和性能折中
- 不同存储技术的价格和性能属性以截然不同的速率变化着
- 对于内存和磁盘技术，增加密度比降低访问时间容易得多
- DRAM和磁盘的性能滞后于CPU的性能



高速缓存、局部性的重要性

Figure 6.16 The gap between disk, DRAM, and CPU speeds.

Locality

- Temporal locality (时间局部性) & Spatial locality (空间局部性)
- 有良好局部性的程序比局部性差的程序运行得更快

Locality of References to Program Data

```
1 int sumvec(int v[N])
2 {
3     int i, sum = 0;
4
5     for (i = 0; i < N; i++)
6         sum += v[i];
7
8     return sum;
}
```

(a)

Address	0	4	8	12	16	20	24	28
Contents	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Access order	1	2	3	4	5	6	7	8

(b)

Figure 6.17 (a) A function with good locality. (b) Reference pattern for vector v ($N = 8$). Notice how the vector elements are accessed in the same order that they are stored in memory.

Stride-1 reference pattern (sequential reference pattern)

Locality of References to Program Data

```
1 int sumarrayrows(int a[M] [N])
2 {
3     int i, j, sum = 0;
4
5     for (i = 0; i < M; i++)
6         for (j = 0; j < N; j++)
7             sum += a[i] [j];
8
9 }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	2	3	4	5	6

(b)

Stride-1 reference pattern



Figure 6.18 (a) Another function with good locality. (b) Reference pattern for array a ($M = 2, N = 3$). There is good spatial locality because the array is accessed in the same row-major order in which it is stored in memory.

```
1 int sumarraycols(int a[M] [N])
2 {
3     int i, j, sum = 0;
4
5     for (j = 0; j < N; j++)
6         for (i = 0; i < M; i++)
7             sum += a[i] [j];
8
9 }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	3	5	2	4	6

(b)

Stride-N reference pattern



Figure 6.19 (a) A function with poor spatial locality. (b) Reference pattern for array a ($M = 2, N = 3$). The function has poor spatial locality because it scans memory with a stride- N reference pattern.

Summary of Locality

- 重复引用相同变量的程序有良好的时间局部性
- 对于具有步长为 k 的引用模式的程序，步长越小，空间局部性就越好。具有步长为1的引用模式的程序有很好的空间局部性。在内存中以大步长跳来跳去的程序空间局部性会很差
- 对于取指令来说，循环有好的时间和空间局部性。循环体越小，循环迭代次数越多，局部性越好

The Memory Hierarchy

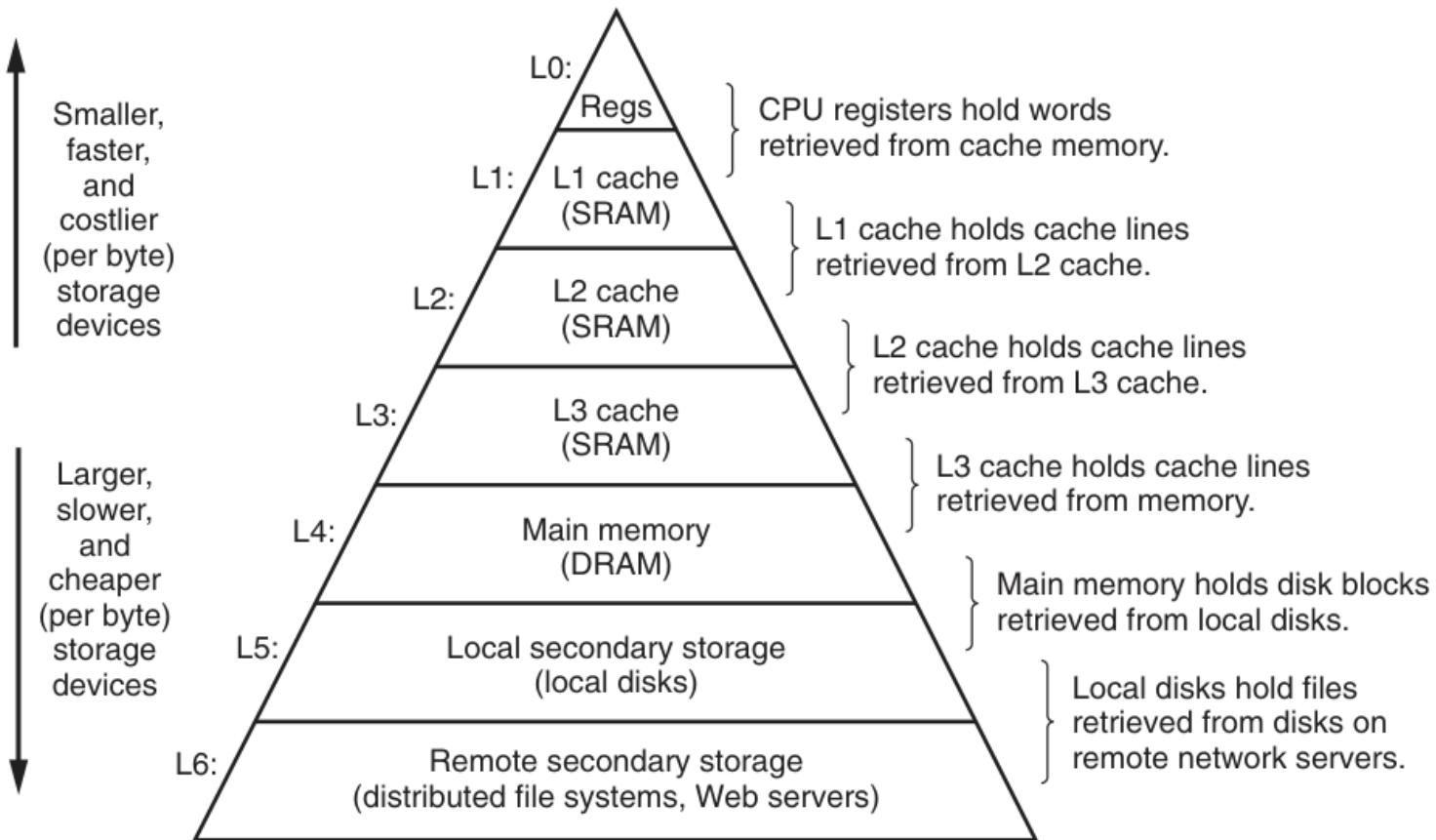


Figure 6.21 The memory hierarchy.

Caching in the Memory Hierarchy

- 高速缓存(cache)是一个小而快速的存储设备，作为存储在更大也更慢的设备中的数据对象的缓冲区域（相对概念）
- 使用高速缓存的过程称为缓存(caching)

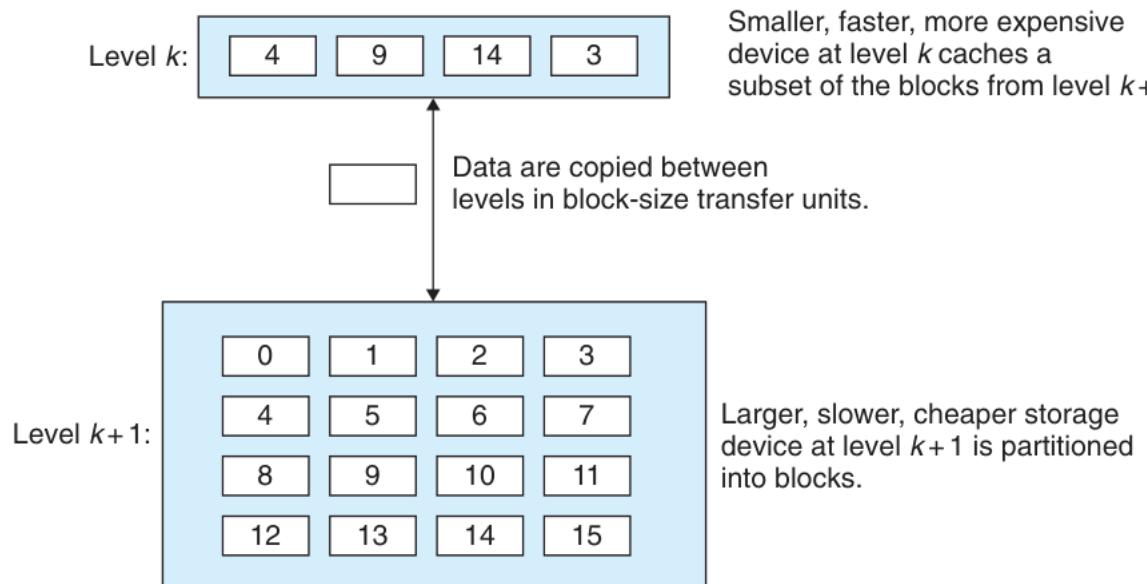


Figure 6.22 The basic principle of caching in a memory hierarchy.

Caching in the Memory Hierarchy

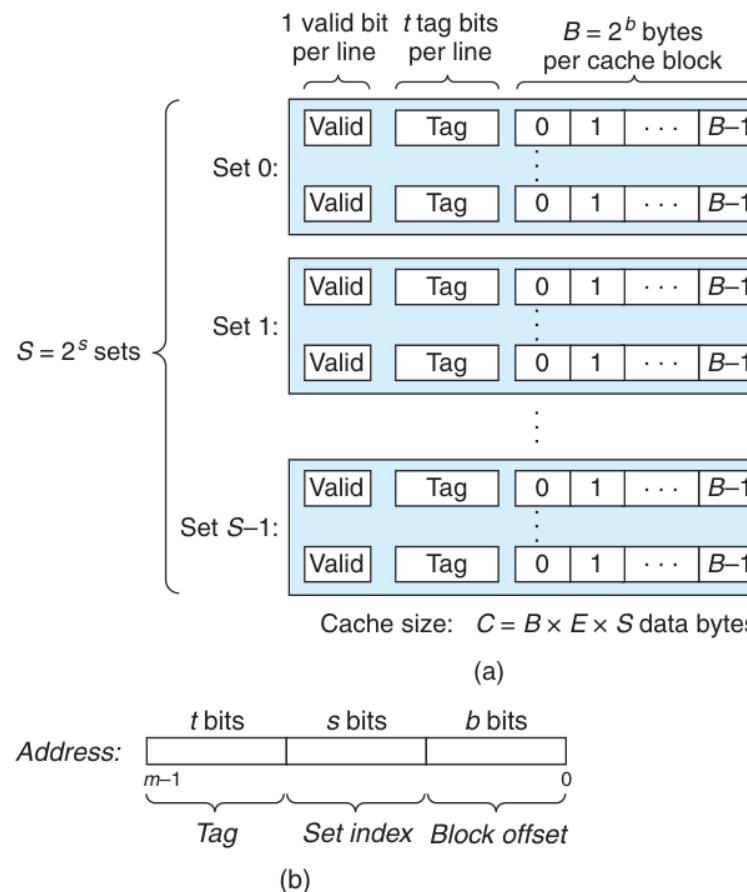
- Cache Hits
- Cache Misses
- Kinds of Cache Misses
 - Compulsory Miss/Cold Miss & Conflict Miss
- Cache Management
 - 注意寄存器是由编译器管理的

Generic Cache Memory Organization

Figure 6.25

General organization of cache (S, E, B, m).

(a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.



Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of set index bits
$b = \log_2(B)$	Number of block offset bits
$t = m - (s + b)$	Number of tag bits
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

Figure 6.26 Summary of cache parameters.

Direct-Mapped Caches

Figure 6.27
Direct-mapped cache
($E = 1$). There is exactly
one line per set.

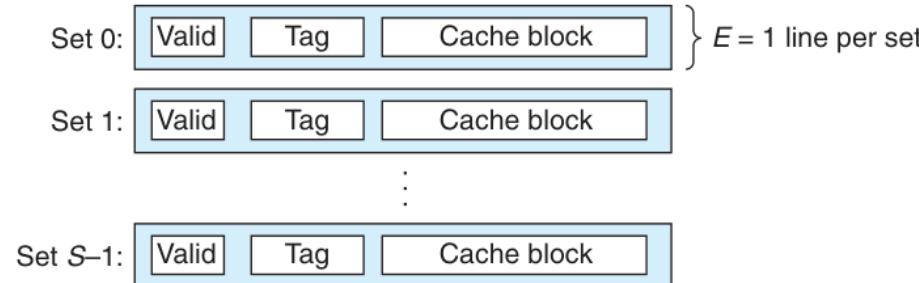
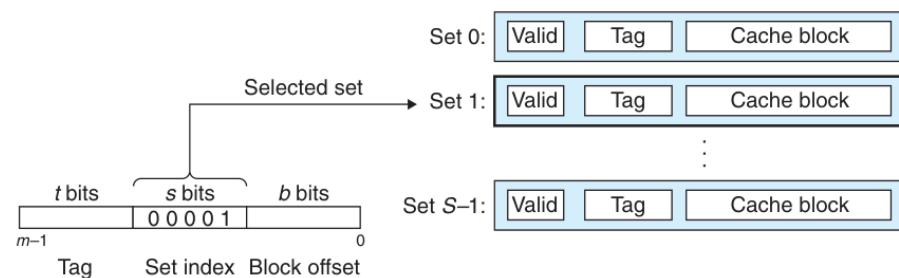
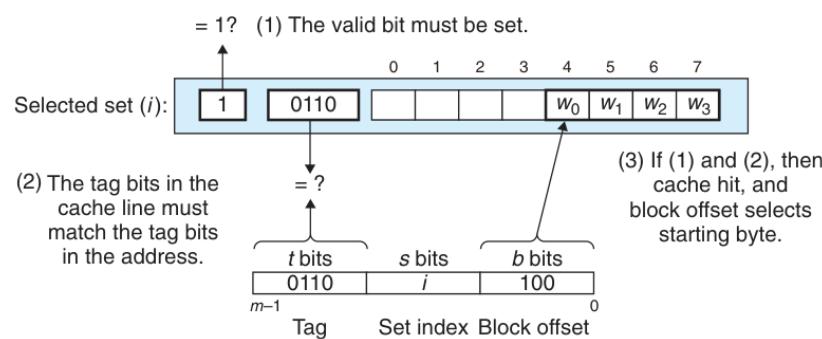


Figure 6.28
**Set selection in a direct-
mapped cache.**



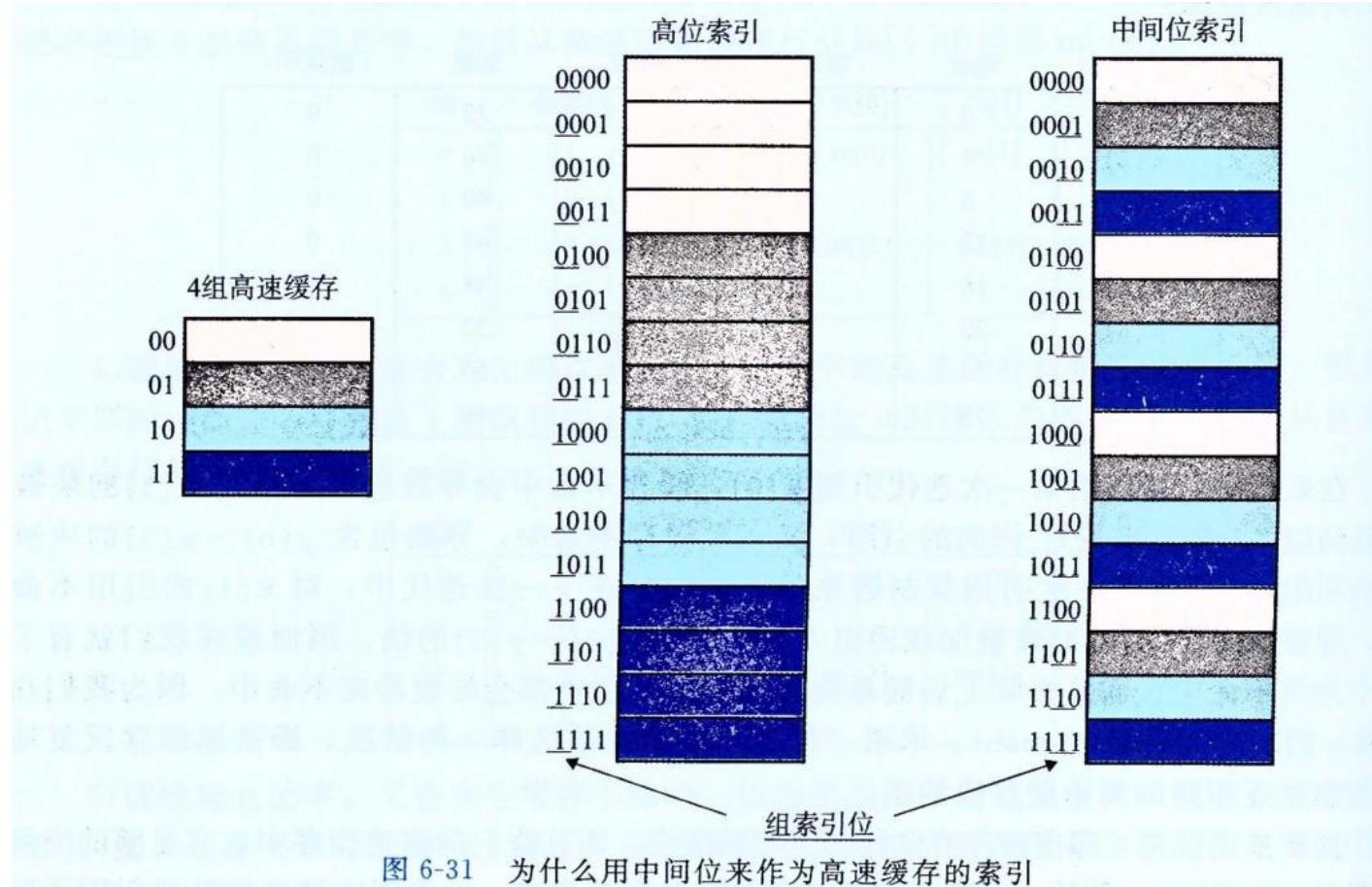
组选择

Figure 6.29
**Line matching and word
selection in a direct-
mapped cache.** Within the
cache block, w_0 denotes
the low-order byte of the
word w , w_1 the next byte,
and so on.



行匹配、字选择

Why Caches Index With the Middle Bits



Conflict Misses in Direct-Mapped Caches

- 抖动(Thrash): 高速缓存反复地加载和驱逐相同的高速缓存块的组

```
1 float dotprod(float x[8], float y[8])
2 {
3     float sum = 0.0;
4     int i;
5
6     for (i = 0; i < 8; i++)
7         sum += x[i] * y[i];
8     return sum;
9 }
```

Element	Address	Set index	Element	Address	Set index
x[0]	0	0	y[0]	32	0
x[1]	4	0	y[1]	36	0
x[2]	8	0	y[2]	40	0
x[3]	12	0	y[3]	44	0
x[4]	16	1	y[4]	48	1
x[5]	20	1	y[5]	52	1
x[6]	24	1	y[6]	56	1
x[7]	28	1	y[7]	60	1

Element	Address	Set index	Element	Address	Set index
x[0]	0	0	y[0]	48	1
x[1]	4	0	y[1]	52	1
x[2]	8	0	y[2]	56	1
x[3]	12	0	y[3]	60	1
x[4]	16	1	y[4]	64	0
x[5]	20	1	y[5]	68	0
x[6]	24	1	y[6]	72	0
x[7]	28	1	y[7]	76	0



Set Associative Caches

Figure 6.32

Set associative cache ($1 < E < C/B$). In a set associative cache, each set contains more than one line. This particular example shows a two-way set associative cache.

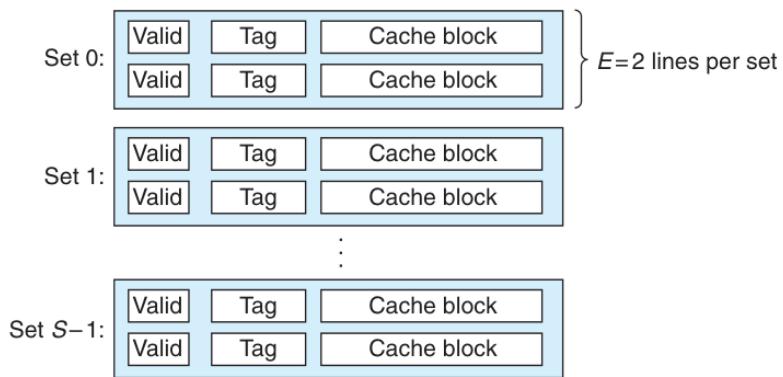


Figure 6.33

Set selection in a set associative cache.

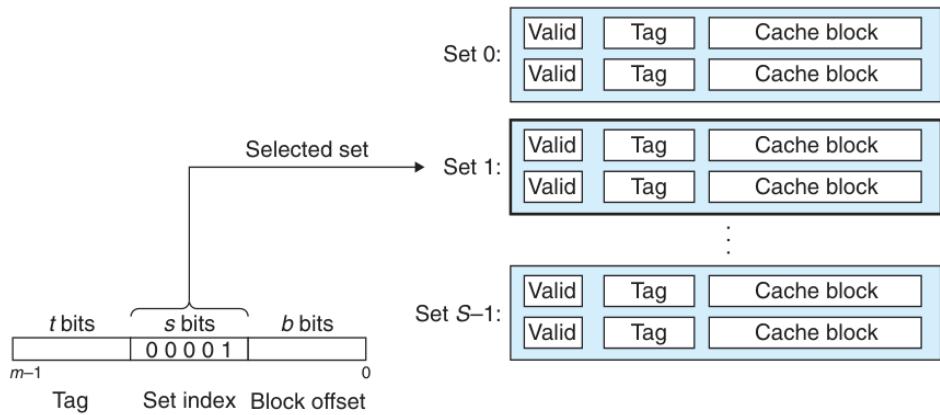
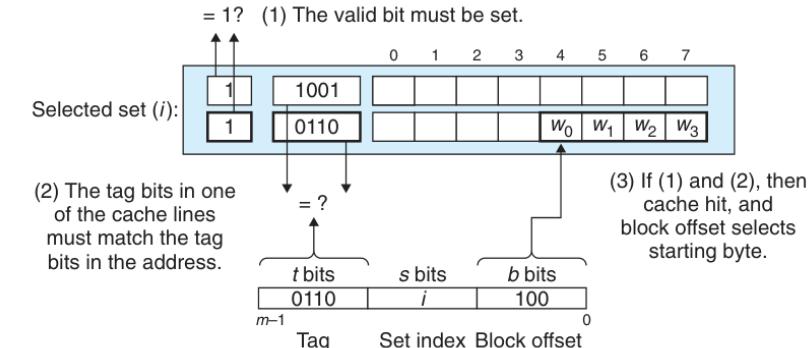


Figure 6.34

Line matching and word selection in a set associative cache.



Line Replacement on Misses in Set Associative Caches

- Least-Frequently-Used (LFU) policy
- Least-Recently-Used (LRU) policy



Fully Associative Caches

Figure 6.35
Fully associative cache ($E = C/B$). In a fully associative cache, a single set contains all of the lines.

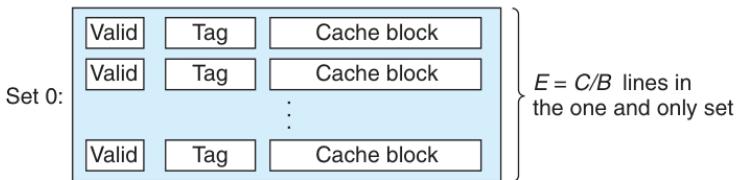


Figure 6.36
Set selection in a fully associative cache. Notice that there are no set index bits.

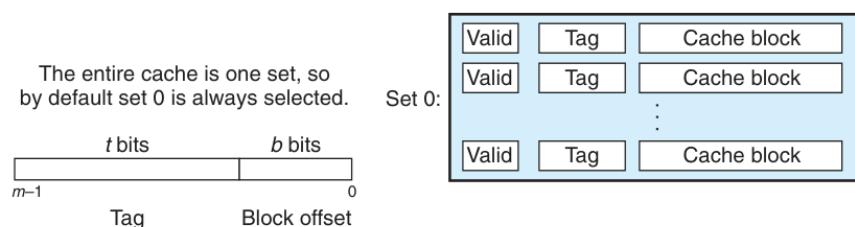
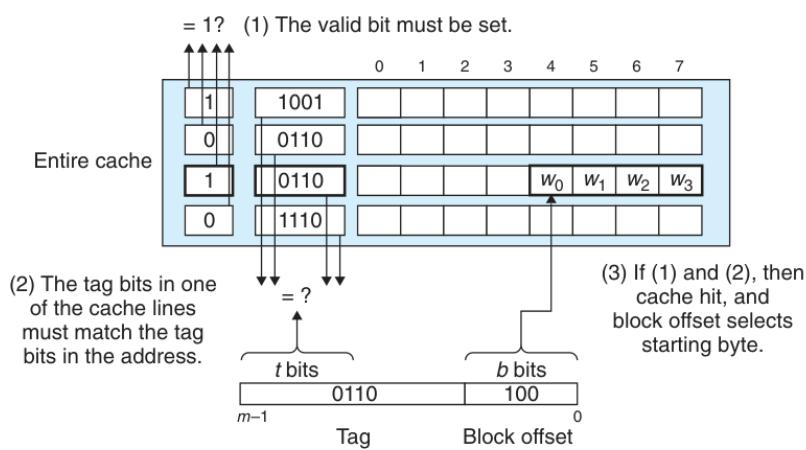


Figure 6.37
Line matching and word selection in a fully associative cache.



那么，全相联是不是最优选择呢？



相比于组相联，全相联和直接映射各自有什么问题？

直接映射：Thrash

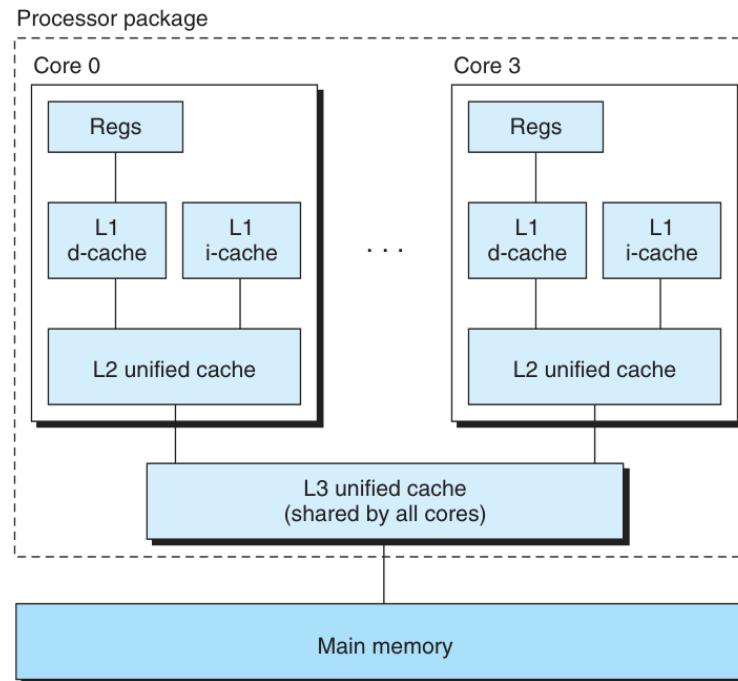
全相联：行匹配时间过长

Issues with Writes

- Write Hit: write-through & write-back (needs a dirty bit for each cache line)
- Write Miss: write-allocate & not-write-allocate
- 在编写程序时可假定高速缓存使用写回和写分配

Anatomy of a Real Cache Hierarchy

Figure 6.38
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	10	256 KB	8	64 B	512
L3 unified cache	40–75	8 MB	16	64 B	8,192

Figure 6.39 Characteristics of the Intel Core i7 cache hierarchy.

Performance Impact of Cache Parameters

- 不命中率(miss rate): $\frac{\# \text{ misses}}{\# \text{ references}}$
- 命中率(hit rate): $1 - \text{miss rate}$
- 命中时间(hit time): 包括组选择、行选择、字选择时间
- 不命中处罚(miss penalty): 由于不命中所需要的额外的时间

Writing Cache-Friendly Code

- 让最常见的情况运行得快（关注内循环）
 - 尽量减小每个循环内部的缓存不命中数量
-
- 对局部变量的反复引用是好的
 - 步长为1的引用模式是好的

Putting It Together: The Impact of Caches on Program Performance

- The Memory Mountain

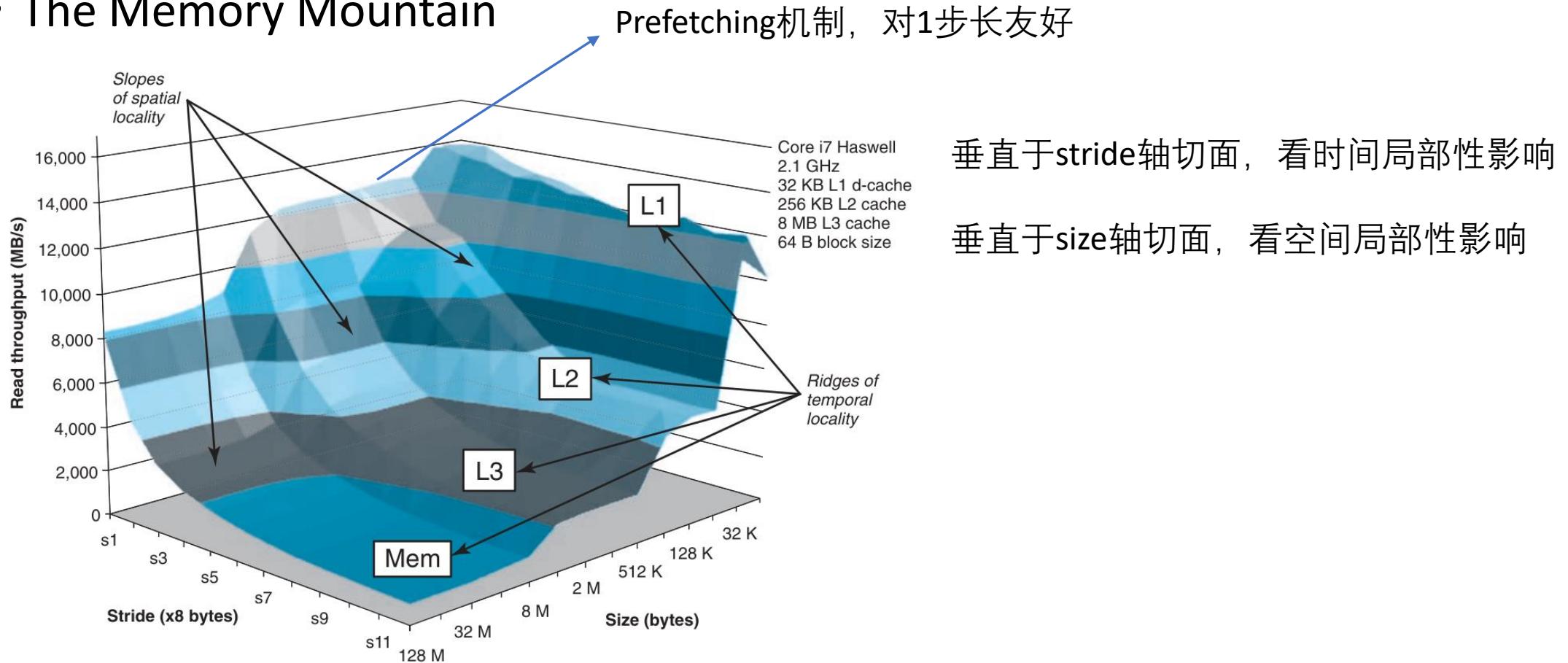


Figure 6.41 A memory mountain. Shows read throughput as a function of temporal and spatial locality.

Writing Cache-Friendly Code 实操

- Rearranging Loops to Increase Spatial Locality

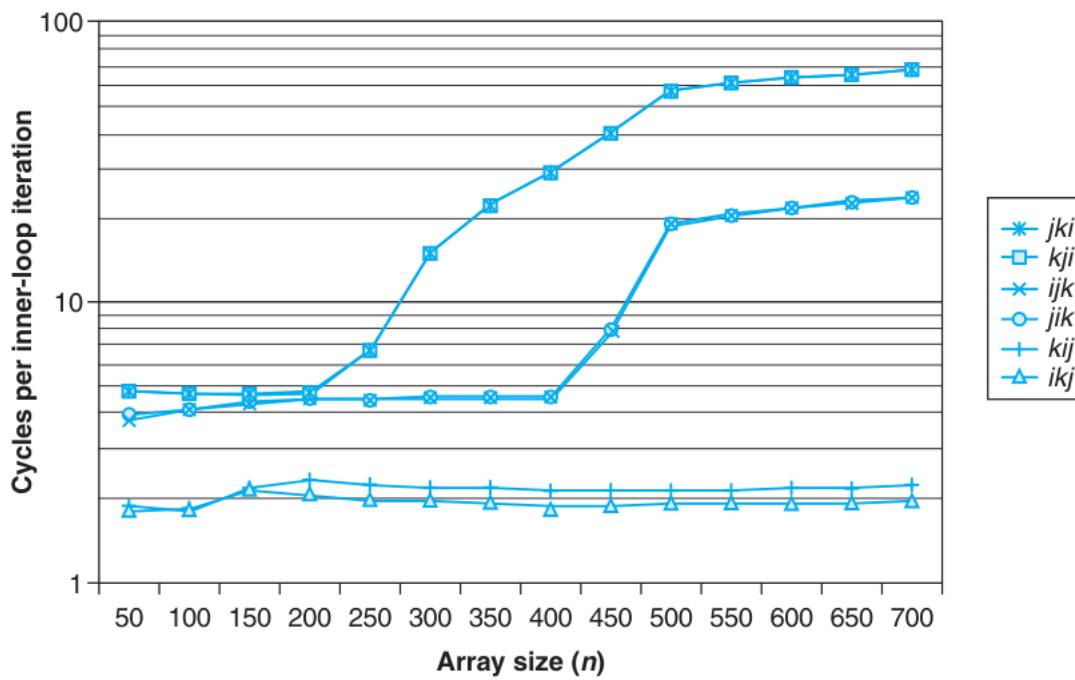


Figure 6.46 Core i7 matrix multiply performance.

(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 $jk i$ — 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     }
```

Figure 6.44 Six versions of matrix multiply. Each version is uniquely identified by the ordering of its loops.

Optimization (CS:APP Ch. 5)

段棋怀

Practice

李天宇

5.1 优化编译器的能力和局限性

编译器必须很小心地对程序只使用安全的优化，保证程序运行的结果与原先一致，这就要求程序员必须显式地写出编译器能有效转换成机器代码的程序

- 两个主要的妨碍优化的因素：
- 1、内存别名使用
- 2、函数调用

内存别名使用：

右侧的两个函数，看上去是相同的行为，twiddle2只进行3次内存引用，而1进行6次

但当xp==yp时，两者的结果不同。

当编译器不能确定两个指针是否指向同一个位置时，就不做优化。

```
1 void twiddle1(long *xp, long *yp)
2 {
3     *xp += *yp;
4     *xp += *yp;
5 }
6
7 void twiddle2(long *xp, long *yp)
8 {
9     *xp += 2* *yp;
10 }
```

函数调用：

当每次函数调用会造成全局程序状态的修改时，就会有“副作用”，
不能进行func2的优化

```
1 long f();
2
3 long func1() {
4     return f() + f() + f() + f();
5 }
6
7 long func2() {
8     return 4*f();
9 }
```

```
1 long counter = 0;
2
3 long f() {
4     return counter++;
5 }
```

程序性能：

- 每元素周期数 (CPE)：每计算一个元素所花费的时间周期数

```
1  /* Compute prefix sum of vector a */
2  void psum1(float a[], float p[], long n)
3  {
4      long i;
5      p[0] = a[0];
6      for (i = 1; i < n; i++)
7          p[i] = p[i-1] + a[i];
8  }
9
10 void psum2(float a[], float p[], long n)
11 {
12     long i;
13     p[0] = a[0];
14     for (i = 1; i < n-1; i+=2) {
15         float mid_val = p[i-1] + a[i];
16         p[i] = mid_val;
17         p[i+1] = mid_val + a[i+1];
18     }
19     /* For even n, finish remaining element */
20     if (i < n)
21         p[i] = p[i-1] + a[i];
22 }
```

普通循环

循环展开

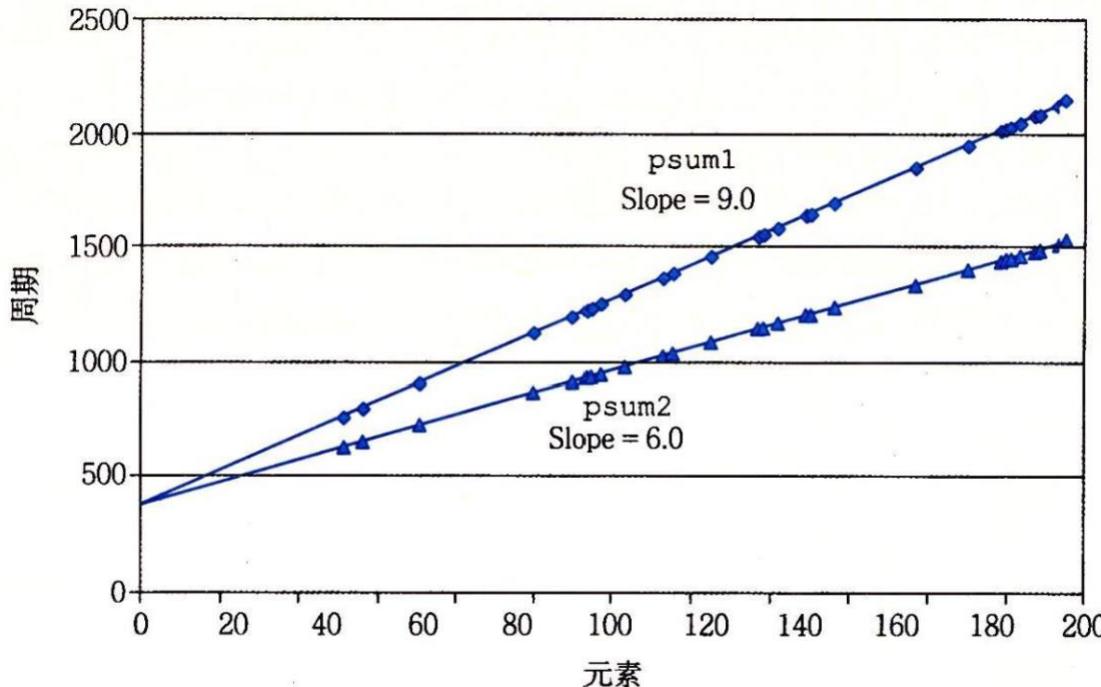


图 5-2 前置和函数的性能。两条线的斜率表明每元素的周期数(CPE)的值

psum1:368+9.0n

psum2:368+6.0n

n较大时运行时间主要由线性因子决定，故psum1的CPE = 9.0， psum2的CPE=6.0

抽象出一个向量数据结构：由头部和数据数组组成

头部：

```
1  /* Create abstract data type for vector */
2  typedef struct {
3      long len;
4      data_t *data;
5  } vec_rec, *vec_ptr;
```

```
#define IDENT 0
#define OP +
```

它对向量的元素求和。使用声明：

```
#define IDENT 1
#define OP *
```

它计算的是向量元素的乘积。

```
1  /* Implementation with maximum use of data abstraction */
2  void combine1(vec_ptr v, data_t *dest)
3  {
4      long i;
5
6      *dest = IDENT;
7      for (i = 0; i < vec_length(v); i++) {
8          data_t val;
9          get_vec_element(v, i, &val);
10         *dest = *dest OP val;
11     }
12 }
```

消除循环中的低效率:

- 代码移动

将vec_length函数移到循环外，
以减少无意义的重复计算

编译器不会轻易进行这样的优化，
因为不能确定“副作用”

```
2 void combine2(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6
7     *dest = IDENT;
8     for (i = 0; i < length; i++) {
9         data_t val;
10        get_vec_element(v, i, &val);
11        *dest = *dest OP val;
12    }
13 }
```

函数	方法	整数		浮点数	
		+	*	+	*
combine1	抽象的 -O1	10.12	10.12	10.17	11.14
combine2	移动 vec_length	7.02	9.03	9.02	11.03

减少过程调用：

combine2中循环中每次都要检查边界，会带来不必要的开销

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

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

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

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

我们发现并没有带来性能的提高
是因为这是一个高度可预测的分支

函数	方法	整数		浮点数	
		+	*	+	*
combine2	移动 vec_length	7.02	9.03	9.02	11.03
combine3	直接数据访问	7.17	9.02	9.02	11.03

消除不必要的内存引用：累积临时变量

```

Inner loop of combine3. data_t = double, OP = *
dest in %rbx, data+i in %rdx, data+length in %rax
.L17:                                loop:
    vmovsd  (%rbx), %xmm0          Read product from dest
    vmulsd  (%rdx), %xmm0, %xmm0   Multiply product by data[i]
    vmovsd  %xmm0, (%rbx)          Store product at dest
    addq    $8, %rdx               Increment data+i
    cmpq    %rax, %rdx             Compare to data+length
    jne     .L17                  If !=, goto loop

```

```

Inner loop of combine4. data_t = double, OP = *
acc in %xmm0, data+i in %rdx, data+length in %rax
.L25:                                loop:
    vmulsd  (%rdx), %xmm0, %xmm0   Multiply acc by data[i]
    addq    $8, %rdx               Increment data+i
    cmpq    %rax, %rdx             Compare to data+length
    jne     .L25                  If !=, goto loop

```

```

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

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

void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}

```

函数	方法	整数		浮点数	
		+	*	+	*
combine3	直接数据访问	7.17	9.02	9.02	11.03
combine4	累积在临时变量中	1.27	3.01	3.01	5.01

由于内存别名的使用，两个函数的行为可能不同，因此编译器不会自动做这样的优化。

```

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

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

combine3(v, get_vec_start(v) + 2);
combine4(v, get_vec_start(v) + 2);

void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

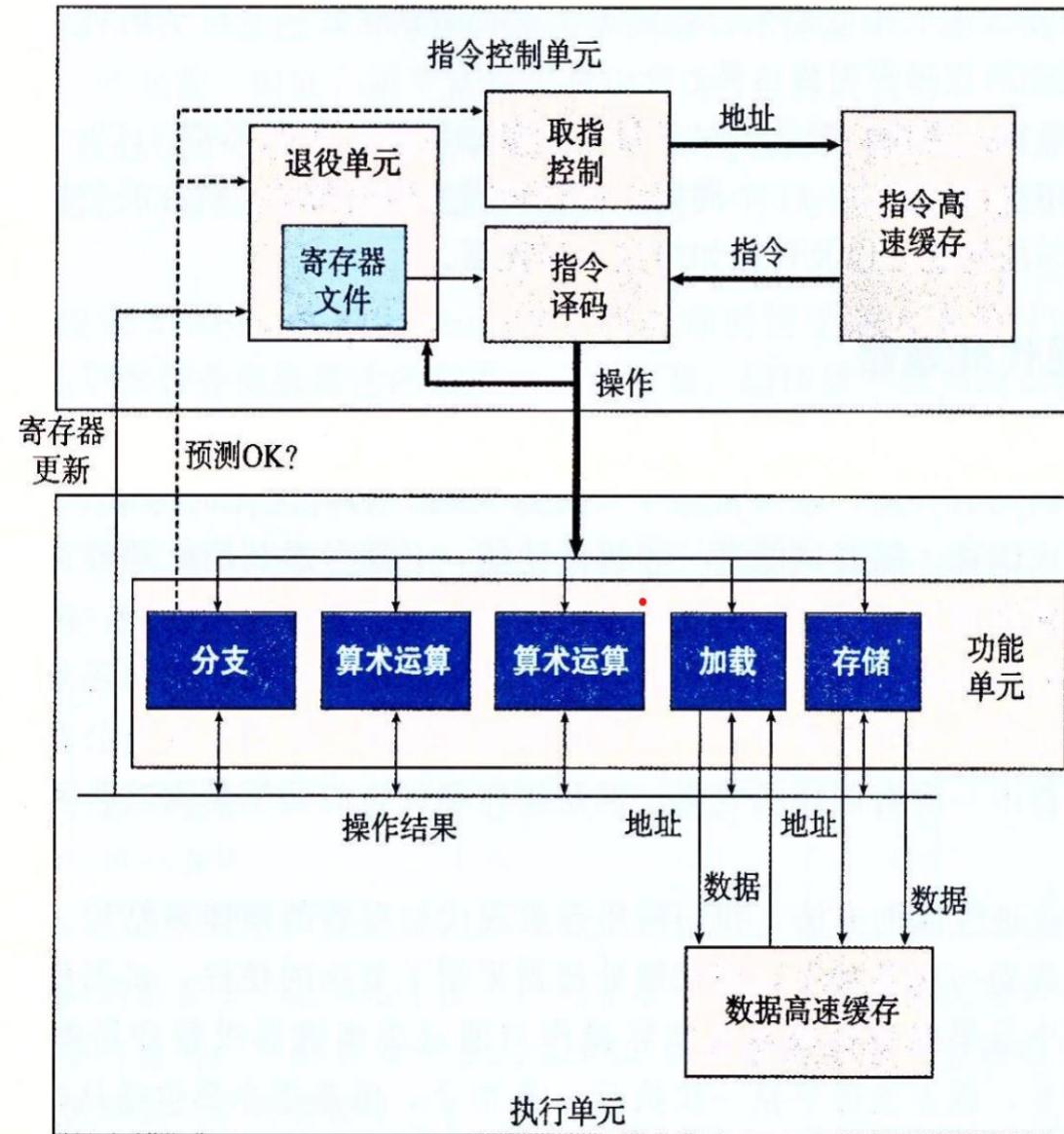
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}

```

也就是在向量最后一个元素和存放结果的目标之间创建一个别名。那么，这两个函数的执行如下：

了解现代处理器，什么在制约程序的性能：

- 超标量处理器：
- 指令级并行，乱序的，但同时达到顺序执行效果
- 两种下界描述程序的最大性能：
- 1、延迟界限
- 2、吞吐量界限（程序性能的终极限制）
- 两个主要部分：
- 指令控制单元ICU
- 执行单元EU



一个乱序处理器的框图。指令控制单元负责从内存中读出指令，并产生一系列基本操作。然后执行单元完成这些操作，以及指出分支预测是否正确

Intel Core i7 Haswell 有8个功能单元：

- 0**: 整数运算、浮点乘、整数和浮点数除法、分支
- 1**: 整数运算、浮点加、整数乘、浮点乘
- 2**: 加载、地址计算
- 3**: 加载、地址计算
- 4**: 存储
- 5**: 整数运算
- 6**: 整数运算、分支
- 7**: 存储、地址计算

功能单元的性能：

延迟时间：完成运算需要的总时间

发射时间：两个连续的同类型运算间需要的最小时钟周期数

运算	整数			浮点数		
	延迟	发射	容量	延迟	发射	容量
加法	1	1	4	3	1	1
乘法	3	1	1	5	1	2
除法	3 ~ 30	3 ~ 30	1	3 ~ 15	3 ~ 15	1

吞吐量=CPE的最小界限

界限	整数		浮点数	
	+	*	+	*
延迟	1.00	3.00	3.00	5.00
吞吐量	0.50	1.00	1.00	0.50

0：整数运算、浮点乘、整数和浮点数除法、分支

1：整数运算、浮点加、整数乘、浮点乘

2：加载、地址计算

3：加载、地址计算

4：存储

5：整数运算

6：整数运算、分支

7：存储、地址计算

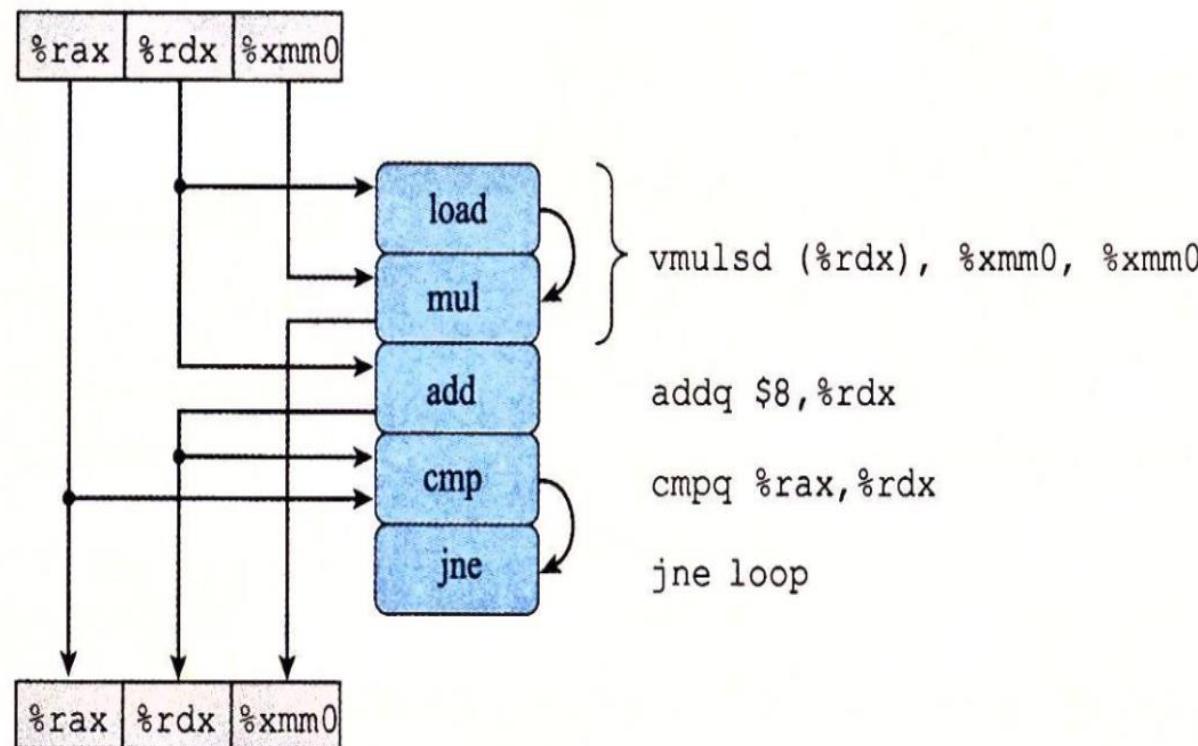
N个某种运算，C个相应功能单元，这些单元的发射时间I，完成这个程序至少要N*I/C个周期

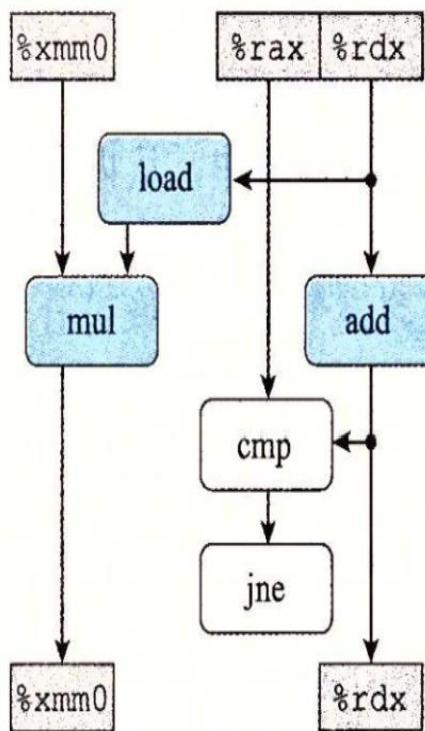
处理器操作的抽象模型

Inner loop of combine4. data_t = double, OP = *
acc in %xmm0, data+i in %rdx, data+length in %rax
.L25: loop:
 vmulsd (%rdx), %xmm0, %xmm0 Multiply acc by data[i]
 addq \$8, %rdx Increment data+i
 cmpq %rax, %rdx Compare to data+length
 jne .L25 If !=, goto loop

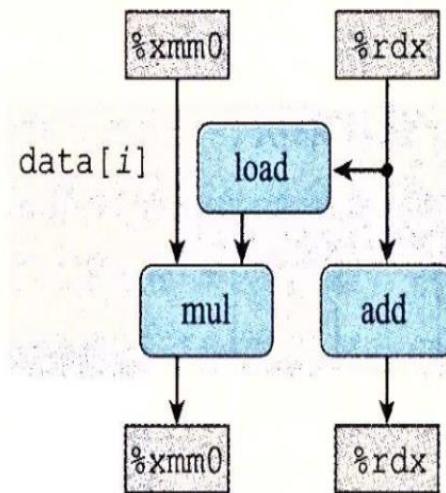
```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

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

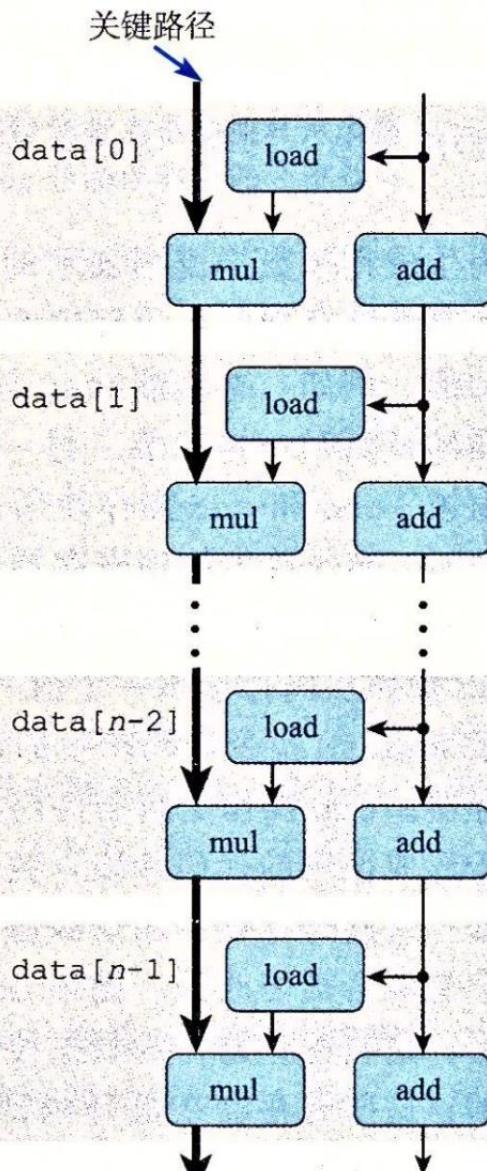




a) 重新排列了图5-13的操作符，更清晰地表明了数据相关



b) 操作在一次迭代中使用某些值，产生出在下一次迭代中需要的新值



关键路径只是需要时钟周期数的下界

函数	方法	整数		浮点数	
		+	*	+	*
combine3	直接数据访问	7.17	9.02	9.02	11.03
combine4	累积在临时变量中	1.27	3.01	3.01	5.01

循环展开：（不能超过延迟界限）

k*1展开：

k是指第一个循环每次处理k个元素，
循环索引加k

第二个循环处理不足k个的元素

1是指一个累积变量

对于整数加法，每次循环连续计算两个元素，循环p次，

$$CPE = (1+1)*p/2*p = 1.00$$

```
1 /* 2 x 1 loop unrolling */
2 void combine5(vec_ptr v, data_t *dest)
3 {
4     long i;
5     long length = vec_length(v);
6     long limit = length-1;
7     data_t *data = get_vec_start(v);
8     data_t acc = IDENT;
9
10    /* Combine 2 elements at a time */
11    for (i = 0; i < limit; i+=2) {
12        acc = (acc OP data[i]) OP data[i+1];
13    }
14
15    /* Finish any remaining elements */
16    for (; i < length; i++) {
17        acc = acc OP data[i];
18    }
19    *dest = acc;
20 }
```

图 5-16 使用 2×1 循环展开。这种变换能减小循环开销的影响

函数	方法	整数		浮点数	
		+	*	+	*
combine4	无展开	1.27	3.01	3.01	5.01
combine5	2×1 展开	1.01	3.01	3.01	5.01
	3×1 展开	1.01	3.01	3.01	5.01
延迟界限		1.00	3.00	3.00	5.00
吞吐量界限		0.50	1.00	1.00	0.50

为什么循环展开不能超过延迟界限： 对于 2×1 展开的combine5

```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    acc = (acc OP data[i]) OP data[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    acc = acc OP data[i];
}
*dest = acc;
```

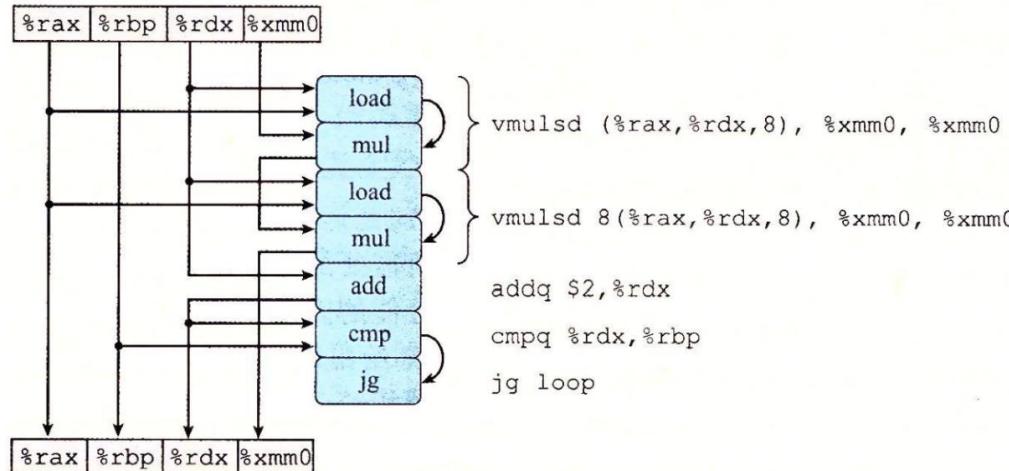
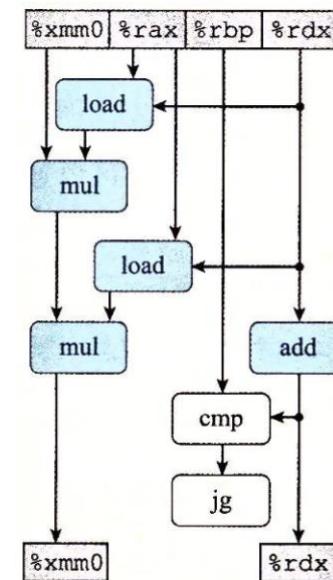
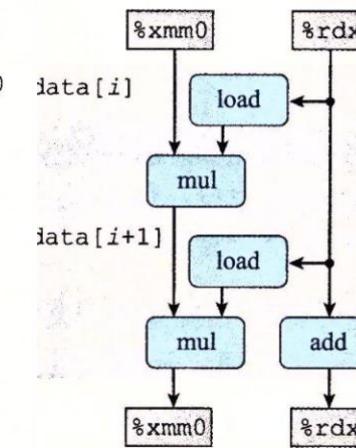


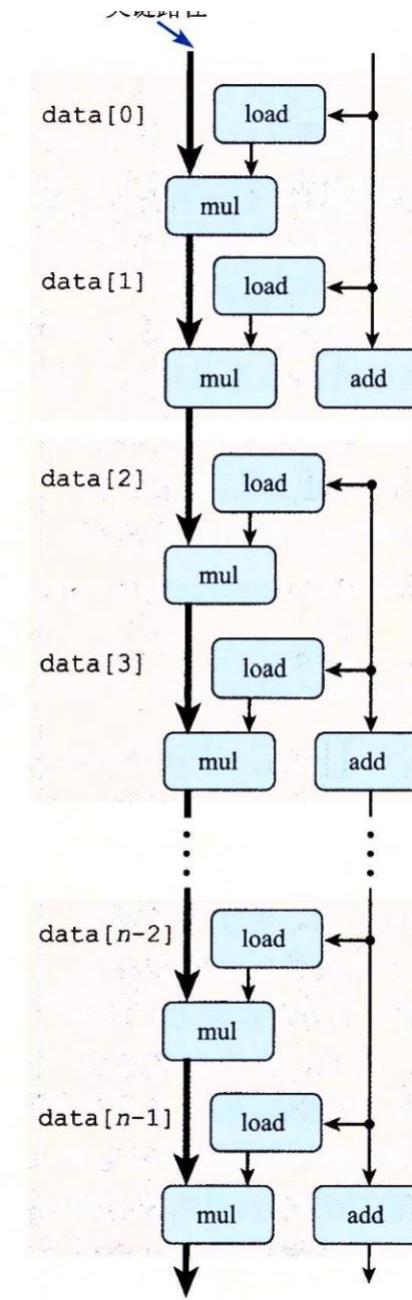
图 5-18 combine5 内循环代码的图形化表示。每次迭代有两条 vmulsd 指令，每条指令被翻译成一个 load 和一个 mul 操作



重新排列、简化和抽象图 5-18 的
示例，给出连续迭代之间的数据相



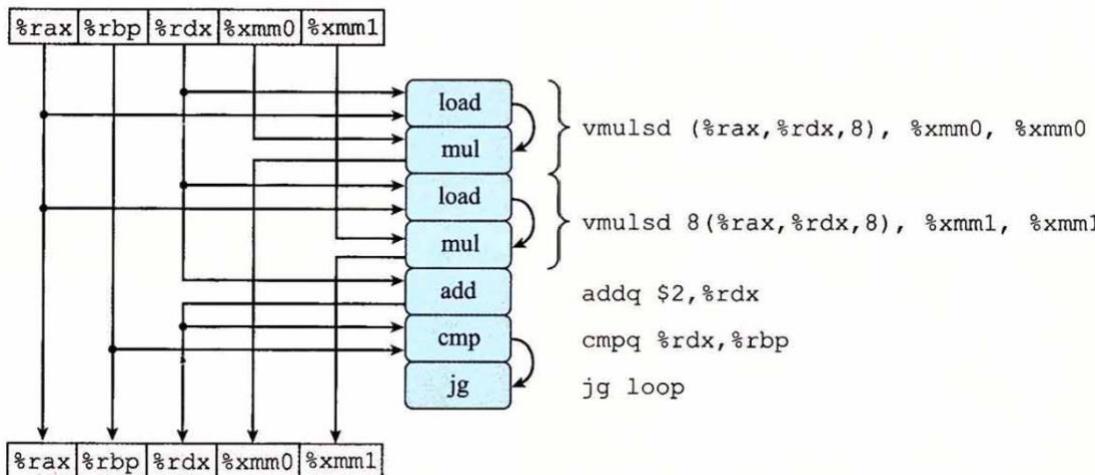
每次迭代必须顺序地执行两个乘



提高并行性

- 多个累积变量 (k*k展开)
 - 前提：可结合和可交换的运算

对于循环因子k, 延迟L, 容量为C, 要达到
吞吐量界限, 要求 $k \geq C * L$



```

void combine6(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;

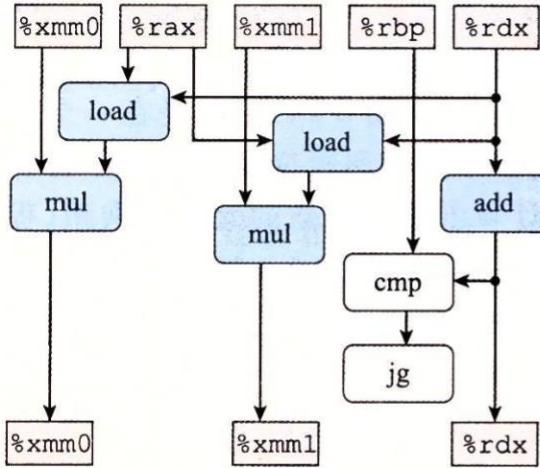
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc0 = acc0 OP data[i];
        acc1 = acc1 OP data[i+1];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }

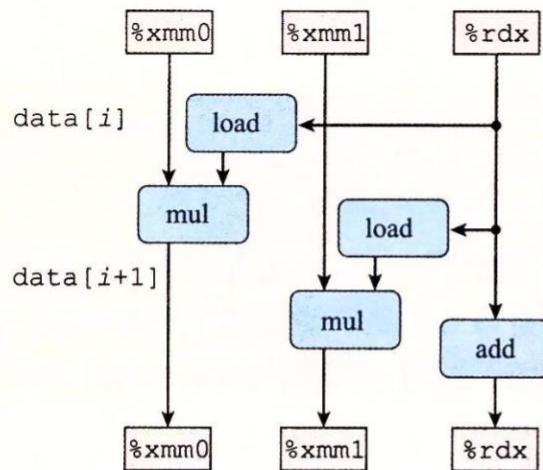
    *dest = acc0 OP acc1;
}

```

函数	方法	整数		浮点数	
		+	*	+	*
combine4	在临时变量中累积	1.27	3.01	3.01	5.01
combine5	2×1 展开	1.01	3.01	3.01	5.01
combine6	2×2 展开	0.81	1.51	1.51	2.51
延迟界限		1.00	3.00	3.00	5.00
吞吐量界限		0.50	1.00	1.00	0.50



a) 重新排列、简化和抽象图5-22的表示，
给出连续迭代之间的数据相关



b) 两个mul操作之间没有相关

图 5-23 将 combine6 的运算
抽象成数据流图

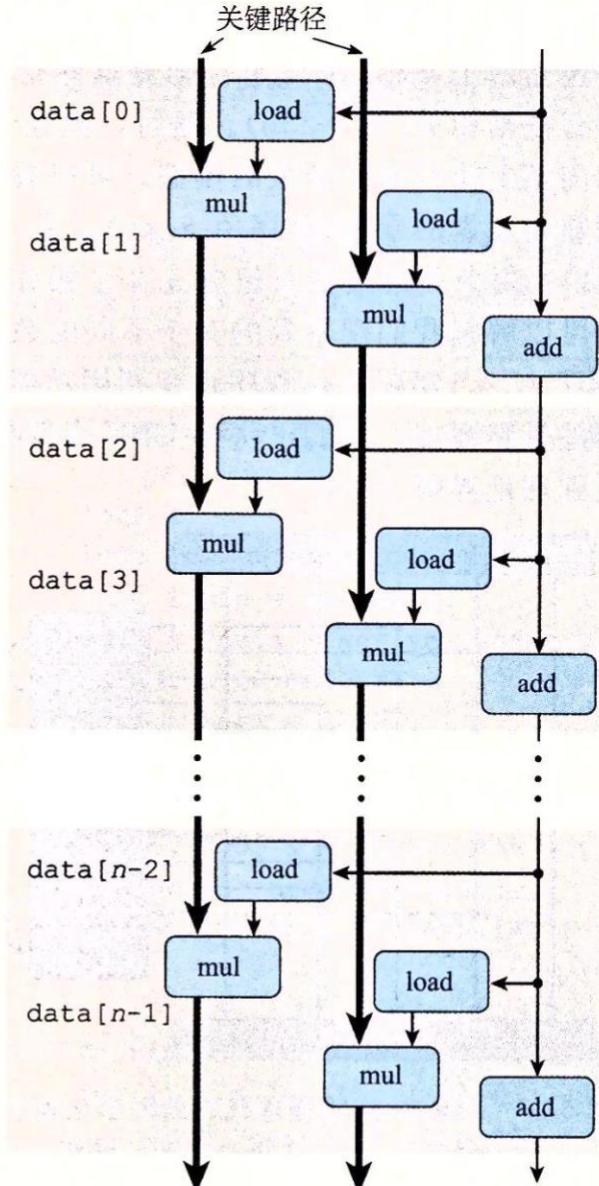
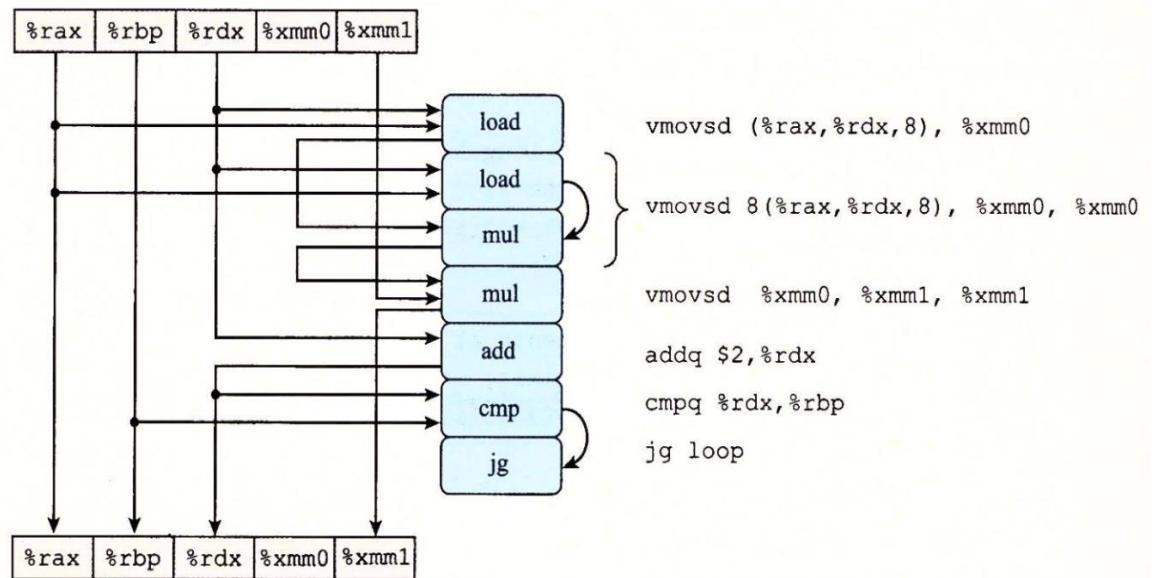


图 5-24 combine6 对一个长度为 n 的向量进行操作的数据流表示。现在有两条关键路径，每条关键路径包含 $n/2$ 个操作

重新结合变换（另一种可以达到吞吐量界限的方法） $2 \times 1a$ 展开

前提：可结合的操作

大多数编译器不会对浮点运算做重新结合



```

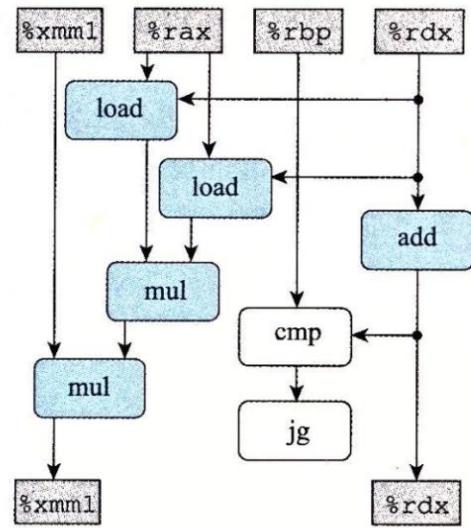
void combine7(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = acc OP (data[i] OP data[i+1]);
    }

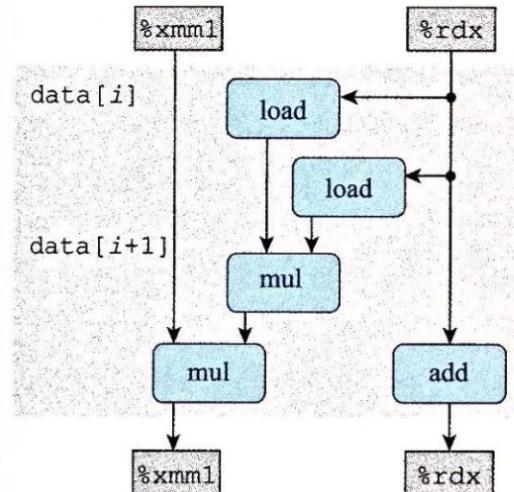
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}

```

函数	方法	整数		浮点数	
		+	*	+	*
combine4	累积在临时变量中	1.27	3.01	3.01	5.01
combine5	2×1 展开	1.01	3.01	3.01	5.01
combine6	2×2 展开	0.81	1.51	1.51	2.51
combine7	$2 \times 1a$ 展开	1.01	1.51	1.51	2.51
延迟界限		1.00	3.00	3.00	5.00
吞吐量界限		0.50	1.00	1.00	0.50



a) 重新排列、简化和抽象图5-27的表示，
给出连续迭代之间的数据相关



b) 上面的mul操作让两个二向量元素相乘，而
下面的mul操作将前面的结果乘以循环变量acc

图 5-28 将 combine7 的操作
抽象成数据流图

关键路径

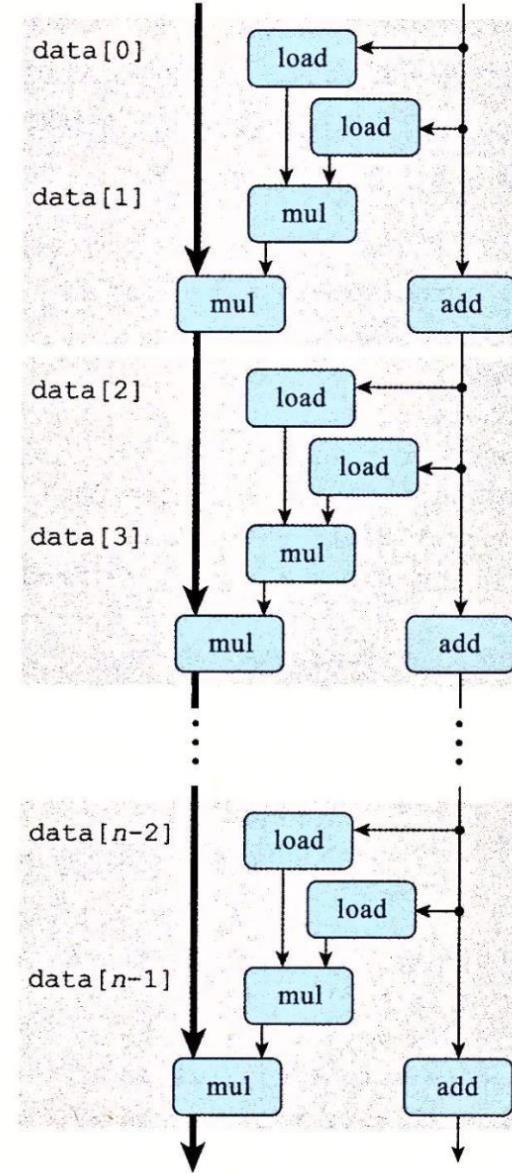


图 5-29 combine7 对一个长度为 n 的向量进行操作的数据流表示。我们只有一条关键路径，它只包含
10 个操作

书上练习题5.8：

```
double aprod(double a[], long n)
{
    long i;
    double x, y, z;
    double r = 1;
    for (i = 0; i < n-2; i+= 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * x * y * z; /* Product computation */
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

对于标记为 Product computation 的行，可以用括号得到该计算的五种不同的结合，如下所示：

```
r = ((r * x) * y) * z; /* A1 */
r = (r * (x * y)) * z; /* A2 */
r = r * ((x * y) * z); /* A3 */
r = r * (x * (y * z)); /* A4 */
r = (r * x) * (y * z); /* A5 */
```

假设在一台浮点数乘法延迟为 5 个时钟周期的机器上运行这些函数。确定由乘法的数据相关限定的 CPE 的下界。(提示：画出每次迭代如何计算 r 的图形化表示会所帮助。)

寄存器溢出：

- 如果并行度超过了寄存器数量，就会导致寄存器溢出，编译器会将部分临时值放到内存，比如栈，这会带来额外开销

函数	方法	整数		浮点数	
		+	*	+	*
combine6	10×10 循环展开	0.55	1.00	1.01	0.52
	20×20 循环展开	0.83	1.03	1.02	0.68
	吞吐量界限	0.50	1.00	1.00	0.50

内存性能：

加载（内存读）：

- 对于只有2个加载单元的处理器而言，每个周期最多同时执行两条加载指令，所以对于需要加载k个值的，CPE至少为k/2

```
1  typedef struct ELE {  
2      struct ELE *next;  
3      long data;  
4  } list_ele, *list_ptr;                                Inner loop of list_len  
5  
6  long list_len(list_ptr ls) {                         ls in %rdi, len in %rax  
7      long len = 0;                                     .L3:                                loop:  
8      while (ls) {  
9          len++;  
10         ls = ls->next;                            2    addq    $1, %rax           Increment len  
11     }  
12     return len;                                    3    movq    (%rdi), %rdi      ls = ls->next  
13 }
```

• 4 testq %rdi, %rdi Test ls
 5 jne .L3 If nonnull, goto loop

图 5-31 链表函数。其性能受限于
加载操作的延迟

存储的性能：（写入内存）

存储操作不会修改任何寄存器的值，因此不会与大部分操作形成数据相关，只有加载操作会受到存储操作的影响
当一个内存读依赖于最近的一个内存写时，这种现象为“写/读相关”

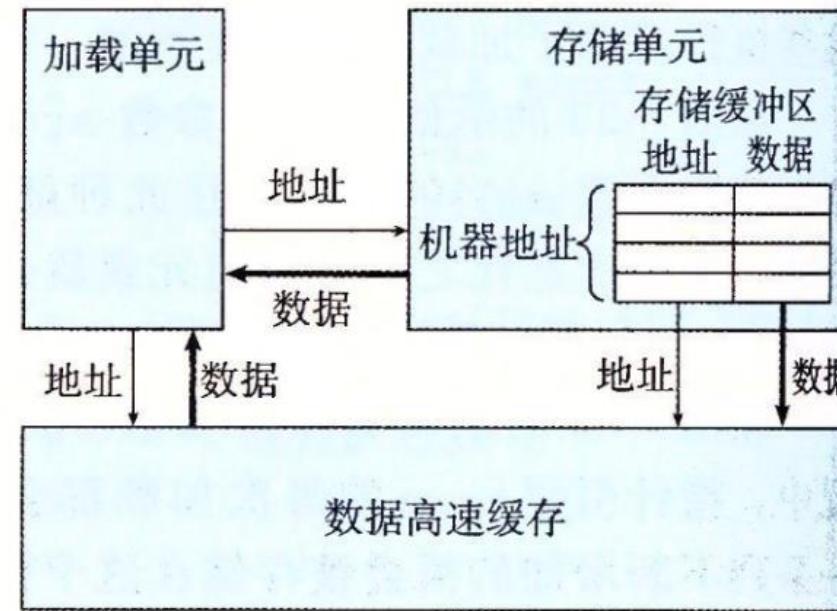


图 5-34 加载和存储单元的细节。存储单元包含一个未执行的写的缓冲区。加载单元必须检查它的地址是否与存储单元中的地址相符，以发现写/读相关

```

/* Write to dest, read from src */
void write_read(long *src, long *dst, long n)
{
    long cnt = n;
    long val = 0;

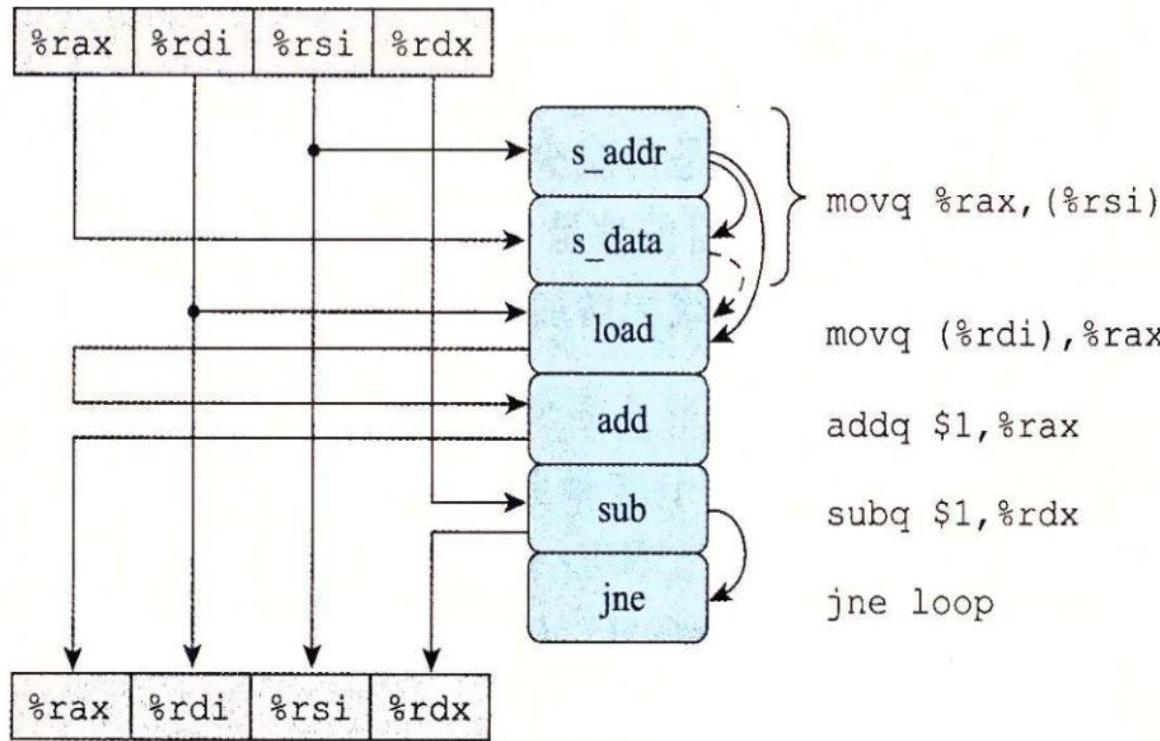
    while (cnt) {
        *dst = val;
        val = (*src)+1;
        cnt--;
    }
}

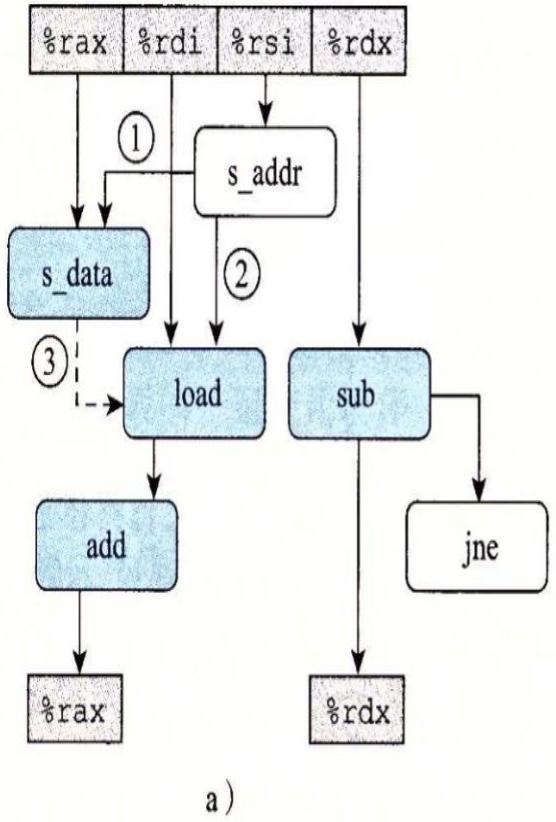
```

Inner loop of write_read
src in %rdi, dst in %rsi, val in %rax

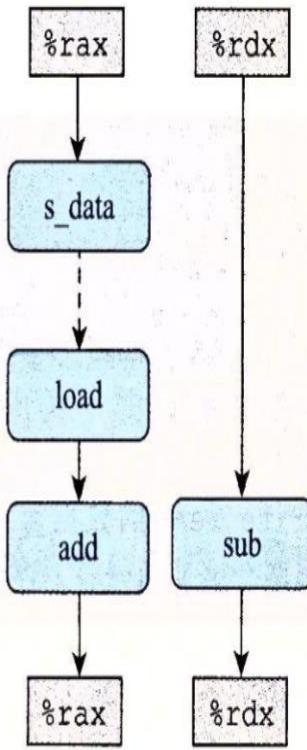
.L3: *loop:*

movq %rax, (%rsi)	<i>Write val to dst</i>
movq (%rdi), %rax	<i>t = *src</i>
addq \$1, %rax	<i>val = t+1</i>
subq \$1, %rdx	<i>cnt--</i>
jne .L3	<i>If != 0, goto loop</i>



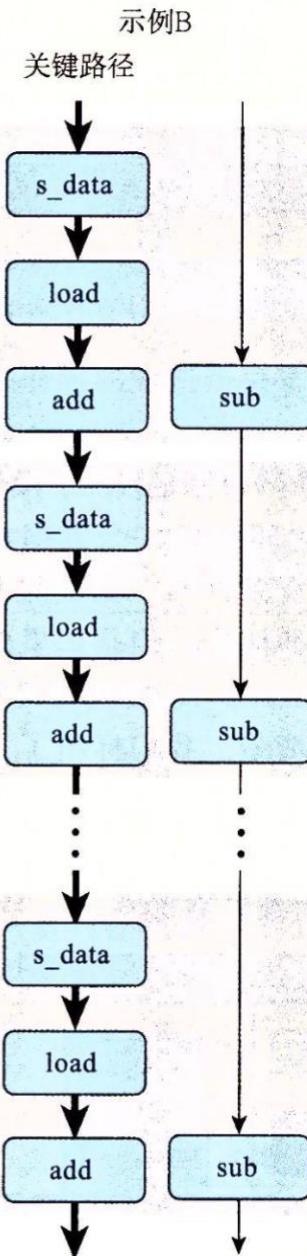
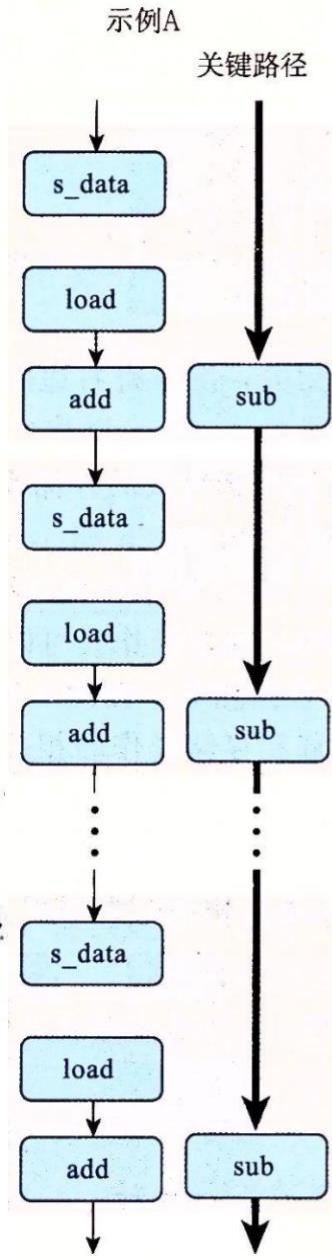


a)



b)

-36 抽象 write_read 的操作。我们首先重新排列图 5-35 的操作(a)，然后只显示那些使用一次迭代中的值为下一次迭代产生新值的操作(b)



练习题 5.11 我们测量出前置和函数 psum1(图 5-1)的 CPE 为 9.00，在测试机器上，要执行的基本操作——浮点加法的延迟只是 3 个时钟周期。试着理解为什么我们的函数执行效果这么差。

下面是这个函数内循环的汇编代码：

```
Inner loop of psum1
a in %rdi, i in %rax, cnt in %rdx
1 .L5:                                loop:
2     vmovss -4(%rsi,%rax,4), %xmm0      Get p[i-1]
3     vaddss (%rdi,%rax,4), %xmm0, %xmm0   Add a[i]
4     vmovss %xmm0, (%rsi,%rax,4)          Store at p[i]
5     addq    $1, %rax                      Increment i
6     cmpq    %rdx, %rax                   Compare i:cnt
7     jne     .L5                           If !=, goto loop
```

可以画出数据流图，分析产生的数据相关

The End