第4章　存储层次结构设计

- 4.1 存储层次结构
- 4.2 Cache基本知识
- 4.3 基本的Cache优化方法
- 4.4 高级的Cache优化方法
- 4.5 存储器技术与优化
- 4.6 虚拟存储器—基本原理

4.4 高级Cache优化方法

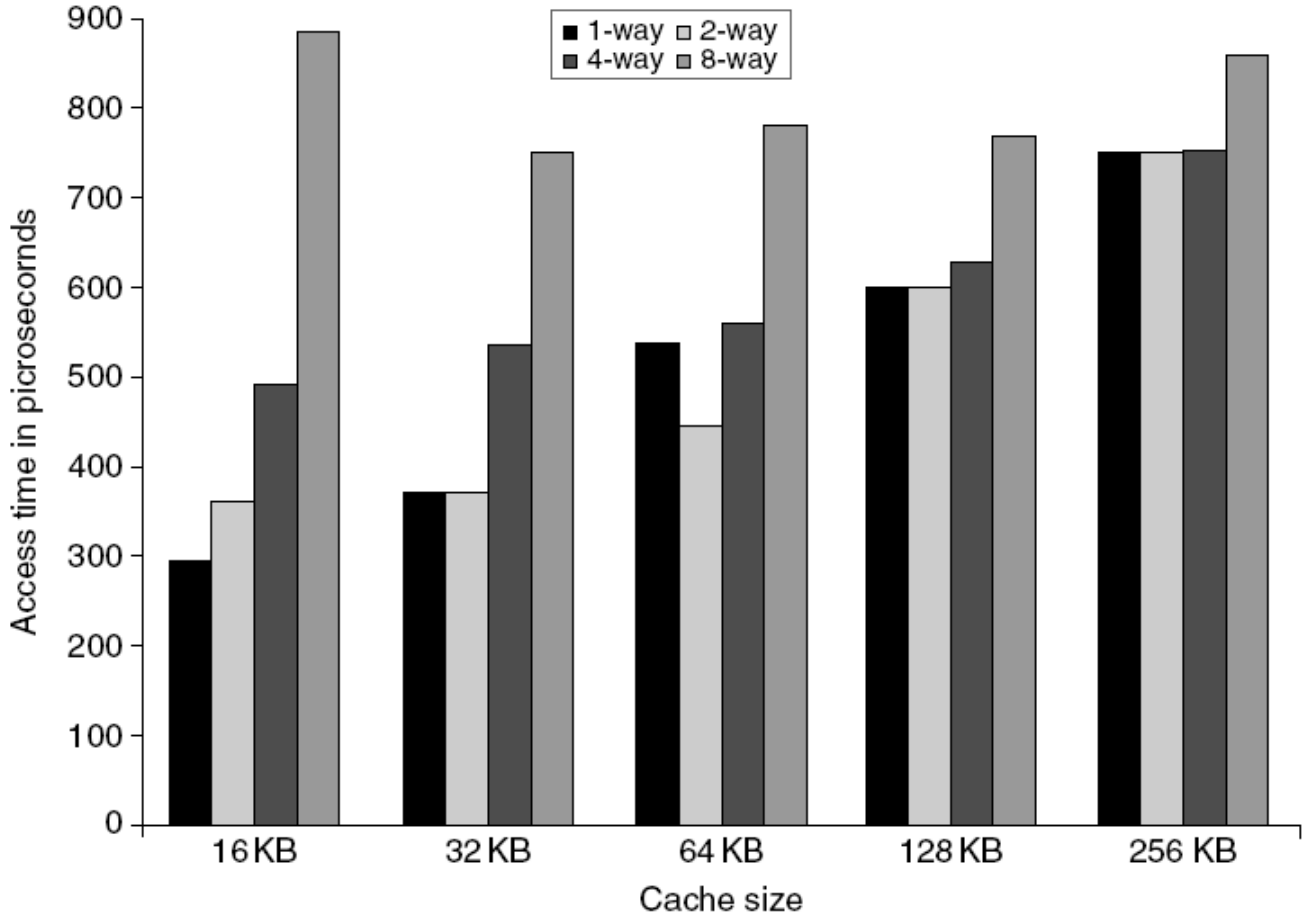
- 缩短命中时间
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- 增加Cache带宽
  - 3、Cache访问流水化
  - 4、无阻塞Cache
- 减小失效开销
  - 5、多体Cache
  - 6、关键字优先和提前重启
  - 7、合并写
- 降低失效率
  - 8、编译优化
- 通过并行降低失效代价或失效率
  - 9、硬件预取
  - 10、编译器控制的预取

1、Small and simple first level caches

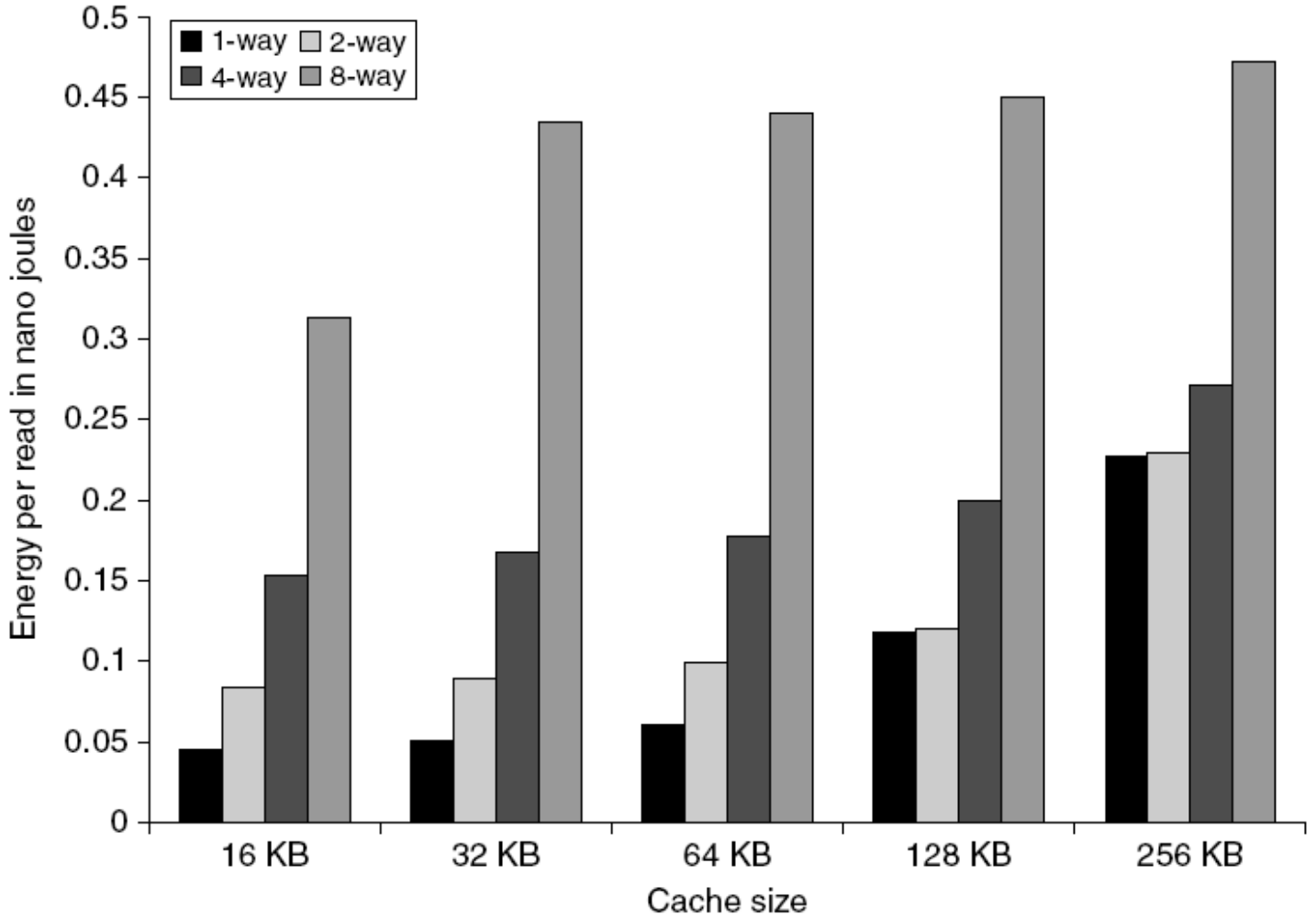
- Small and simple first level caches
  - 容量小，一般命中时间短，有可能做在片内
  - 另一方案，保持Tag在片内，块数据在片外，如DEC Alpha
  - 第一级Cache应选择容量小且结构简单的设计方案
- Critical timing path:

  1) 定位组(tag), 确定tag的位置
  2) 比较tags,
  3) 选择正确的块
- Direct-mapped caches can overlap tag compare and transmission of data
  - 数据传输和tag 比较并行
- Lower associativity reduces power because fewer cache lines are accessed
  - 简单的Cache结构、可有效减少tag比较的次数，进而降低功耗

L1 Size and Associativity



L1 Size and Associativity



2、Way Prediction

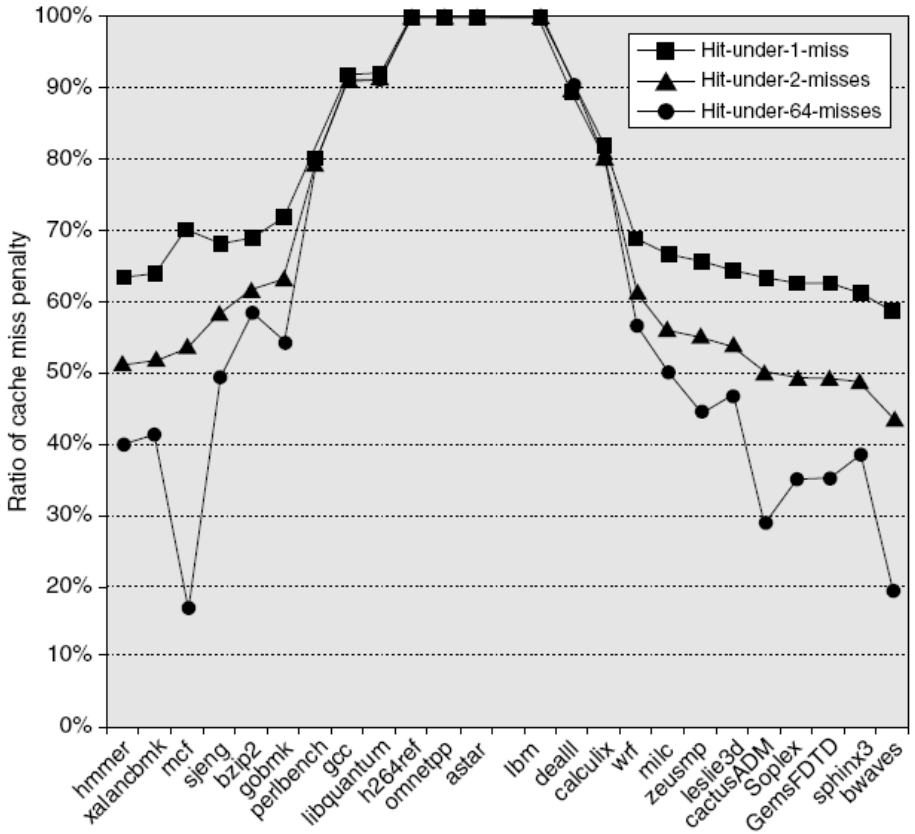
- To improve hit time, predict the way to pre-set mux
  - Mis-prediction gives longer hit time
  - Prediction accuracy
    - > 90% for two-way
    - > 80% for four-way
    - I-cache has better accuracy than D-cache
  - First used on MIPS R10000 in mid-90s
  - Used on ARM Cortex-A8
- Extend to predict block as well
  - “Way selection”
  - Increases mis-prediction penalty

3、Pipelining Cache

- 实现Cache访问的流水化，提高Cache的带宽
  - Examples:
    - Pentium: 1 cycle
    - Pentium Pro – Pentium III: 2 cycles
    - Pentium 4 – Core i7: 4 cycles
- 增加了流水线的段数，优缺点：
  - 增加了分支预测错误造成的额外开销
  - 增加了Load指令与要使用其结果的指令间的latency
  - 有利于采用高相联度的缓存

4、Nonblocking Caches

- 对有些允许乱序执行的机器（采用动态调度方法）,CPU无需在Cache失效时等待。即在等待数据Cache失效时，可以继续取指令。
- 采用非阻塞Cache，在某一访问Cache失效时，仍然允许CPU进行其他的命中访问，可以有效地提高CPU性能。
  - “Hit under miss”
  - “Hit under multiple miss”
- 通常非阻塞Cache对于允许乱序执行的处理器可以隐藏大多数L1的失效开销，但很难隐藏L2 的失效开销



5、Multibanked Caches

- 将Cache组织为多个独立的banks，以支持并行访问
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2
- 根据块号进行多体交叉编址

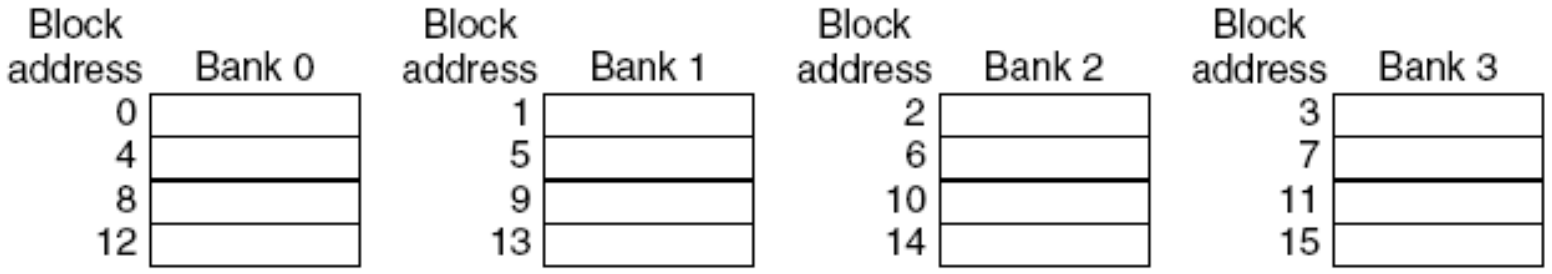


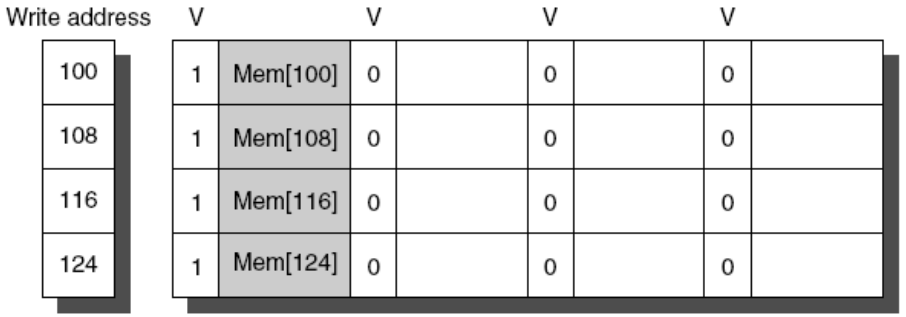
Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

6、Critical Word First, Early Restart

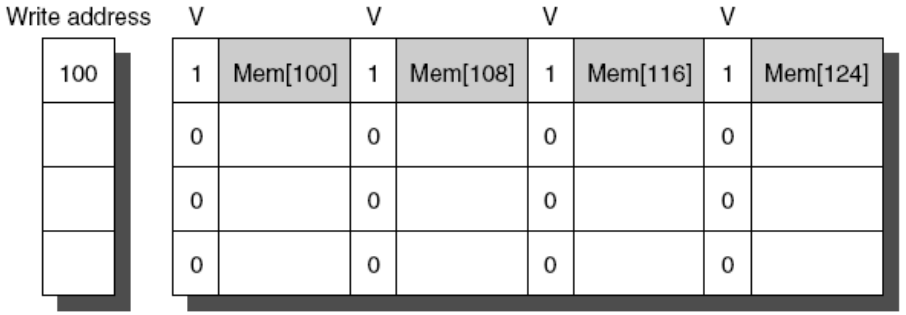
- 关键字优先
  - 首先请求CPU所需要的字
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- 提前重启
  - 请求字的顺序不变
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- 通常在块比较大时，这些技术才有效

7、Merging Write Buffer

- 在向写缓冲器写入地址和数据时，如果写缓冲器中存在被修改过的块，就检查其地址，看看本次写入数据的地址是否与写缓冲器内的某个有效块地址匹配，如果匹配，就把新数据与该块合并，称为“合并写”
- 可以缓解由于写缓冲满而造成的CPU停顿
- 不适用于I/O地址空间？？



No write buffering



Write buffering

8、编译器优化

- 无需对硬件做任何改动，通过软件优化降低失效率
- 研究从两方面展开：
  - 减少指令失效
  - 减少数据失效
- 减少指令失效，重新组织程序（指令调度）而不影响程序的正确性
  - Mc-Farling 的研究结果：通过使用profiling信息来判断指令组间可能发生的冲突，并将指令重新排序以减少失效。
  - 研究表明：容量为2KB, 块大小为 4Bytes的直接映象Icache，通过使用指令调度可以使失效率降低50％。容量增大到 8KB, 失效率可降低75％
  - 在有些情况下，当能够使某些指令不进入ICache时，可以得到最佳性能。即使不这样做，优化后的程序在直接映象Cache中的失效率也低于未优化程序在同样大小的8路组相联Cache中的失效率。
- 减少数据失效，主要通过优化来改善数据的空间局部性和时间局部性，基本方法为：
  - 数据合并
  - 内外循环交换，循环融合
  - 分块

编译器优化方法举例之一：数组合并

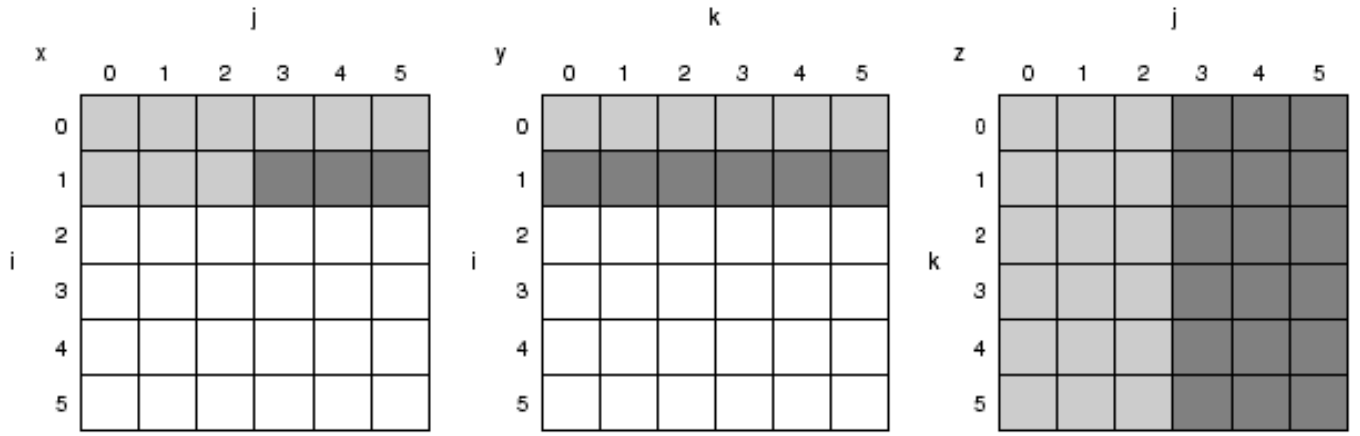
```
/* 修改前*/
int val[SIZE];
int key[SIZE];
/* 修改后*/
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];
```

编译器优化方法举例之二：内外循环交换

```
/* Before */
for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
        x[i][j] = 2 * x[i][j];
/* After */
for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
```

编译器优化方法举例之三：分块 (1/2)

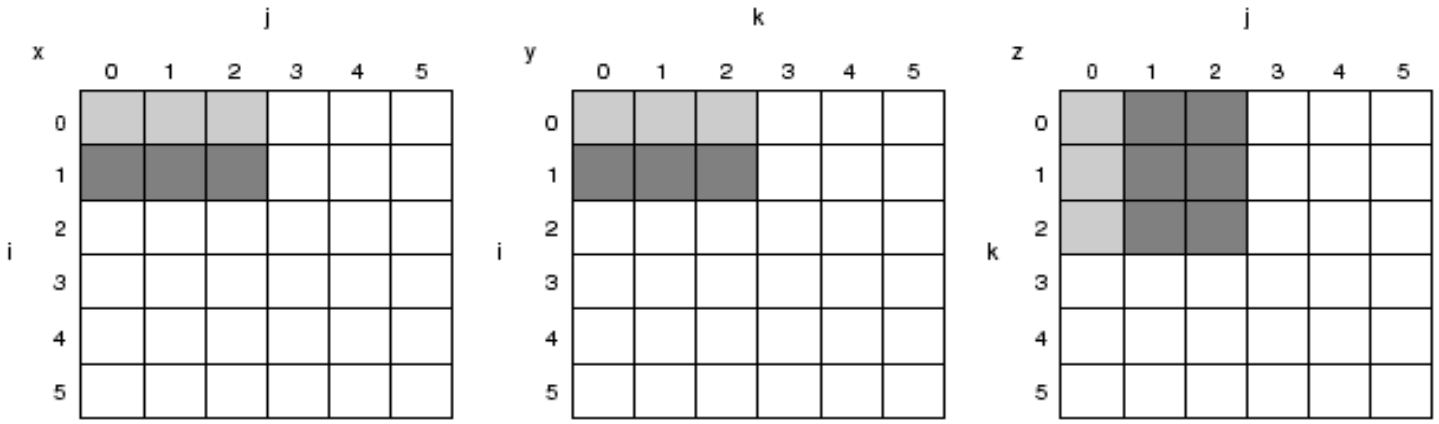
```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {   r = 0;
        for (k = 0; k < N; k = k + 1)
            r = r + y[i][k]*z[k][j];
        x[i][j] = r;
    };
```



最坏情况下，N^3次操作，产生的存储器访问次数为: $2N^3+N^2$

编译器优化方法举例之三：分块 (2/2)

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
for (j = jj; j < min(jj+B,N); j = j+1)
    {r = 0;
     for (k = kk; k < min(kk+B,N); k = k + 1)
         r = r + y[i][k]*z[k][j];
      x[i][j] = x[i][j] + r;
    };
```



9、Hardware Prefetching

- CPU在执行这块代码时，硬件预取下一块代码，因为CPU可能马上就要执行这块代码，这样可以降低或消除Cache的访问失效
- 当块中有控制指令时，预取失效
- 预取的指令可以放在Icache中，也可以放在其他地方（存取速度比Memory块的地方
- AXP21064失效时，取2块指令块
  - 目标块放在Icache，下一块放在ISB(指令流缓冲）中
  - 如果访问的块在ISB中，取消访存请求，直接从ISB中读，并发出对下一指令块的预取访存请求
- Jouppi研究结果：块大小为16字节，容量为4KB的直接映象Cache，1个块的指令流缓冲器，可以捕获15％－25％的失效，4个块 ISB可捕获50％的失效，16块ISB可捕获72％的失效

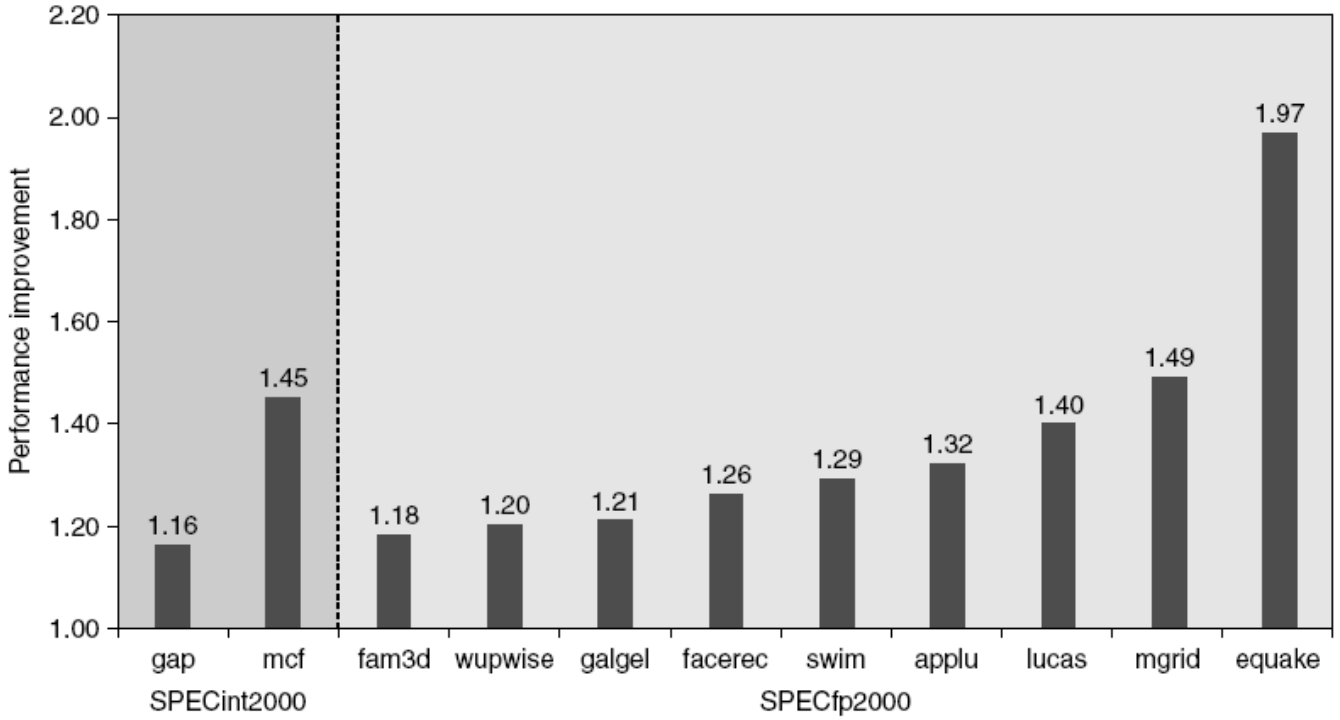
硬件预取

- 预取数据
  - 出发点：CPU访问一块数据，可能马上要访问下一块数据
  - Jouppi研究结果：块大小16字节，4KB直接映象Cache， 1Block DSB－25％
    4Block DSB － 43％
  - Palacharla和Kessler 1994年研究
    - 一个具有两个64KB四路组相联(Icache, Dcache)的处理器来说，8Blocks流缓冲器能够捕获其50％－70％的失效
- 举例：Alpha AXP21064采用指令预取技术，其实际失效率是多少？若不采用指令预取技术，Alpha AXP 21064的指令Cache必须为多大才能保持平均访存时间不变？
  - 假设当指令不在指令Cache中，在预取缓冲区中找到时，需要多花1个时钟周期。
  - 假设预取命中率为25%，命中时间为1个时钟周期，失效开销为50个时钟周期
  - 8KB指令Cache的失效率为1.10%，16KB指令cache的失效率为0.64%

  AMAT（预取）= 命中时间+失效率*预取命中率*1+失效率*（1-预取命中率）*失效开销
- 注意：预取是利用存储器的空闲带宽，而不是与正常的存储器操作竞争。

硬件预取

- Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

10、Compiler Prefetching (1/2)

- 在ISA中增加预取指令，让编译器控制预取
- 预取的种类
  - 寄存器预取：把数据取到R中
  - Cache预取：只将数据取到Cache中，不放入寄存器
- 故障问题
  - 两种类型的预取可以是故障性预取，也可以是非故障性预取
  - 所谓故障性预取指在预取时若出现虚地址故障，或违反保护权限，就会有异常发生
  - 非故障性预取，如导致异常就转化为空操作
- 只有在预取时，CPU可以继续执行的情况下，预取才有意义
  - Cache在等待预取数据返回的同时，可以正常提供指令和数据，称为非阻塞Cache或非锁定Cache

由编译器控制预取 （2/2）

- 循环是预取优化的主要目标
  - 失效开销较小时，Compiler简单的展开一两次，调度好预取与执行的重叠
  - 失效开销较大时，编译器将循环体展开多次，以便为较远的循环预取数据
  - 由于发出预取指令需要花费一条指令的开销，因此要避免不必要的预取
  - 重点放在那些可能导致失效的访问
- 举例1：P93（70）（Hennessy & Patterson 5th /中译本）
- 举例2：P94（71）（Hennessy & Patterson 5th/中译本）

举例1

Example： For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate thenumber of prefetch instructions executed and the misses avoided by prefetching.

Let's assume we have an 8 KB direct-mapped data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of a and b are 8 bytes long since they are double-precision floating-point arrays. There are 3 rows and 100 columns for a and 101 rows and 3 columns for b. Let's also assume they are not in the cache at the start of the program.

```
for (i = 0; i < 3; i = i+1)
    for (j = 0; j < 100; j = j+1)
        a[i][j] = b[j][0] * b[j+1][0];
```

举例2

Example：

Calculate the time saved in the example above.

(1) Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache.

(2) Assume that prefetches can overlap with each other and with cache misses, thereby transferring at the maximum memory bandwidth.

(3) Here are the key loop times ignoring cache misses: The original loop takes 7 clock cycles per iteration, the first prefetch loop takes 9 clock cycles per iteration, and the second prefetch loop takes 8 clock cycles per iteration (including the overhead of the outer for loop). A miss takes 100 clock cycles.

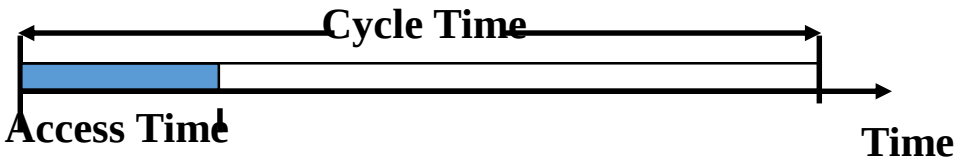
Summary

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

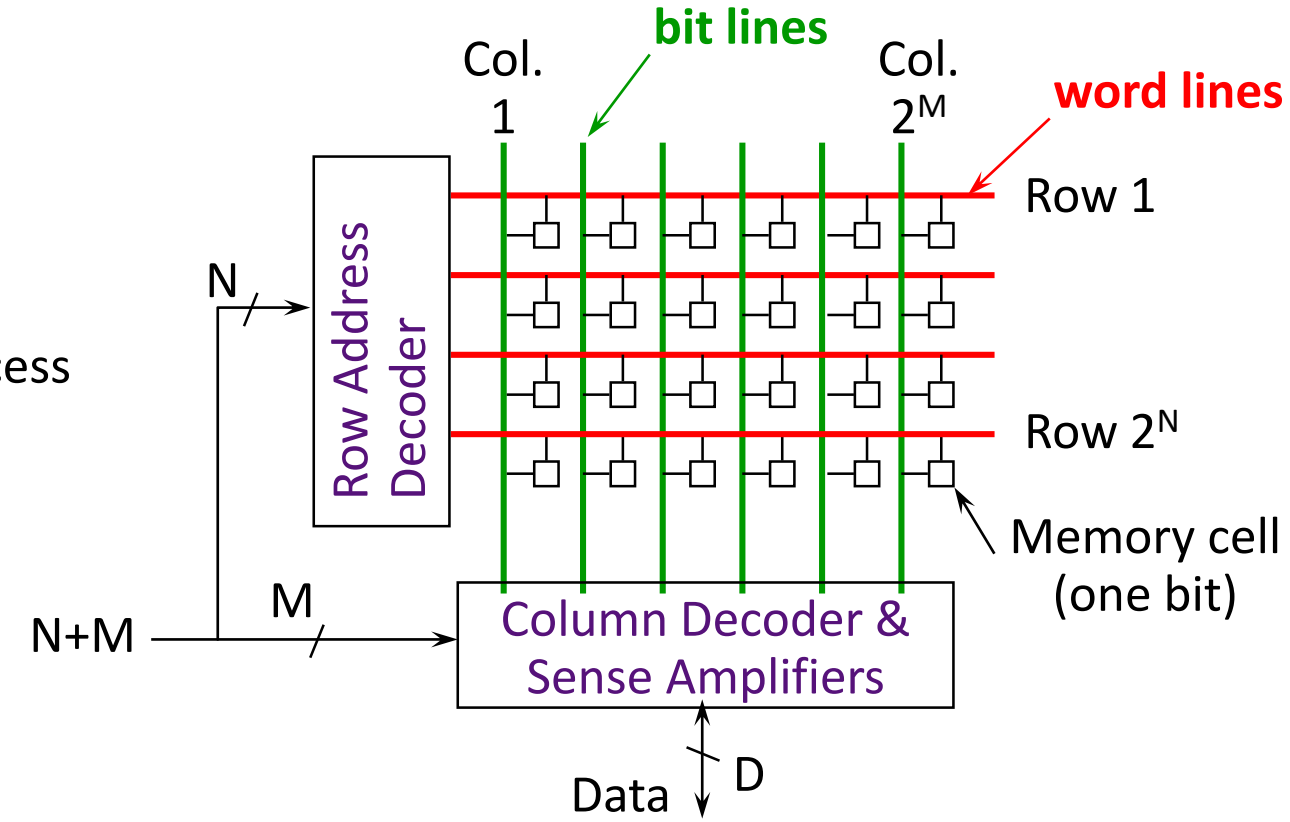
4.5 存储器技术与优化

- 存储器的访问源
  - 取指令、取操作数、写操作数和I/O
- 存储器性能指标
  - 容量、速度和每位价格
  - 访问时间（Access Time)
  - 存储周期（Cycle Time)
- 种类：DRAM和SRAM
  - Memory: DRAM, Cache: SRAM

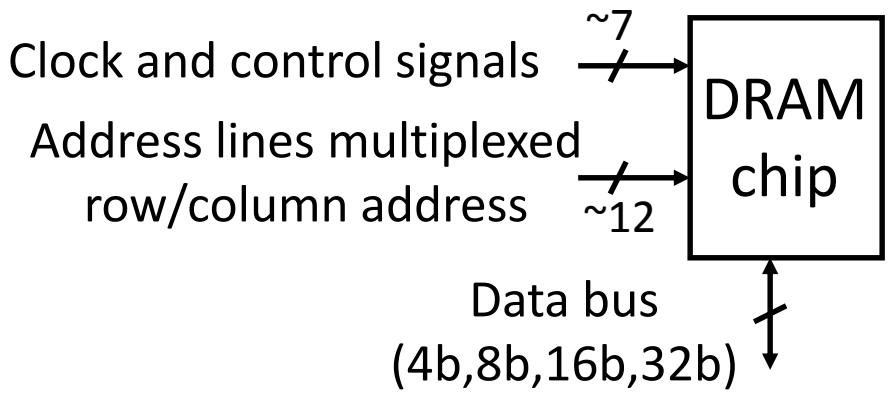


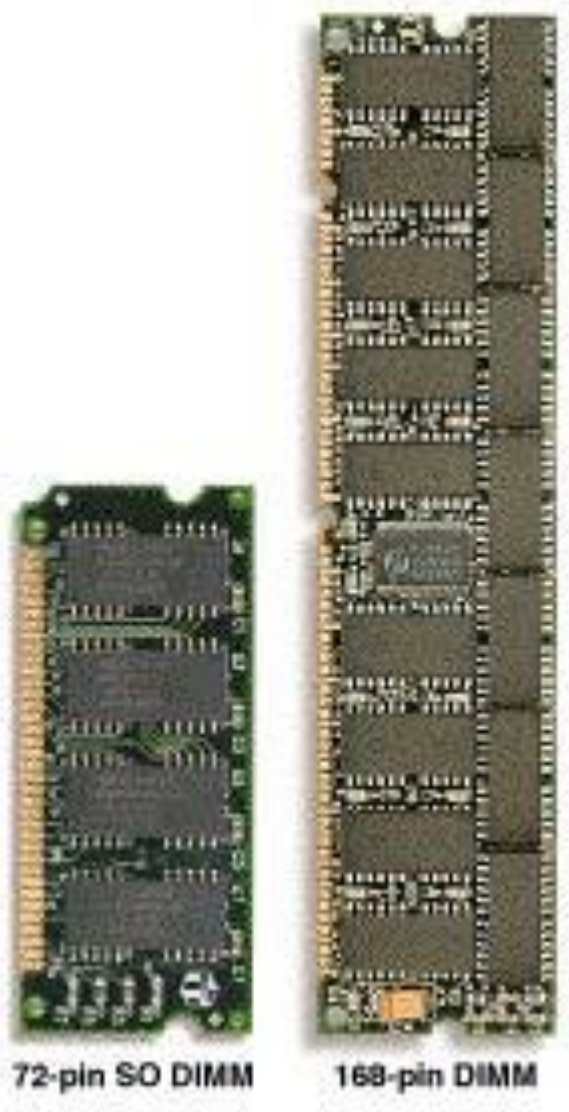
DRAM

- DRAM
  - 破坏性读：读后需要重新写回
  - 必须要周期性的刷新
  - 每位1个 transistor
  - 地址线复用:
    - Lower half of address: column access strobe (CAS)
    - Upper half of address: row access strobe (RAS)
- SRAM
  - 每位6个transistors
  - 只需较低的功率来保持位状态



DRAM Packaging
(Laptops/Desktops/Servers)

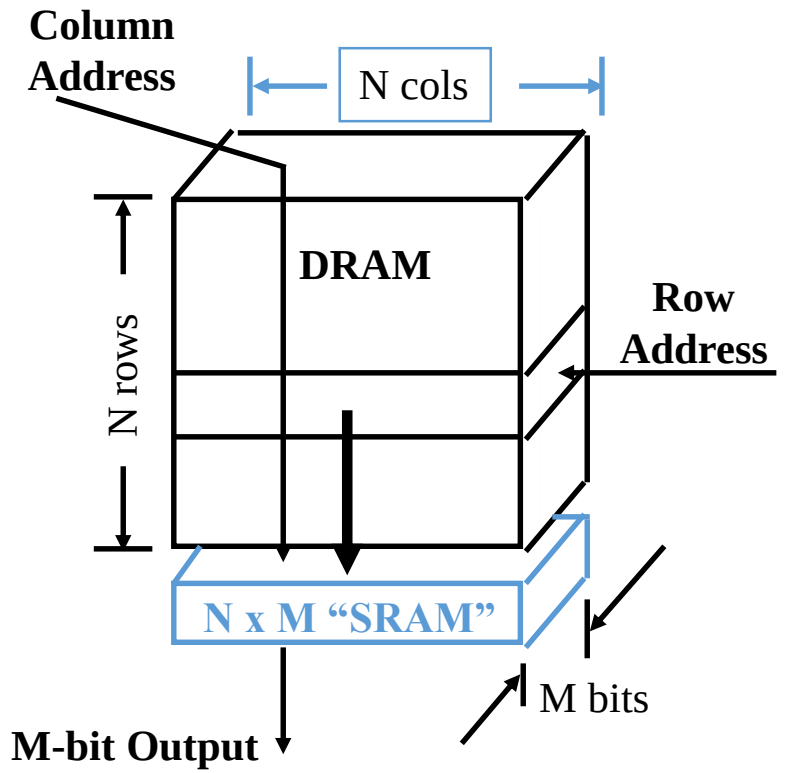


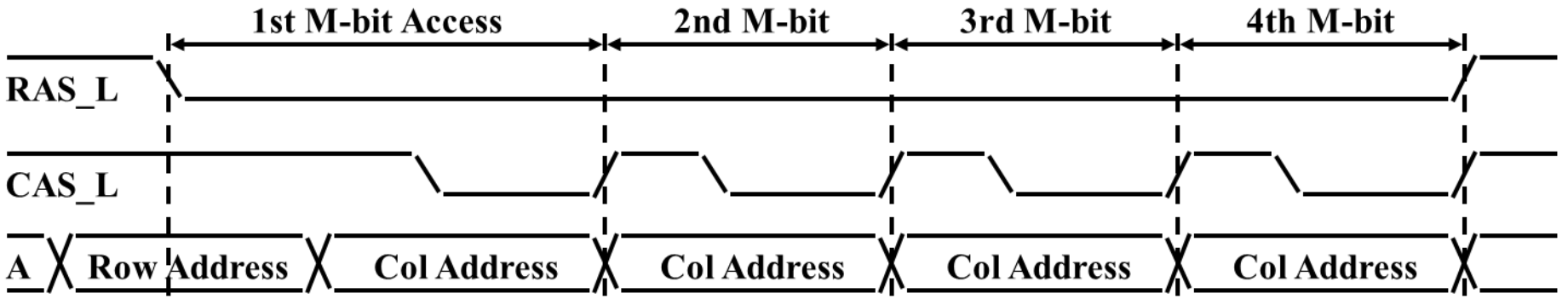


- DIMM (Dual Inline Memory Module) contains multiple chips with clock/control/address signals connected in parallel (sometimes need buffers to drive signals to all chips)
- Data pins work together to return wide word (e.g., 64-bit data bus using 16x4-bit parts)

Memory 优化

- Some optimizations:
  - Fast Page Mode Operation
    - Multiple accesses to same row
  - Synchronous DRAM
    - Added clock to DRAM interface
    - Burst mode with critical word first
  - Wider interfaces
  - Double data rate (DDR)
  - Multiple banks on each DRAM device





Memory Optimizations

Production year	Chip size	DRAM Type	Row access strobe (RAS)		Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
			Slowest DRAM (ns)	Fastest DRAM (ns)		
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

Figure 2.13 Times of fast and slow DRAMs vary with each generation. (Cycle time is defined on page 95.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs. The introduction of various burst transfer modes in the mid-1990s and SDRAMs in the late 1990s has significantly complicated the calculation of access time for blocks of data; we discuss this later in this section when we talk about SDRAM access time and power. The DDR4 designs are due for introduction in mid- to late 2012. We discuss these various forms of DRAMs in the next few pages.

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Figure 2.14 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2010. Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. Although not shown in this figure, DDRs also specify latency in clock cycles as four numbers, which are specified by the DDR standard. For example, DDR3-2000 CL 9 has latencies of 9-9-9-28. What does this mean? With a 1 ns clock (clock cycle is one-half the transfer rate), this indicate 9 ns for row to columns address (RAS time), 9 ns for column access to data (CAS time), and a minimum read time of 28 ns. Closing the row takes 9 ns for precharge but happens only when the reads from that row are finished. In burst mode, transfers occur on every clock on both edges, when the first RAS and CAS times have elapsed. Furthermore, the precharge in not needed until the entire row is read. DDR4 will be produced in 2012 and is expected to reach clock rates of 1600 MHz in 2014, when DDR5 is expected to take over. The exercises explore these details further.

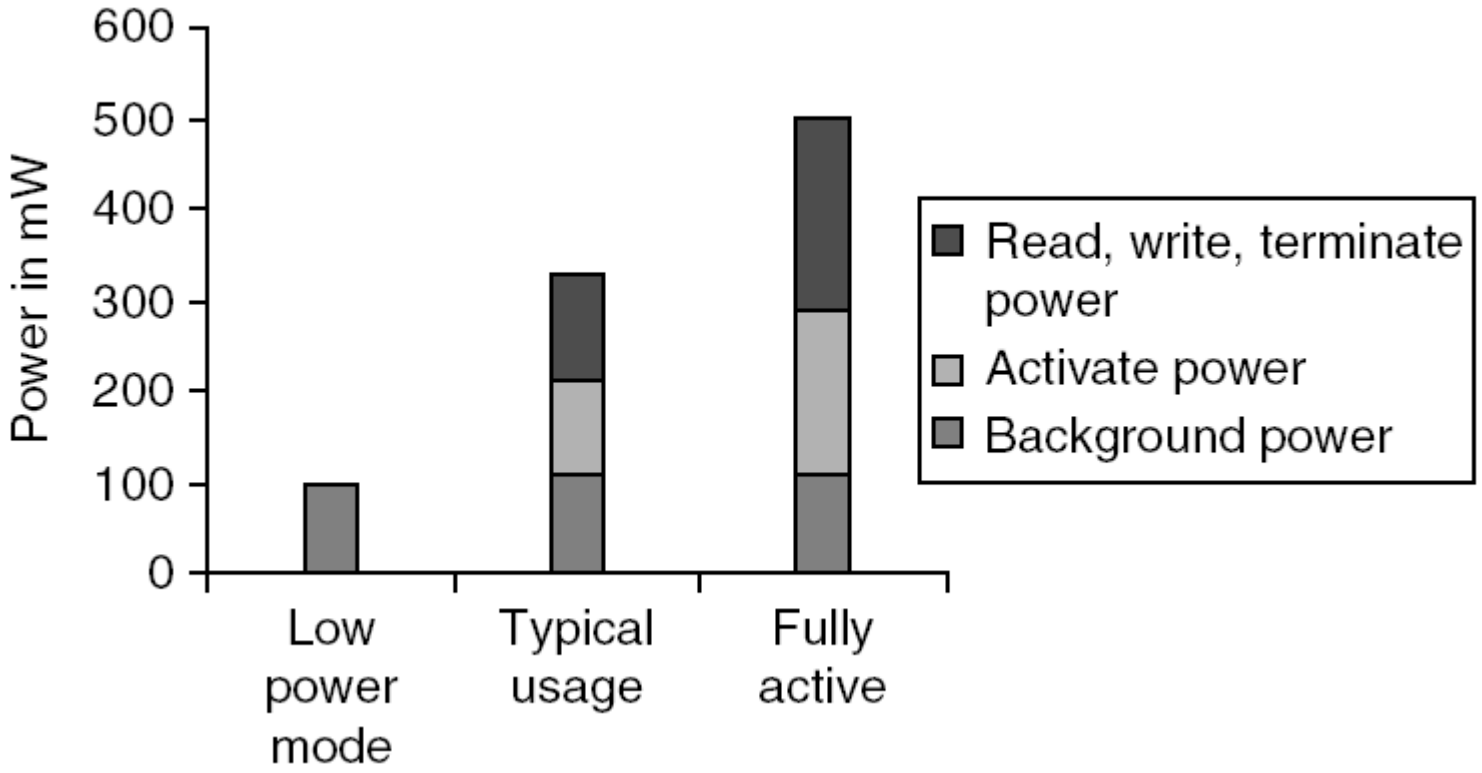
Memory Optimizations

- DDR:
  - DDR2
    - Lower power (2.5 V -> 1.8 V)
    - Higher clock rates (266 MHz, 333 MHz, 400 MHz)
  - DDR3
    - 1.5 V
    - 800 MHz
  - DDR4
    - 1-1.2 V
    - 1600 MHz
- GDDR5 is graphics memory based on DDR3

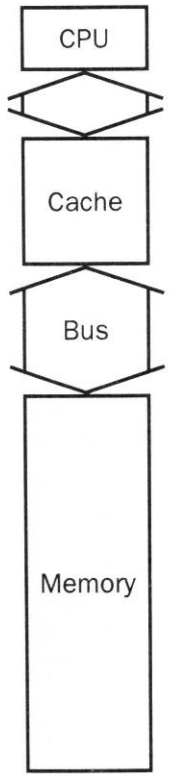
Memory Optimizations

- Graphics memory:
  - Achieve 2-5 X bandwidth per DRAM vs. DDR3
    - Wider interfaces (32 vs. 16 bit)
    - Higher clock rate
      - Possible because they are attached via soldering instead of socketted DIMM modules
- Reducing power in SDRAMs:
  - Lower voltage
  - Low power mode (ignores clock, continues to refresh)

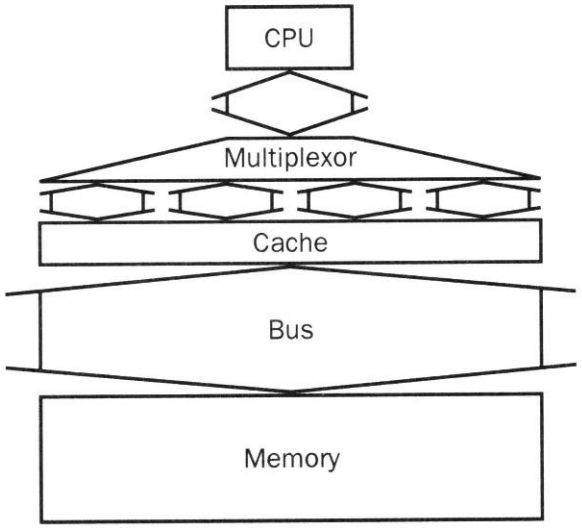
Memory Power Consumption



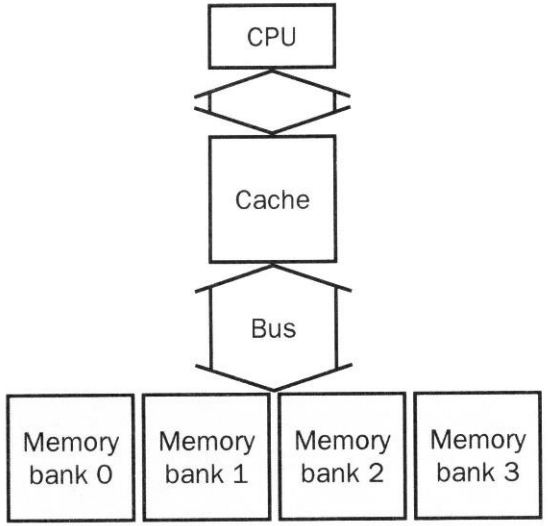
三种存储器组织方式



a. One-word-wide memory organization



b. Wide memory organization



c. Interleaved memory organization

Simple:

CPU, Cache, Bus, Memory same width (32 bits)

Wide:

CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits)

Interleaved:

CPU, Cache, Bus 1 word: Memory N Modules (4 Modules); example is *word interleaved*

提高主存性能的方法-增大存储器的宽度（并行访问存储器）

- 最简单直接的方法
- 优点：简单、直接，可有效增加带宽
- 缺点
  - 增加了CPU与存储器之间的连接通路的宽度，实现代价提高
  - 主存容量扩充时，增量应该是存储器的宽度
  - 写操作问题（部分写操作）
- 冲突问题
  - 取指令冲突，遇到程序转移时，一个存储周期中读出的n条指令中，后面的指令将无用
  - 读操作数冲突。一次同时读出的几个操作数，不一定都有用
  - 写操作冲突。这种并行访问，必须凑齐n个字之后一起写入。如果只写一个字，必须先把属于同一个存储字的数据读到数据寄存器中，然后在地址码的控制下修改其中一个字，最后一起写。
  - 读写冲突。当要读写的字在同一个存储字内时，无法并行操作。
- 冲突的原因
  - 从存储器本身看，主要是地址寄存器和控制逻辑只有一套。

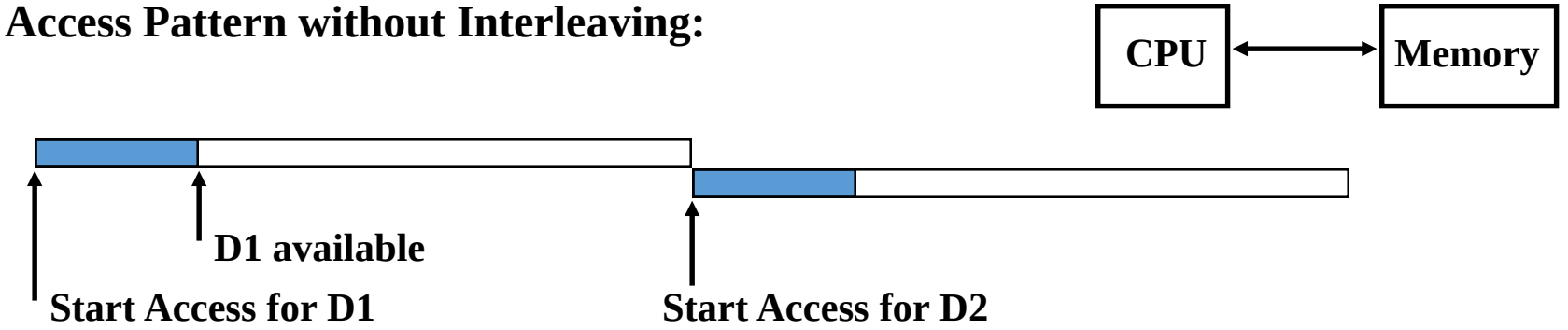
采用简单的多体交叉存储器

- 一套地址寄存器和控制逻辑
- 存储器芯片组织为多个体（Bank）
- 存储体的宽度，通常为一个字，不需要改变总线的宽度
- 目的：在总线宽度不变的情况下，完成多个字的并行读写

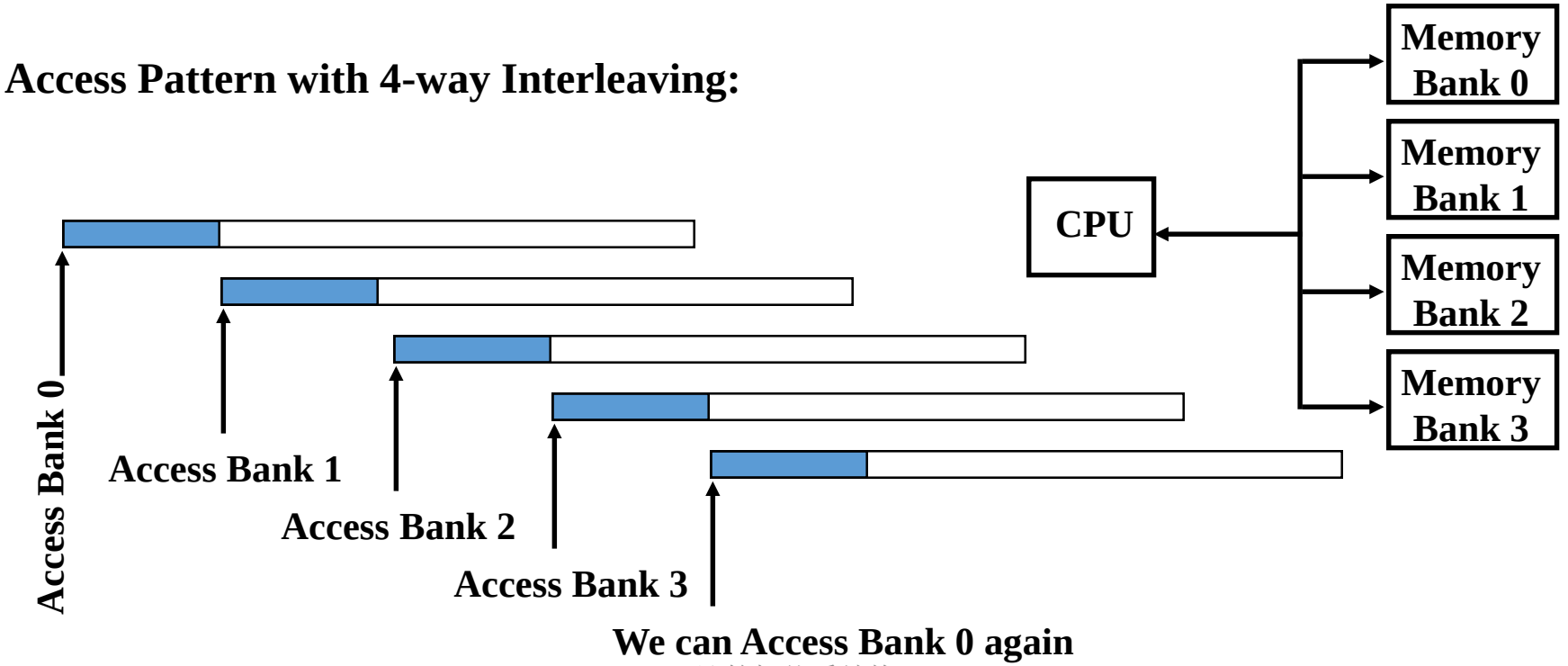
- 存储器中所包含的体数，为避免访问冲突，基本原则为：
  体的数目 >= 访问体中一个字所需的时钟周期数
  例如：某一向量机的存储系统，CPU发出访存请求10个时钟周期后，CPU将从存储体0得到一个字，随后体0开始读该存储体的下一个字，而CPU依次从其余7个存储体中得到后继的7个字。在第18个周期，CPU 将需要存储体0提供下一个字，但该字要到第20个时钟周期才被读出，CPU只好等待。
- 缺陷：不能对单个体单独访问，对解决冲突没有帮助，逻辑上是一种宽存储器，对各个存储体的访问被安排在不同的时间段

Increasing Bandwidth - Interleaving

Access Pattern without Interleaving:



Access Pattern with 4-way Interleaving:



地址映射方法（ m个存储体，每个存储体容量为n）

- 高位交叉编址：相当于对存储单元矩阵按列优先的方式进行编址

  考虑处于第 i 行第 j 列的单元，其线性地址为：

  A = j X n + i

  其中： j =0~m-1 ； i = 0~n-1

  已知：一个单元的线性地址为A， 则：

  j = A / n 取下限　　i = A mod n

  取线性地址高位 $\log_2 m$位就是体号，其余低位部分就是体内地址

- 低位交叉编址：相当于对存储单元矩阵按行优先的方式进行编址

  考虑处于第 i 行第 j 列的单元，其线性地址为：

  A = i X m + j

  其中： i = 0~n-1 ； j = 0~m-1

  已知：一个单元的线性地址为A， 则：

  i = A / m 取下限　　j = A mod m

  取线性地址低位 $\log_2 m$位就是体号，其余高位部分就是体内地址

例题：

- 举例：假设某台机器的特性及其Cache的性能为：
  - 块大小为1个字
  - 存储器总线宽度为1个字
  - Cache失效率为3%
  - 平均每条指令访存1.2次
  - Cache失效开销为32个时钟周期
  - 平均CPI（忽略Cache失效）为2
- 试问多体交叉和增加存储器宽度对提高性能各有何作用？

- 假设：送地址：4cycles,
  每个字的访问时间为24cycles,
  传送一个字的数据需要4cycles

如果当把Cache块大小变为2个字时，失效率降为2%；块大小变为4个字时，失效率降为1%。
根据前面给出的访问时间，求在采用2路、4路多体交叉存取以及将存储器和总线宽度增加一倍时，性能分别提高多少?

在改变前的机器中，Cache块大小为一个字，其CPI为：

$$2+(1.2\times3\%\times32)=3.15$$

当将块大小增加为2个字时，在下面三种情况下的CPI分别为：

32位总线和存储器，不采用多体交叉：

2＋(1.2×2%×2×32)＝3.54

32位总线和存储器，采用多体交叉：

2＋(1.2×2%×(4＋24＋8))＝2.86

性能提高了10%

64位总线和存储器，不采用多体交叉：

2＋(1.2×2%×1×32)＝2.77

性能提高了14%

如果将块大小增加到4个字，则：

32位总线和存储器，不采用多体交叉：

$2+(1.2\times1\%\times4\times32)=3.54$

32位总线和存储器，采用多体交叉：

$2+(1.2\times1\%\times(4+24+16))=2.53$

性能提高了25%

64位总线和存储器，不采用多体交叉：

$2+(1.2\times1\%\times2\times32)=2.77$

性能提高了14%

4.6 虚拟存储器—基本原理

- 允许应用程序的大小，超过主存容量。目的是提高存储系统的容量
- 帮助OS进行多进程管理
  - 每个进程可以有自己的地址空间
  - 提供多个进程空间的保护
  - 可以将多个逻辑块映射到共享的物理存储器上
  - 静态重定位和动态重定位
    - 应用程序运行在虚地址空间
    - 虚实地址转换对用户是透明的
- 虚拟存储管理的是主存—辅助存储器这个层面上
  - 失效：页失效或地址失效
  - 块：页或段

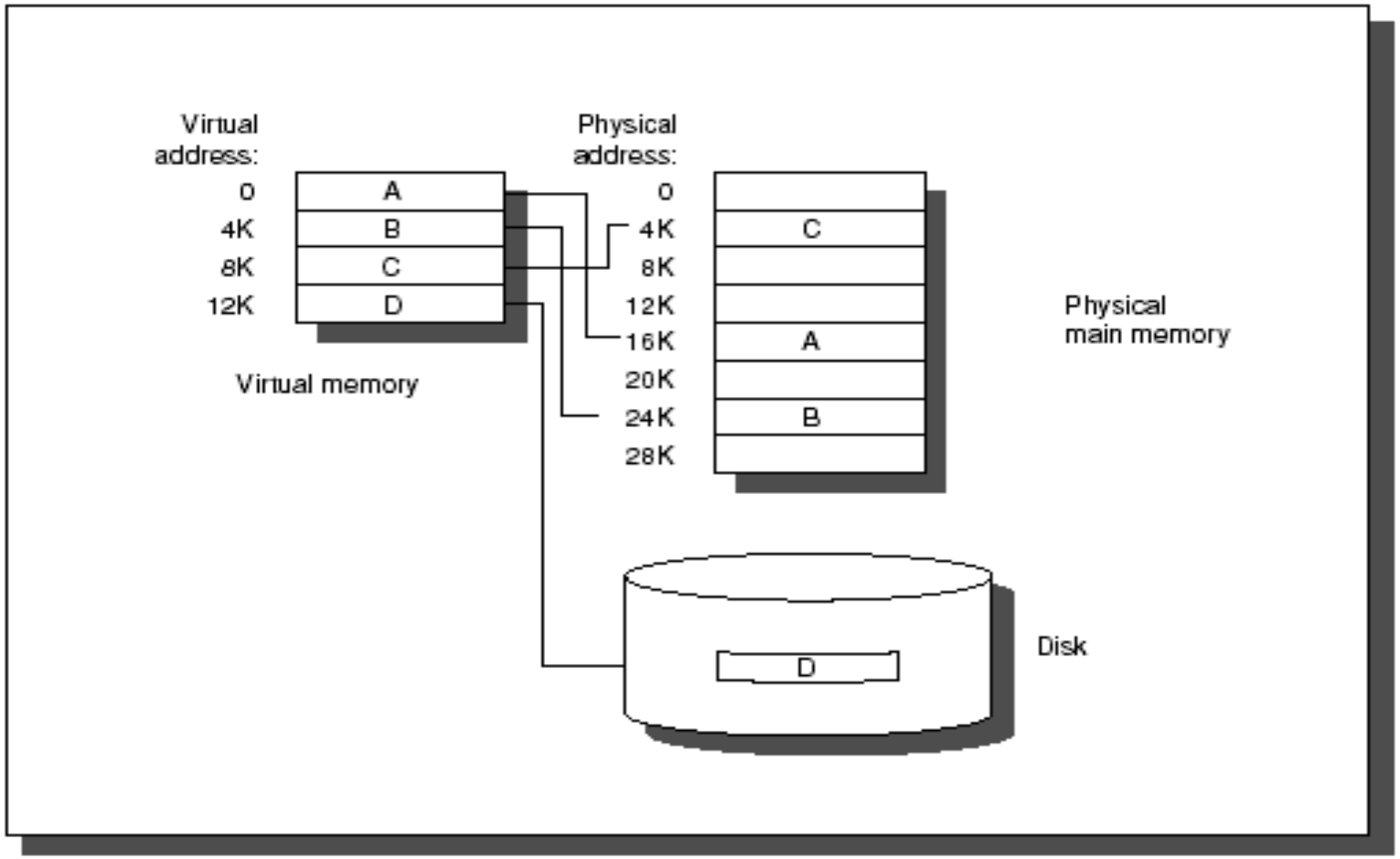


FIGURE 5.31 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

Cache与VM的区别

- 目的不同
  - Cache是为了提高访存速度
  - VM是为了提高存储容量
- 替换的控制者不同
  - Cache失效由硬件处理
  - VM的页失效通常由OS处理
    - 一般页失效开销很大，因此替换算法非常重要
- 地址空间
  - VM空间由CPU的地址尺寸确定
  - Cache的大小与CPU地址尺寸无关
- 下一级存储器
  - Cache下一级是主存
  - VM下一级是磁盘，大多数磁盘含有文件系统，文件系统寻址与主存不同，它通常在I/O空间中，VM的下一级通常称为SWAP空间

虚拟存储器页式管理的典型参数与Cache的比较

Parameter	First-level cache	Virtual memory
Block (page) size	16-128 bytes	4096-65,536 bytes
Hit time	1-3 clock cycles	50-150 clock cycles
Miss penalty	8-150 clock cycles	1,000,000-10,000,000 clock cycles
(Access time)	(6-130 clock cycles)	(800,000-8,000,000 clock cycles)
(Transfer time)	(2-20 clock cycles)	(200,000-2,000,000 clock cycles)
Miss rate	0.1-10%	0.00001- 0.001%
Address mapping	25- 45 bit physical address to 14- 20 bit cache address	32-64 bit virtual address to 25-45 bit physical address

FIGURE 5.32 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10 to 1,000,000 times over cache parameters. Normally first level caches contain at most 1 megabyte of data while physical memory contains 32 megabytes to 1 terabyte.

- 从表中看 （与Cache参数相比）
  - 除了失效率较低，其他参数都比Cache大

页式管理和段式管理

Aspect	Page	Segment
Words/Address	One - contains page and offset	Two - possible large max-size hence need Seg and offset address words
Programmer visible	No	Sometimes yes
Replacement	Trivial - due to fixed size	Hard - need to find contiguous space ==> GC necessary or wasted memory
Memory Inefficiency	Internal fragmentation - wasted part of a page	External fragmentation - due to variable size blocks
Disk Efficiency	Yes - adjust page size to balance access and transfer time	Not always - segment size varies

- VM可分为两类：页式和段式
  - 页式：每页大小固定
  - 段式：每段大小不等
  - 两者区别：

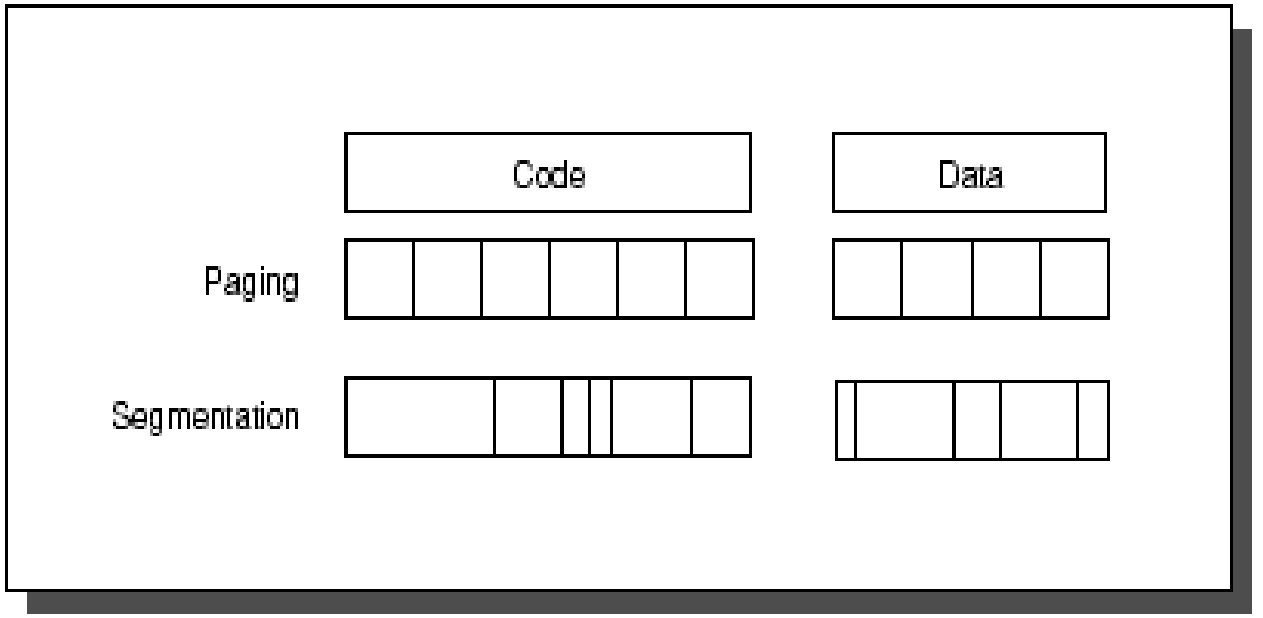


FIGURE 5.33 Example of how paging and segmentation divide a program.

VM的四个问题（1/2）

- 映象规则
  - 选择策略：低失效率和复杂的映象算法，还是简单的映射方法，高失效率
    - 由于失效开销很大，一般选择低失效率方法，即全相联映射
- 查找算法—用附加数据结构
  - 固定页大小—用页表
    - VPN —> PPN
    - Tag标识该页是否在主存
  - 可变长段 —段表
    - 段表中存放所有可能的段信息
    - 段号 —>段基址 再加段内偏移量
    - 可能由许多小尺寸段
  - 页表
    - 页表中所含项数：一般为虚页的数量
    - 功能: VPN—>PPN，方便页重新分配，有一位标识该页是否在内存

VM的四个问题（2/2）

- 替换规则
  - LRU是最好的
  - 但真正的LRU方法，硬件代价较大
  - 用硬件简化，通过OS来完成
    - 为了帮助OS寻找LRU页，每个页面设置一个 use bit
    - 当访问主存中一个页面时，其use bit置位
    - OS定期复位所有使用位，这样每次复位之前，使用位的值就反映了从上次复位到现在的这段时间中，哪些页曾被访问过。
    - 当有失效冲突时，由OS来决定哪些页将被换出去。
- 写策略
  - 总是用写回法，因为访问硬盘速度很慢。

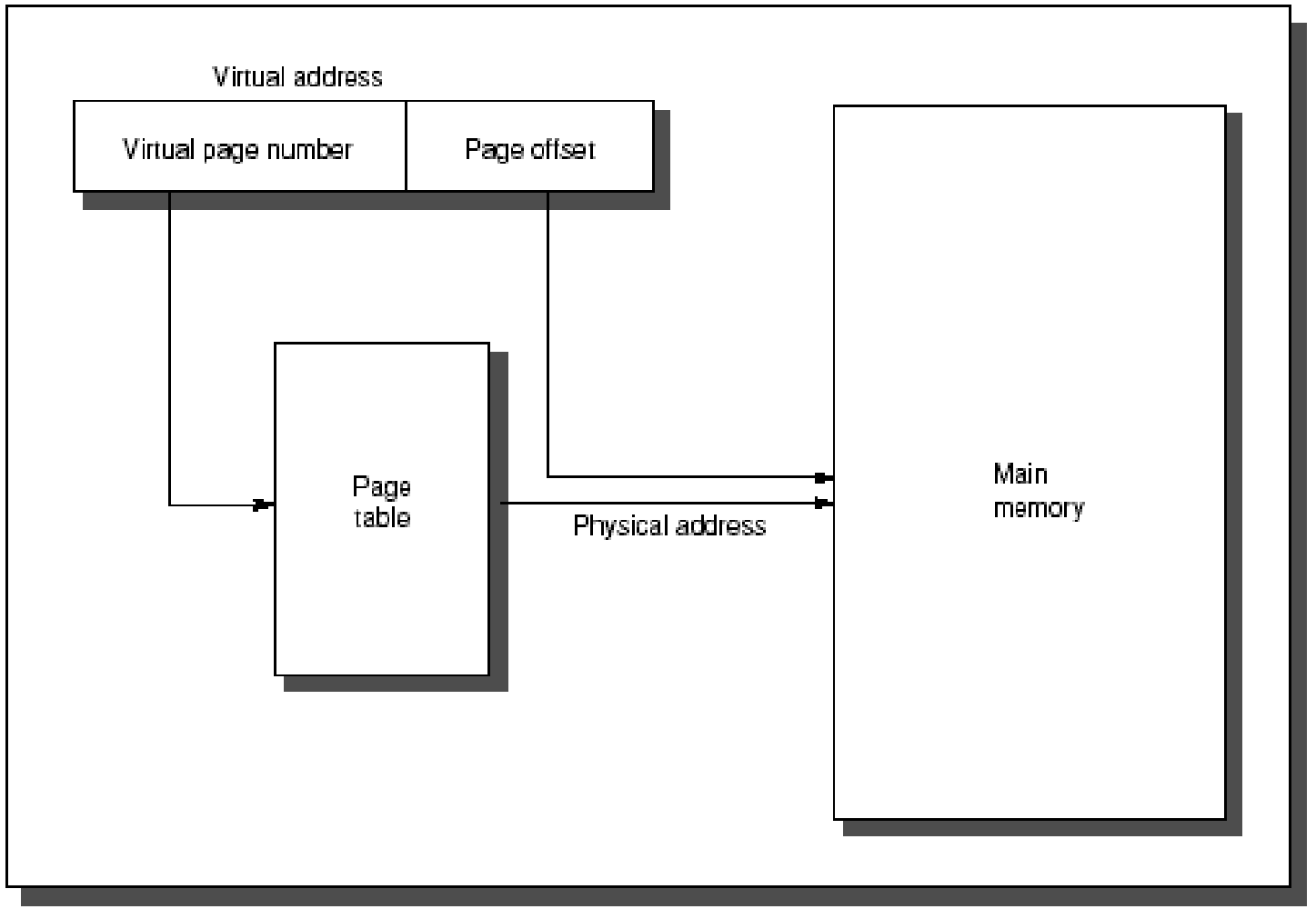
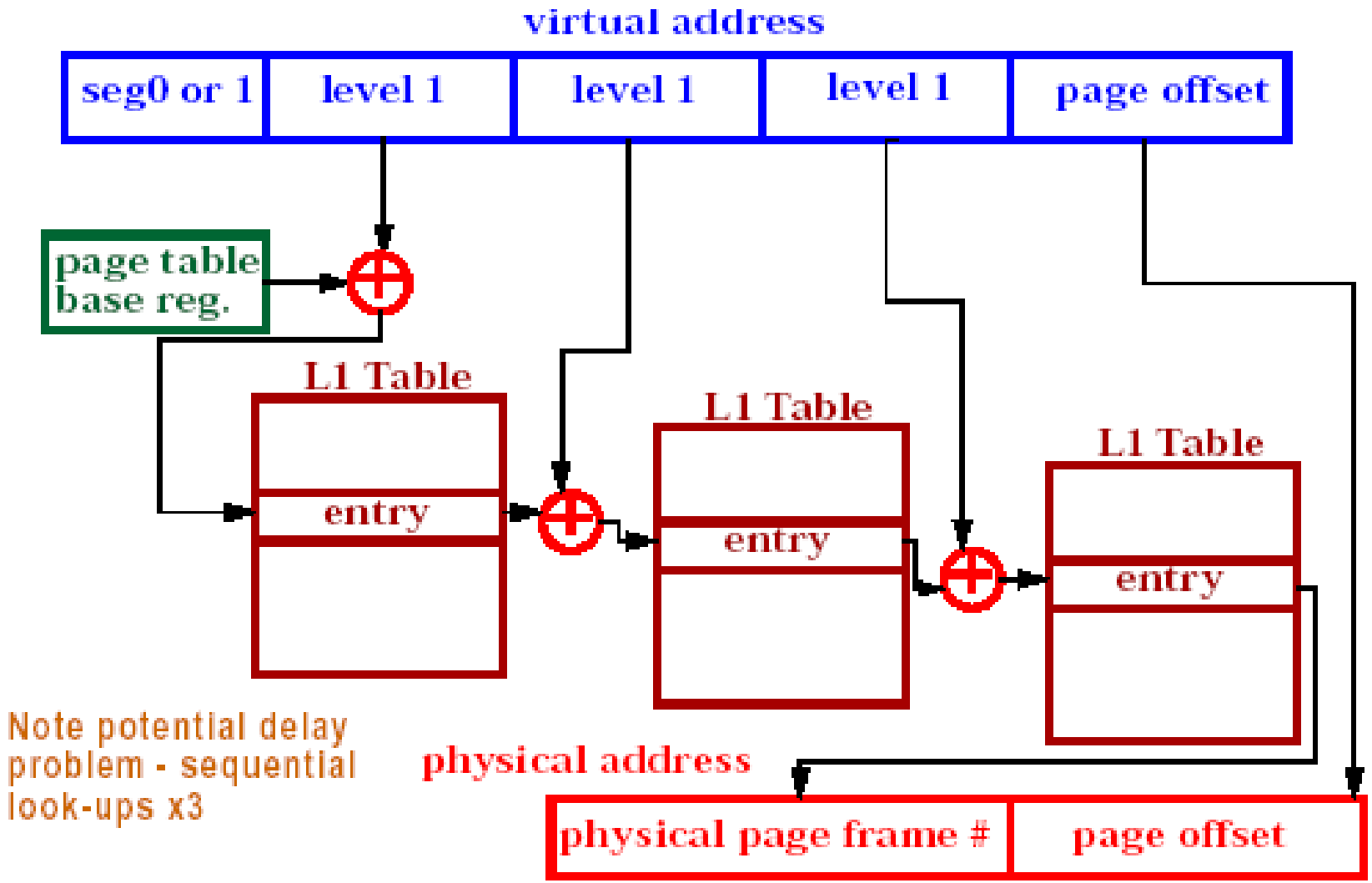


FIGURE 5.35 The mapping of a virtual address to a physical address via a page table.

页面大小的选择

- 页面选择较大的优点
  - 减少了页表的大小
  - 如果局部性较好，可以提高命中率
- 页面选择较大的缺点
  - 内存中的碎片较多，内存利用率低
  - 进程启动时间长
  - 失效开销加大


virtual address
seg0 or 1
level 1
level 1
level 1
page offset
page table base reg.
L1 Table
entry
L1 Table
entry
L1 Table
entry
Note potential delay problem - sequential look-ups x3
physical address
physical page frame #
page offset

TLB （Translation look-aside Buffer)

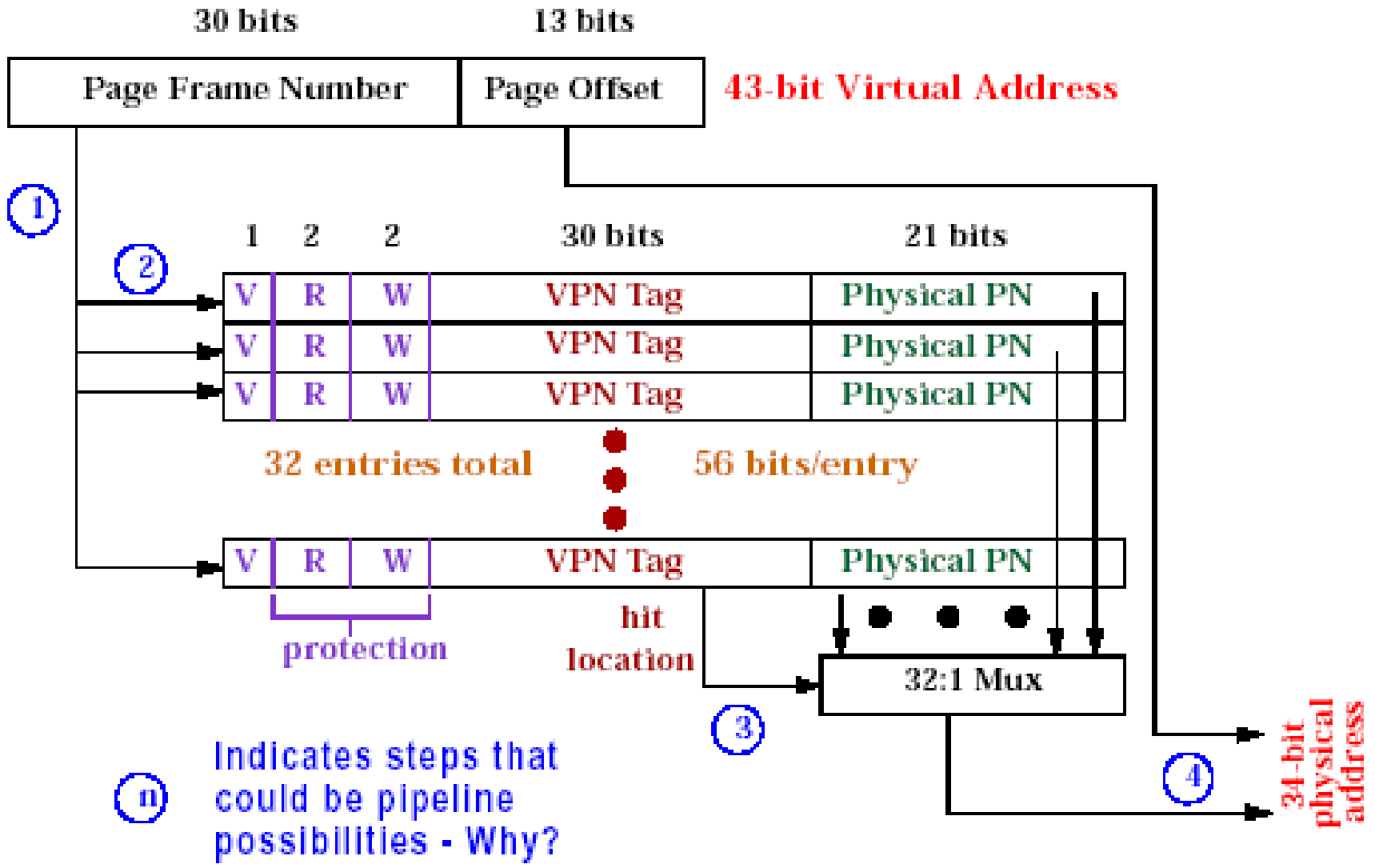
- 页表一般很大，存放在主存中。
  - 导致每次访存可能要两次访问主存，一次读取页表项，一次读写数据
  - 解决办法：采用 TLB
- TLB
  - 存放近期经常使用的页表项，是整个页表的部分内容的副本。
  - 基本信息：

    VPN##PPN##Protection Field##use bit ## dirty bit
  - OS修改页表项时，需要刷新TLB，或保证TLB中没有该页表项的副本
  - TLB必须在片内
    - 速度至关重要
    - TLB过小，意义不大
    - TLB过大，代价较高
    - 相联度较高（容量小）

TLB的典型参数

- block size - same as a page table entry - 1 or 2 words
- hit time - 1 cycle
- miss penalty - 10 to 30 cycles
- miss rate - .1% to 2%
- TLB size - 32 B to 8 KB

举例：Alpha 21064的TLB



Summary of Virtual Memory and Caches

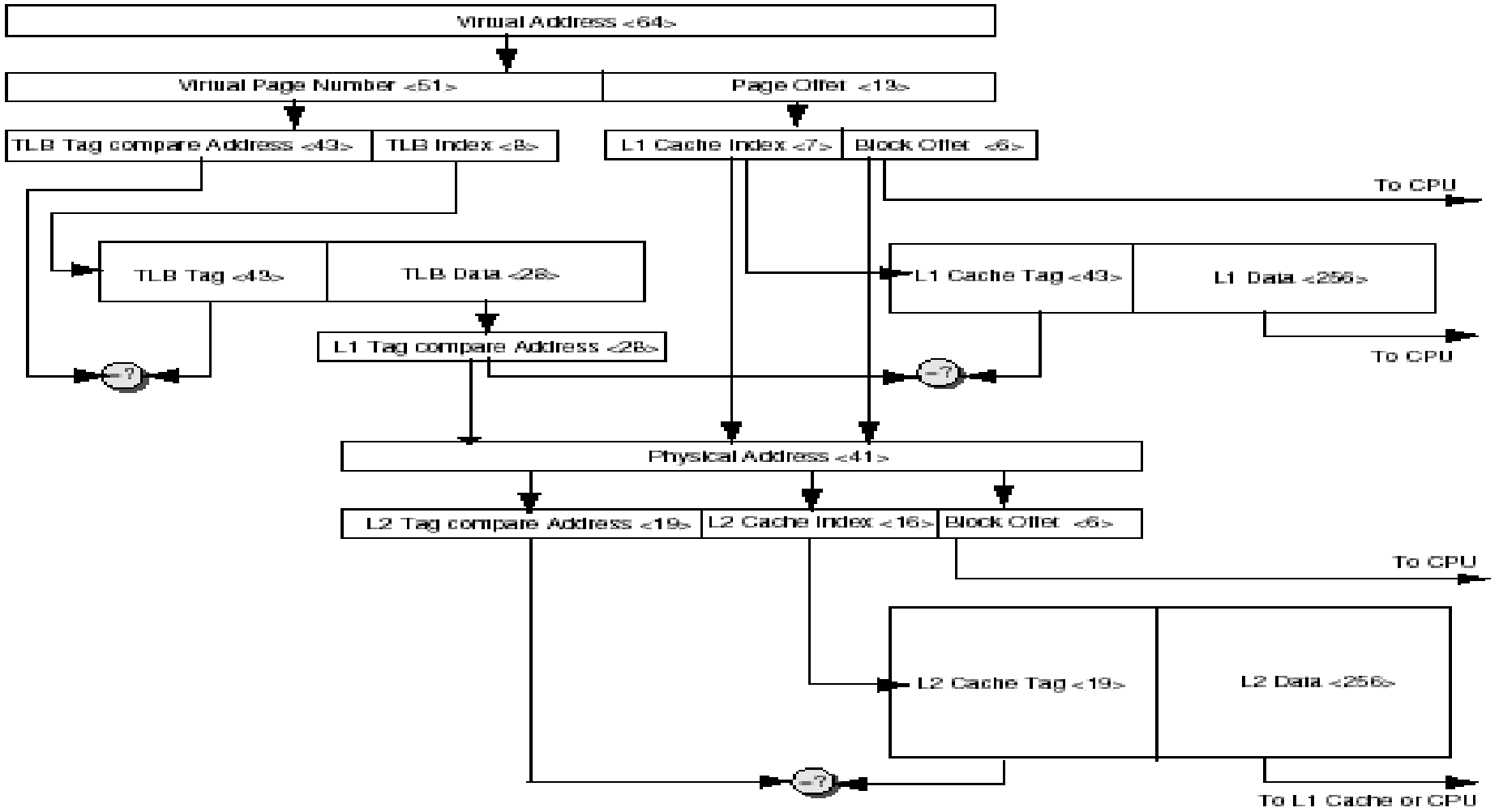


FIGURE 5.37 The overall picture of an hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB and the L2 cache is a direct-mapped 4 MB. Both using 64 byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache, as in Figure 5.43 on page 472 , is replication of pieces of this figure.