Lecture 12 内存的性能和可靠的内存层次结构

1. cache 性能评估

cache 的性能公式

CPU 时间可以划分为 CPU 执行程序花费的时钟周期和 CPU 等待存储系统花费的时钟周期，通常来说，我们假定 cache 访问命中的开销是 CPU 正常执行周期的一部分

$$CPU\text{时间} = (CPU\text{执行时钟周期数} + \text{存储器阻塞时钟周期数}) \times \text{时钟周期}$$

我们假设存储器阻塞时钟周期数主要来源于 cache 的缺失

存储器阻塞的时钟周期数可以被定义为读操作和写操作所引起的足额的时钟周期之和

$$\text{存储器阻塞时钟周期数} = (\text{读操作引起的阻塞时钟周期数} + \text{写操作引起的阻塞时钟周期数})$$

出于简化模型，我们定义

$$\text{读操作时钟周期数} = \frac{\text{读的次数}}{\text{程序数}} \times \text{读缺失率} \times \text{读缺失代价}$$
$$\text{写操作时钟周期数} = \frac{\text{写的次数}}{\text{程序数}} \times \text{写缺失率} \times \text{写缺失代价}$$

如果读操作和写操作的缺失率和缺失代价一样，我们可以定义

$$\text{存储器阻塞时钟周期数} = \frac{\text{存储器访问次数}}{\text{程序数}} \times \text{缺失率} \times \text{缺失代价}$$

也可以表示成

$$\text{存储器阻塞时钟周期数} = \frac{\text{指令数}}{\text{程序数}} \times \frac{\text{缺失数}}{\text{指令}} \times \text{缺失代价}$$

例题1 计算 cache 的性能

- I-cache miss rate = 2% （指令 cache）
- D-cache miss rate = 4% （数据 cache）
- miss penalty = 100 cycles （缺失代价）

- Base CPI = 2
- Load & Store 占比 36%（访问数据 cache 的概率）

计算 CPI 的大小

根据题目，假设指令有 I 条

由指令引起的时钟周期损失数为：

$$\text{指令缺失时钟周期数} = I \times 2\% \times 100 = 2.00 \times I$$

由数据缺失引起的时钟周期数为：

$$\text{数据缺失时钟周期数} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

总阻塞时钟周期数为 $2.00I + 1.44I = 3.44I$，包括阻塞在内的 CPI 总和为 $2 + 3.44 = 5.44$，在理想情况下，$CPI = 2$

所以，如果配置一个理想的 cache，处理器的性能为

$$\frac{5.44}{2} = 2.72$$

我们下面的情况

如果处理器速度很快，而存储系统速度不快？

- 假设 CPI = 1，那么具有 cache 缺失的系统的总 CPI 为 $1 + 3.44 = 4.44$，而配置理想 cache 的系统性能是它的 4.44 倍
- 存储器阻塞所花费的时间占比为 $\frac{3.44}{4.44} = 77\%$，而原来是 $\frac{3.44}{5.44} = 63\%$

同样，仅仅提高时钟频率而不改进存储系统也会因 cache 缺失的增加而加剧性能的流失

平均存储器访问时间 AMAT

有时候会用 AMAT 作为检测 cache 设计的方法，

$$\begin{aligned} AMAT &= \text{命中时间} + \text{缺失率} \times \text{缺失代价} \\ AMAT &= \text{命中率} \times \text{命中时间} + \text{缺失率} \times (\text{命中时间} + \text{缺失代价}) \end{aligned}$$

缺失率 = 1 - 命中率

例题2 计算平均存储器访问时间

处理器的时钟周期为 1ns，缺失代价是 20 个时钟周期，缺失率为每条指令 0.05 次缺失，cache 访问时间（包括命中判断）为 1 个时钟周期，计算 AMAT

$$AMAT = 1 + 0.05 \times 20 = 2\text{个时钟周期} = 2ns$$
$$AMAT = 0.95 \times 1 + 0.05 \times (1 + 20) = 2(cycle) = 2ns$$

2. 更灵活的放置 block 来减少 cache 缺失

映射

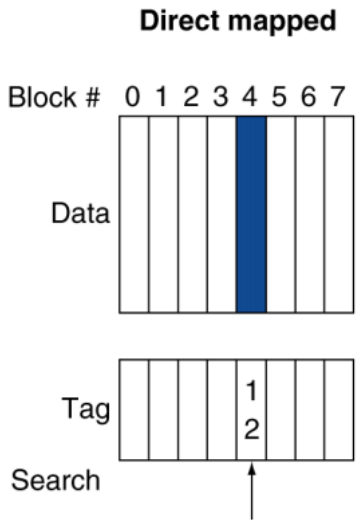
直接映射 direct mapped

前面我们将的都是**直接映射**，一个块只能放到 cache 中一个明确的位置，事实上，一共有三种放置方式

存储器中任何一块都被直接映射到存储器层次结构中较高层的唯一位置

直接映射中，一个存储块的位置是定义是

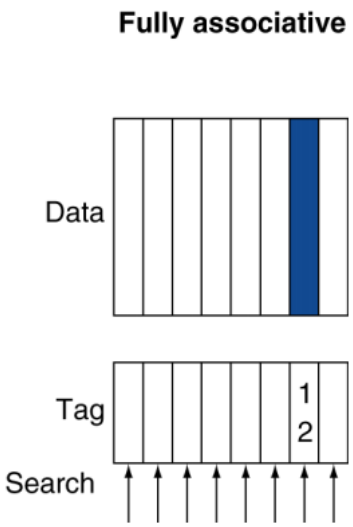
$$(\text{块号})\ mod\ (cache\text{中的块数})$$



这里 $12\ mod\ 8 = 4$，所以放在第 4 块

全相联 full associative

存储器中的块可以放在 cache 中的任何一个位置



组相联 set associative

在组相联 cache 中，每个 block 可被放置的位置数是固定的，每个 block 有 n 个位置可放的 cache 被称作 n 路组相联 cache （**n-way set associative**）

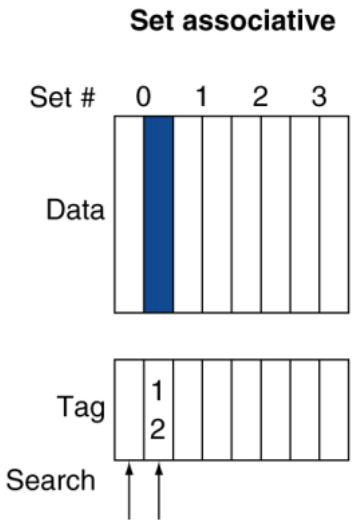
一个 n 路组相联 cache 由很多个组构成，每个组有 n 个 block

根据 index，存储器中每个block 对应到 cache 中唯一的组，并且可以放置在该组任何位置上

检索一个 block 是否在 cache 上时，需要检索整个组内的所有 block 的 tag 来判断

组相连 cache 中，包含存储块的组的编号为

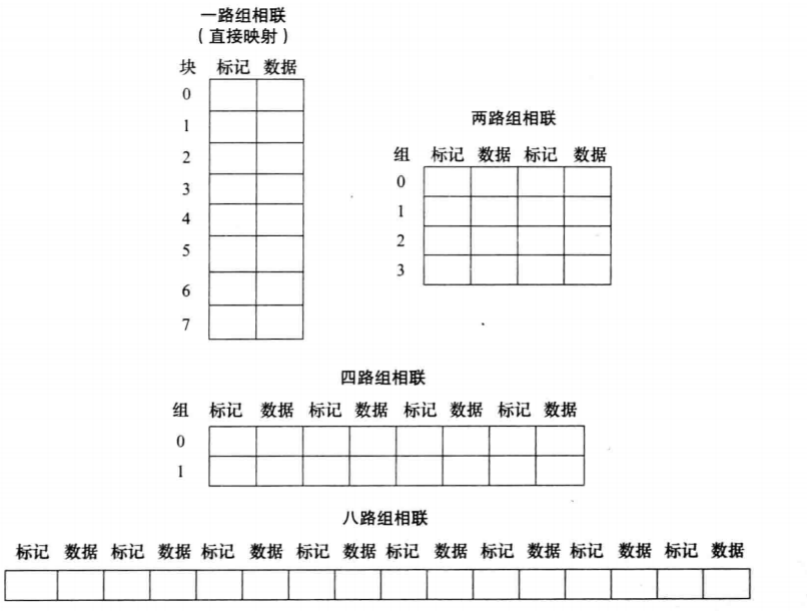
$$(\text{块号})\ mod\ (cache\text{中的组数})$$



这里 12 mod 4 = 0，可以放在第 0 组（两块）中的任意位置

映射示例

假设一个 cache 有 8 个 block



例题3 不同的映射

假设有一个 4 个 block 的cache，分别使用 direct mapped、2-way set associative 和 fully associative，比较 cache 的 hit 次数

- 假设访问地址为：0、8、0、6、8

direct map

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0	miss	Mem[0]	Mem[6]		
8	0	miss	Mem[8]	Mem[6]		

fully associative

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

替换 block 的选择

在相连的 cache 中，被请求的块放置在什么位置需要进行选择，因此替换哪一块也要进行选择

最常用的方法是**最近最少使用 Least Recently Used， LRU** 法，被替换掉的块是最久没有使用的那一块

LRU 替换算法的实现是通过跟踪每一块的相对使用情况

在组相联 cache 中查找一个块

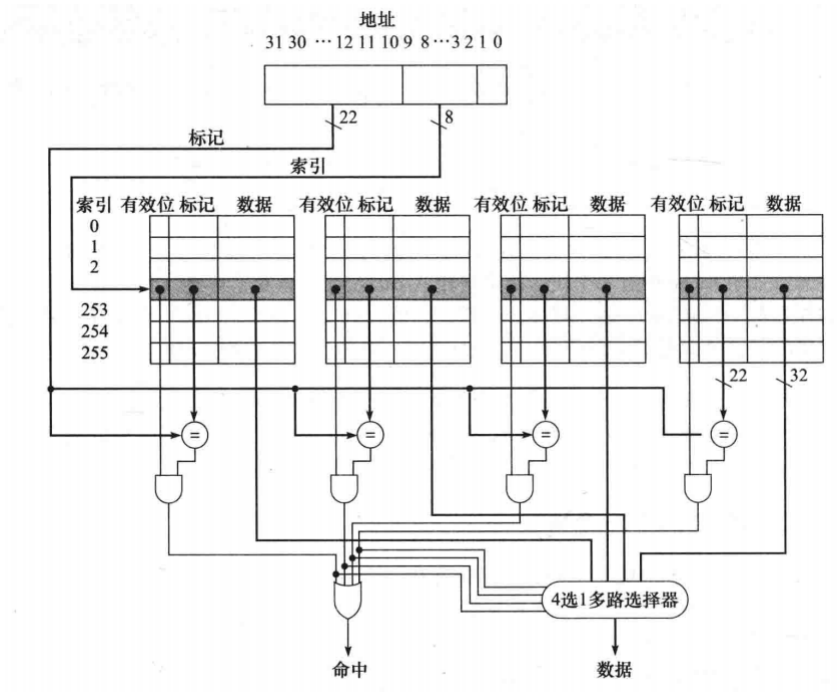
现在我们考虑组相连 cache 中如何查找一个 cache 块，cache 中的地址由三部分组成

- tag：用来和选中组中的块进行比较来选择块
- index：用来标识一个组，该组中的所有的块的 tag 都会被检索
- offset：块中被请求数据的地址

在组相联中，index 所占的 bit 数量**等于组相连中的组数**，其它计算与直接相连保持不变，offset 都是根据一个 block 等于多少个 byte 来确定的

在全相联 cache 中，只有一组有效，故没有 index 部分，整个 address 都需要和每个 cache 块中的 tag 进行比较

下图是一个四路组相连 cache，需要 4 个比较器以及一个 4 选 1 多路选择器，用来在选定组中 4 个成员之间进行选择



相比直接映射 cache， 4 路组相联 cache

- 需要 4 个比较器
- 需要一个 4 选 1 多路选择器

一个组相连 cache 的开销包括

- 额外的比较器
- 由于对组内的 block 进行比较和选择而产生的延迟

例题4 tag 大小和组相连

假设一个 cache 有 4096 个 block，block 的大小为 4 个 word，总地址位为 32 bit

请分别计算直接映射、两路组相连、四路组相联和全相连映射中，cache 的总组数以及比总的 tag 位数

由于 1 block = 4 word = 16 byte，故 offset = $log_2 16 = 4(bit)$

直接映射

- index = $log_2 4096 = 12(bit)$
- tag = $32 - 4 - 12 = 16(bit)$

故总的 tag 位数为 $16 \times 4096 = 65536(bit)$

两路组相连

两路组相联中，一共有 2048 个组

- index = $log_2 2048 = 11(bit)$
- tag = $32 - 4 - 11 = 17(bit)$

故总的 tag 位数为 $17 \times 4096 = 69632(bit)$

四路组相联

四路组相联中，一共有 1024 个组

- index = $log_2 1024 = 10(bit)$
- tag = $32 - 4 - 10 = 18(bit)$

故总的 tag 位数为 $18 \times 4096 = 73728(bit)$

全相联

全相联中，只有 1 个组，故没有 index

- tag = $32 - 4 = 28$

故总的 tag 位数为 $28 \times 4096 = 114688(bit)$

3. 使用多级 cache 结构减少损失代价

多级 cache 的层次结构

为了减少现代处理器高时钟频率和日益增长的 DRAM 访问时间之间的差距，大多数微处理器都会额外增加一级 cache，这种二级 cache 通产谓语芯片内，当一级 cache 缺失的时候就访问它

- Primary cache 与 CPU 相连
    - 很小，但是很快
- Level-2 cache 为 primary cache 的缺失提供服务
    - 更大、更慢，但还是比主存快
- Main memory 为 Level-2 cache 的缺失提供服务

当然，有些高级的系统有三层 cache

例题5 计算多级 cache 的性能

- base CPI = 1
- clock rate = 4GHz
- miss rate/instruction = 2%
- main memory access time = 100ns

计算如果只有一个 cache 的情况

$\frac{1}{4\times10^9} = 0.25ns = cycle\ time$

所以，miss penalty = $\frac{100ns}{0.25ns} = 400(cycle)$

假设指令数为 I，则因为访问存储器所需要的时间为

$$I \times 2\% \times 400 = 8I$$

所以有效的 CPI 为 1 + 8 = 9

- base CPI = 1
- clock rate = 4GHz
- miss rate to L-2/instruction = 2%
- L-2 cache access time = 5ns
- miss rate to main memory/instruction = 0.5%
- main memory access time = 100ns

L-2 cache 的 miss penalty = $\frac{5ns}{0.25ns} = 20(cycle)$

假设指令数为 I，则访问 L-2 cache 所需要的时间为

$$I \times 2\% \times 20 = 0.4I$$

访问 main memory 所需要的时间为

$$I \times 0.5\% \times 400 = 2I$$

所以有效的 CPI = 1 + 0.4 + 2 = 3.4

性能相比只有一层的 cache 高 $\frac{9}{3.4} = 2.6$

多级 cache 的性能权衡

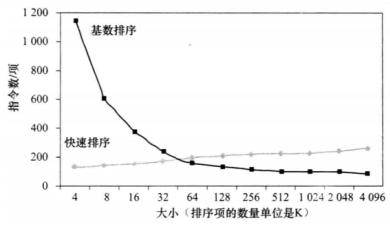
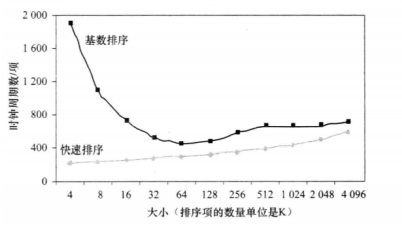
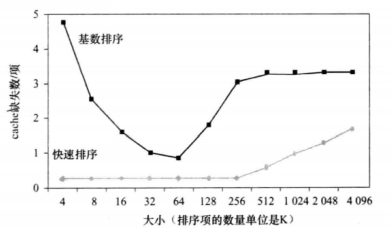
- Primary cache 尽量减少 hit time
- L-2 cache 主要减少 miss rate，hit time 不是非常重要

4. 采用软件优化技术

软件的优化主要是使用算法优化时间复杂度和 cache 的命中次数

排序问题

我们希望找到最好的排序方法，下面比较基数排序 Radix Sort 和快速排序 Quick Sort 两种算法

指令数	耗时（时钟周期）	CACHE 的缺失数
		

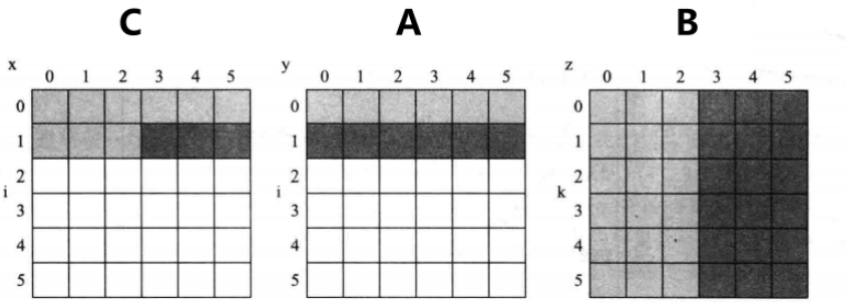
随着排序数据的增加，基数排序的指令数逐渐小于快速排序，可是为什么耗时还是比快速排序多呢？

- 原因随着排序数据的增加，基数排序 cache 缺失次数有一个明显的转折，而快速排序一直有比基数排序少得多的每项缺失数，访问内存耗费了很大的时间

矩阵乘法

原始算法

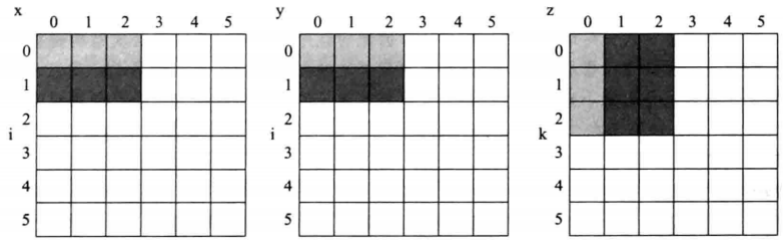
矩阵乘法中，程序读取了数组 B 中所有的 $N \times N$ 个元素，另外反复读取数组 A 对应行中的 N 个元素，并对数组 C 中对应行的 N 个元素进行了写操作



缺失次数依赖 N 和 cache 的容量，如果 cache 的容量很小，访问 memory 的次数需要很多次

分块算法

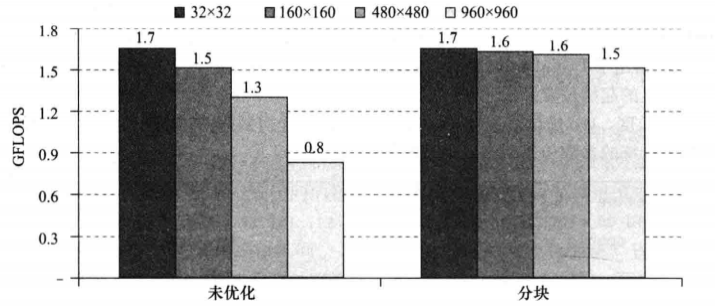
为了保证 cache 的命中率，可以把原先程序改为每次循环迭代指计算一个子矩阵



这样一个 block 可以更多的停留在 cache 中

性能比较

下图表示了采用两种算法的性能随着矩阵大小变化的情况



当矩阵尺寸增大到不能在 cache 中完全容纳这三个尺寸时

- 未优化算法：性能下降为最优情况的一半
- 分块算法：性能仅下降了 10%

5. 可靠性 Dependability

可靠性的衡量

可靠性是一个系统或模块能够持续提供用户需求的服务的度量

可靠性 Reliability

- 平均无故障时间 Mean Time To Failure（MTTF）
  从一次错误结束到下一次错误产生间隔时间的平均值（从开始使用到失效的时间间隔）
- 年失效率 Annual Failure Rate（AFR）：
  给定 MTTF 的情况下，一年内预期的器件失效比例
- 维修平均时间 Mean Time To Repair （MTTR）
  从一次错误开始到它结束产生的间隔时间的平均值（故障时间长度）
- 失效间隔平均时间 Mean Time Between Failures（MTBF）
  - MTBF = MTTF + MTTR
- 可用性 Availability = $\frac{MTTF}{MTTF+MTTR}$

如何增强 MTTF

- 故障避免 fault avoidance
  通过合理构建系统来避免故障的出现
- 故障容忍 fault tolerance
  采用冗余措施，当发生故障时，通过冗余错是来保证系统正常工作
- 故障预报 fault forecasting
  对故障进行预测，从而允许在器件失效前进行替换

如何减小 MTTR

- 故障检测 fault detection
- 故障诊断 fault diganosis
- 故障修复 fault repair

例题6 磁盘的 MTTF 和 AFR

当今的一些磁盘号称其 MTTF 为 1000000 小时，大约是 114 年，也就意味着这些磁盘永远不会失效，运行搜索引擎等网络服务的仓储式计算机大约有 50000 台服务器，假定每台服务器有两块磁盘，使用 AFR 计算每年将会有多少块磁盘失效

一年有 $365 \times 24 = 8760(h)$，1000000 小时的 MTTF 意味着

$$AFR = \frac{8760}{1000000} = 0.876\%$$

由于系统中有 100000 块磁盘，也就意味着 $100000 \times 0.876\% = 876$ 块磁盘失效，即平均两条就有超过两块磁盘失效

Hamming SEC Code

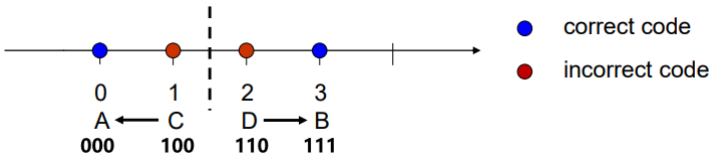
通常，我们对 64 bit 的数据加上 8 bit 的校验位（共 72 bit），用于检查

Hamming 距离

- 两个 bit pattern 之间不同的 bit 数
- 例如：如果使用 111 来表达 1，000 来表达 0，此时 hamming 距离 d = 3
- 例如：奇偶校验中奇数校验 10 → 101，11 → 110，此时 hamming 距离 d = 2

如果检测两个距离不满足 hamming 距离，那么必然是出错了

hamming 距离 = 3 可以提供 1 bit 的纠错，2 bit 的检测



如上图，如果 hamming 距离 d = 3

- 编码 = 100，此时发生错误，d = 1，我们有理由认为 100 距离 000 更近，可以把它修改成 000

如果 hamming 距离为 n，那么我么能够修改的错误为 $\frac{n-1}{2}$ bit，我们能够检查出来的错误为 $n-1$ bit

Hamming Code 编码

如下表所示

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
		0	1	1	1	0	0	1	0	1	0	1	0
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

- 对数据部分从左到右由 1 开始依次编号（不是从 0 开始）
- 将所有编号为 2 的整数次幂的 bit 标记为奇偶校验位（1，2，4，8，16...）
- 其它剩余 bit 位数据位（3，5，6，7，9，10，11，12，13，14，15）
- 奇偶校验位的位置决定了其对应的数据位
- 进行偶校验

为了填写校验位的值，我们观察

- p1 中 × 选取地址的二进制中最低 bit 为 1 的部分
  对应的值为 1、0、1、1、1，此时一共有偶数个 1，故校验位填 0
- p2 中 × 选取地址的二进制中倒数第二 bit 为 1 的部分
  对应的值为 1、0、1、0、1，此时一共有奇数个 1，故校验位填 1
- p4 中 × 选取地址的二进制中倒数第三 bit 为 1 的部分
  对应的值为0、0、1、0，此时一共有奇数个 1，故校验位填 1
- p8 中 × 选取地址的二进制中倒数第四 bit 为 1 的部分
  对应的值为1、0、1、0，此时一共有偶数个 1，故校验位填 0

	1	2	3	4	5	6	7	8	9	10	11	12
2进制	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
校验位	0	1		1				0				
值			1		0	0	1		1	0	1	0
p1	×		×		×		×		×		×	
p2		×	×			×	×			×	×	
p4				×	×	×	×					×
p8								×	×	×	×	×

Hamming Code 解码

我们把用于校验的 p1、p2、p4、p8 行逐行的 × 的部分与校验的部分加起来，如果加和为 0，说明没有错误，如果加和为 1，说明存在错误

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	
		0	1	1	1	0	0	1	0	1	1	1	0	
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	
Parity bit coverate	p1	X		X		X		X		X		X		√ 0
	p2		X	X			X	X			X	X		X 1
	p4				X	X	X	X					X	√ 0
	p8								X	X	X	X	X	X 1

如图，假设位置 10 上发生了错误

我们加和校验码的时候可以发现 p2 和 p8 发生了错误，按照 p8-p4-p2-p1 的顺序写出 $1010_2 = 10$，可以找到位置 10 上发生了错误，可以纠正该 bit 的错误

检测 2 bit 错误

如果我们要检测 2 bit 的错误呢？

我们在最后添加一个 p_n bit 用于整体的偶校验

让 H = [p8,p4,p2,p1]

- 如果 H 为偶，p_n 为偶 → 没有错误
- 如果 H 为奇，p_n 为奇 → 1 bit 错误
- 如果 H 为偶，p_n 为奇 → p_n 错误
- 如果 H 为奇，p_n 为偶 → 2 bit 错误