Chapter 5

大容量、高性能: 利用内存层次结构

Outline

- **内存概述**
- **缓存基础**
- **缓存的性能**
- 可靠的内存技术
- 虚拟内存
- 一个通用框架

测量缓存的性能

- CPU时间的组成
  - 程序执行所需的时钟
    - 包括缓存命中的时间: #insts * CPI (* 时钟周期时间)
  - 内存阻塞所需的时钟（此时CPU空转）
    - 主要源自缓存失效
- 简化的假设:

$$\text{程序的内存阻塞周期数}$$

$$= \text{程序的内存访问次数} \times \text{失效率} \times \text{失效惩罚}$$

$$= \text{程序的指令数目} \times \frac{\text{失效数}}{\text{指令数目}} \times \text{失效惩罚}$$

缓存性能的例子

- 假设
  - I-cache 失效率为 2%
  - D-cache 失效率为 4%
  - 失效惩罚 = 100 cycles
  - 基础CPI (理想缓存情况) = 2
  - 36%的指令为Load & stores
- 计算：每条指令失效所需的周期数
  - I-cache: 0.02 × 100 = 2
  - D-cache: 0.36 × (0.04 × 100) = 1.44
- 计算：实际的CPI = 2 + 2 + 1.44 = 5.44
  - 比理想的CPU 慢：5.44/2 =2.72倍

平均的内存访问时间

- 命中时间对性能也很重要
- 平均内存访问时间(AMAT)
  - AMAT = 命中时间 + 失效率 x 失效惩罚
- 例子
  - 假设CPU时钟为1ns，命中时间为1个时钟，缓存失效率为5%CPU，失效惩罚为20个时钟
  - AMAT = 1 + 0.05 × 20 = 2ns
    - CPI = 2
  - 为什么不是 ...
    - AMAT = 0.95 x 1 + 0.05 x 20 = 1.95ns?

性能小结

- 当CPU性能提升时
  - 失效惩罚 * 失效率带来的性能下降，会更突出
- 因此，在评估系统性能时，不能忽视缓存
  - 降低失效惩罚
  - 降低失效率
- 能否增加缓存块的大小?
- 还有别的方法吗?

相联缓存

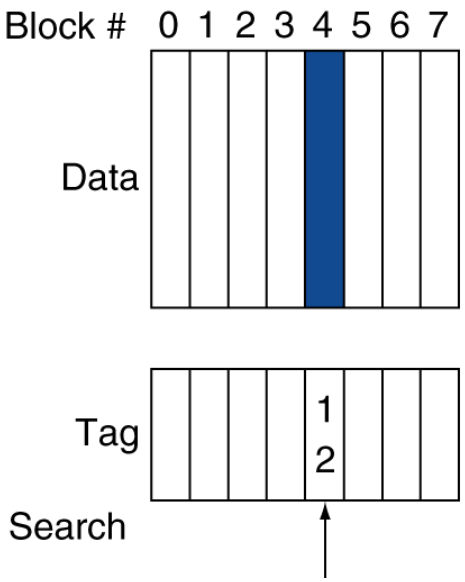
- 直接映射：内存块号直接映射到唯一缓存块
- 全相联
  - 允许内存块映射到任意缓存块
  - 查找时，需要同时搜索所有缓存块
  - 所有缓存块都需要配备一个比较器（很昂贵）
- k路-组相联
  - 允许内存块映射到一个组内的任意缓存块（每组k个）
  - 内存块直接映射到唯一缓存组（组中的任意块）
    - (内存块号) modulo (缓存中的组数)
  - 查找时，需要同时搜索一个组内的所有缓存块
  - *k*个比较器 (没那么昂贵)

组相联缓存的例子

- 在不同的映射下，内存块存放在不同的缓存块
  - 内存地址的块号: 12

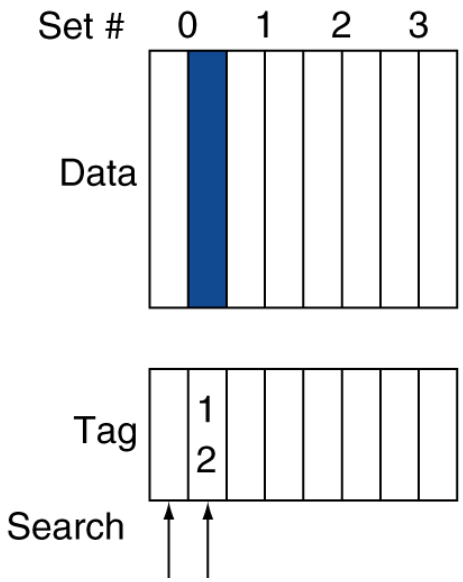
12%8=4

Direct mapped



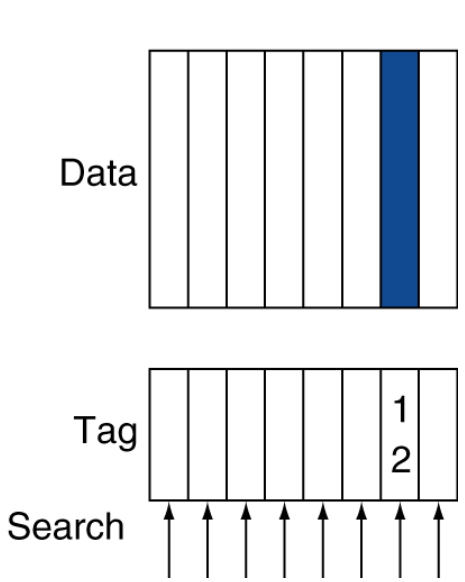
12%4=0

Set associative



12%1=any

Fully associative



相联度的不同选择

- 计算缓存有8个缓存块

One-way set associative

(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

相联度的例子

- 假设缓存有4个块
  - 比较：直接映射、2路组相联，全相联
  - 内存块的访问序列为：0, 8, 0, 6, 8

- 直接映射：(0%4, 8%4, 0%4, 6%4, 8%4)

内存块号	缓存块号	命中/失效	缓存的内容			
			缓存块0	缓存块1	缓存块2	缓存块3
0	0	失效, why?	Mem[0]			
8	0	失效	Mem[8]			
0	0	失效, why?	Mem[0]			
6	2	失效	Mem[0]		Mem[6]	
8	0	失效, why?	Mem[8]		Mem[6]	

相联度的例子

- 2路组相联 (0%2, 8%2, 0%2, 6%2, 8%2)

内存块号	缓存块号	命中/失效	缓存的内容			
内存块号	缓存块号	命中/失效	缓存组 0		缓存组 1	
0	0	失效	**Mem[0]**			
8	0	失效	Mem[0]	**Mem[8]**		
0	0	命中, why?	**Mem[0]**	Mem[8]		
6	0	失效	Mem[0]	**Mem[6]**		
8	0	失效, why?	**Mem[8]**	Mem[6]		

- 全相联

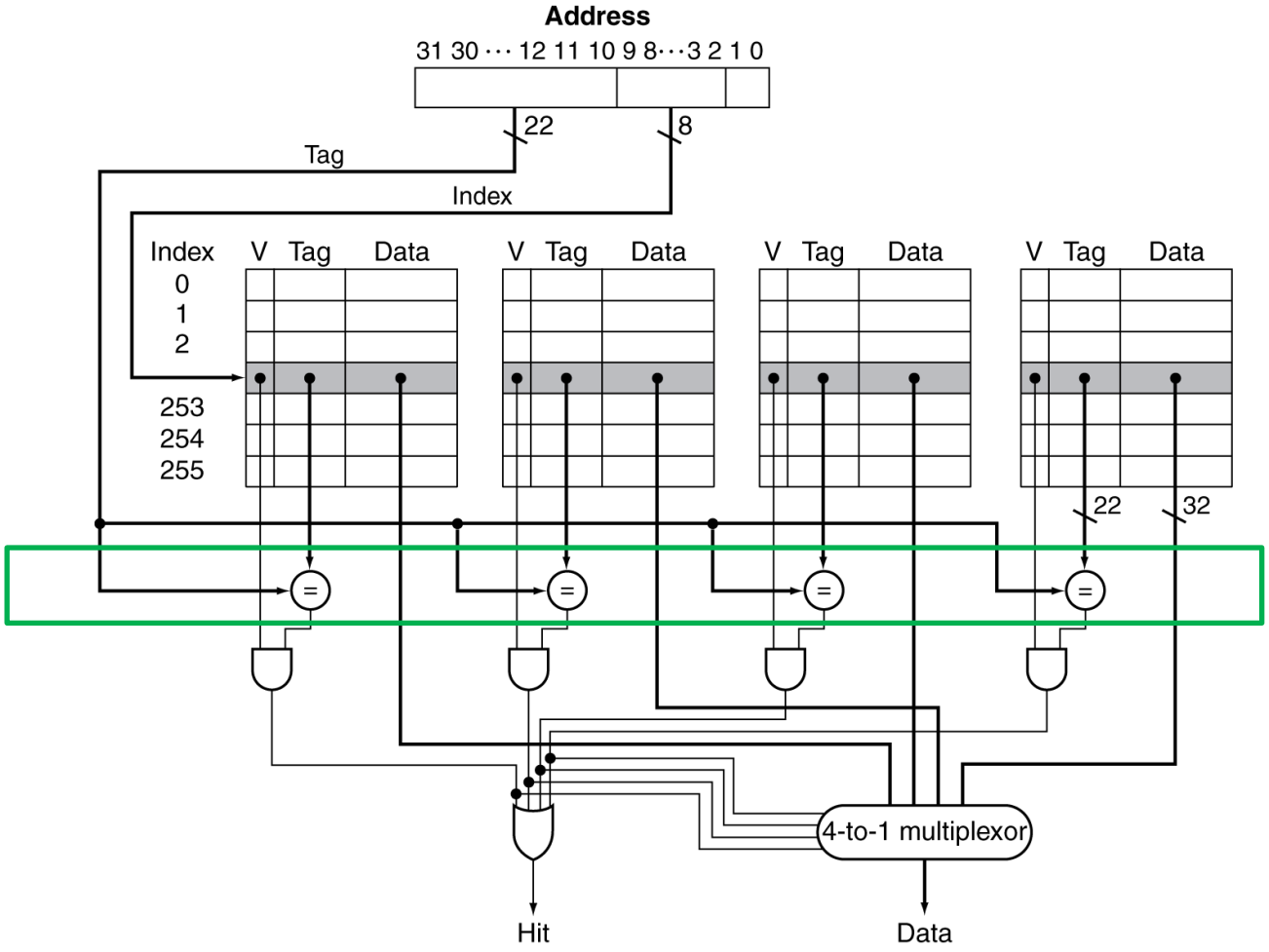
内存块号		命中/失效	缓存的内容			
0		失效	**Mem[0]**			
8		失效	Mem[0]	**Mem[8]**		
0		命中	**Mem[0]**	Mem[8]		
6		失效	Mem[0]	Mem[8]	**Mem[6]**	
8		命中, why?	Mem[0]	**Mem[8]**	Mem[6]	

相联度的影响

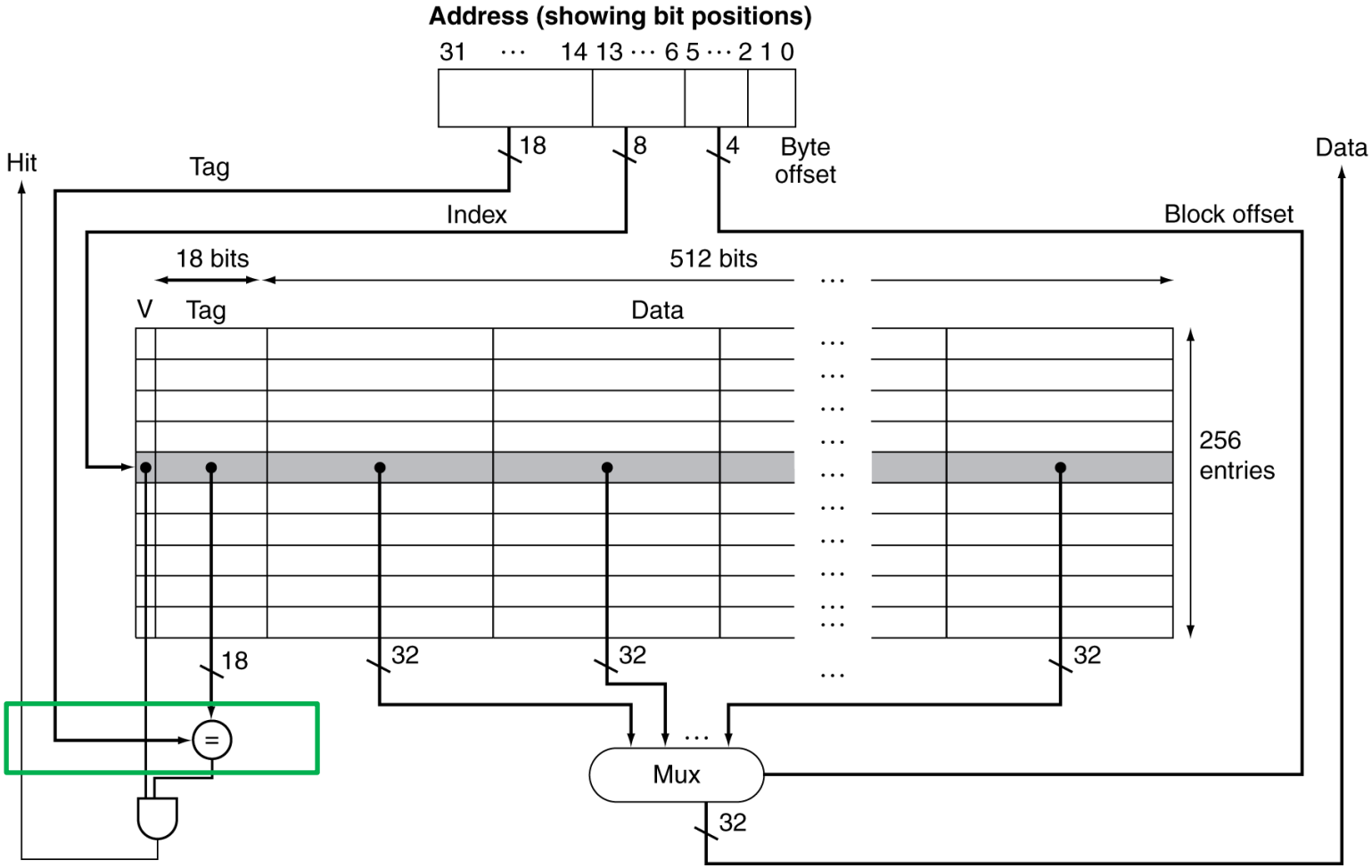
- 相联度越高，失效率越低
  - 但是效益递减
  - 而且，需要更多的比较器
- 模拟实验：64KB的数据缓存，每块有16个字，应用程序SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

组相联缓存的构成

- k 个比较器



回顾: 直接映射缓存的构成



替换策略

- 直接映射: 唯一选择
- 组相联
  - 优先空闲缓存块（有效位为0的块）
  - 其次，在组内挑选一个缓存块
- 策略1: 近期最少使用 (LRU)
  - 选择最长时间内未被使用的缓存块
    - 实现难度：2路简单，4路可控，超过4路很难
- 策略2：随机
  - 对于高度相联缓存，其性能接近LRU

多级缓存

- 附属在CPU的主缓存（L1缓存）
  - 容量小，速度块
- L2缓存
  - L1缓存失效时，从L2缓存供应
  - 更快，更慢，但是仍然比主存快
- 主存
  - L2缓存失效时，从主存供应
  - 有些高端系统，在主存之上还有L3缓存

多级缓存的例子

- 假设
  - CPU基础CPI为1, 时钟频率为4GHz
  - 每条指令失效率为 2%
  - 主存访问时间为100ns
- 如果只有主缓存
  - 1 时钟时间: 1/时钟频率 = 1s/4G = 0.25 ns
  - 失效惩罚 = 100ns/0.25ns = 400 cycles
  - 实际CPI = 1 + 0.02 × 400 = 9
- 9x slower?

例子 (续.)

- 现在增加L2缓存，假设
  - 访问时间为 5ns
  - 失效率为 0.5%
- L1缓存的失效惩罚（L2缓存命中时间）：
  - 5ns/0.25ns = 20 cycles
- L2缓存的失效惩罚（主存访问时间）
  - 400 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- 加速比：9/3.4 = 2.6

多级缓存的影响

- L-1缓存
  - 关注最小的命中时间（~1个周期）
- L-2缓存
  - 关注更低的失效率，以避免主存访问
  - 命中时间的影响，没有那么大
- 因此，
  - L-1缓存的容量通常比较小
  - L-1缓存的块大小，通常比L2缓存的块大小要小

扩展: 真实系统中的缓存系统

- Windows
  - taskmgr
- MacOS
  - about this Mac
  - sysctl -a|grep hw|grep cache
- Linux
  - cat /proc/cpuinfo, meminfo
  - lscpu
- Try a linux system to learn better

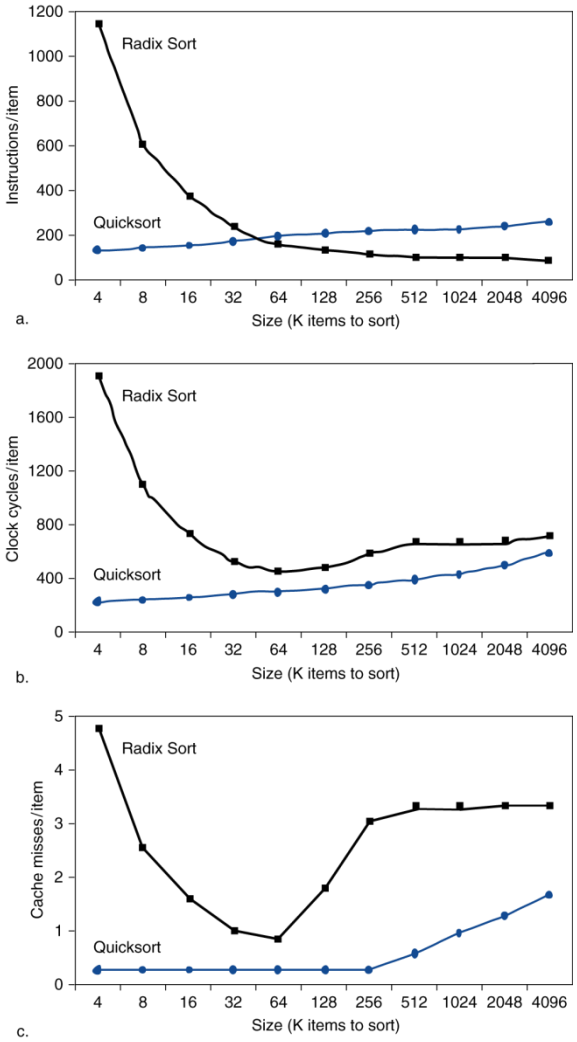
```
qali@Rosen-MacBook-Pro-old Projects % sysctl -a|grep hw|grep cache

hw.cacheconfig: 8 2 2 8 0 0 0 0 0 0
hw.cachesize: 17179869184 32768 262144 6291456 0 0 0 0 0 0
hw.cachelinesize: 64
hw.l1icachesize: 32768
hw.l1dcachesize: 32768
hw.l2cachesize: 262144
hw.l3cachesize: 6291456
```

缓存对高级CPU中的影响

- 乱序执行的CPU在发生缓存失效时，可以执行别的指令
  - 被阻塞的load/store指令等待漫长的内存访问
  - 依赖load/store指令的那些指令在保留站等待
    - 无关的其他指令可以执行
- 失效时的行为依赖于程序的数据流
    - 很难静态分析
    - 通常用系统仿真

缓存对软件的影响

- 奇怪的结果!
- 内存访问模式引起失效
  - 算法
  - 编译优化
    - 让代码利用缓存局部性



局部性的量化估计-局部性的例子1

- 对于一个专业程序员来说，阅读代码获得其局部性的感觉，是一个重要的技能
- 这个函数中的数组a有很好的局部吗？
  - 假设块大小为32字节，命中率是多少？

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

局部性的量化估计-局部性的例子1

- 对于一个专业程序员来说，阅读代码获得其局部性的感觉，是一个重要的技能
- 这个函数中的数组a有很好的局部吗？
  - 假设块大小为32字节，命中率是多少？
  - **hit ratio: 7/8**

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

局部性的例子2

- 这个函数中的数组a有很好的局部吗?
  - 假设块大小为32字节，命中率是多少?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

局部性的例子2

- 这个函数中的数组a有很好的局部吗?
  - 假设块大小为32字节，命中率是多少?
    - **hit ratio: 1/8**

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

局部性的例子3

- 能否重排循环，让该函数以步长为1的内存访问模式来扫描3d数组a?（这样，代码就会有很好的空间局部性）

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
       for (j = 0; j < N; j++)
          for (k = 0; k < N; k++)
             sum += a[k][i][j];
    return sum;
}
```

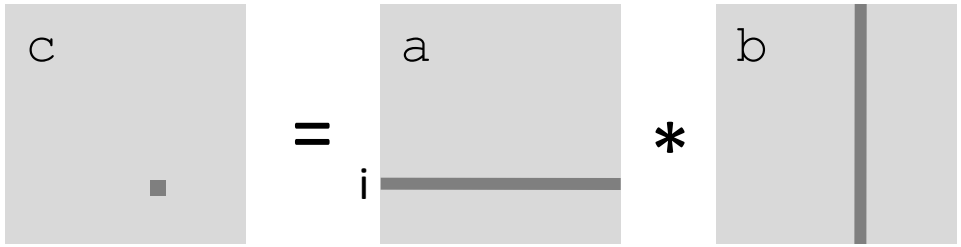
局部性的例子3

- 能否重排循环，让该函数以步长为1的内存访问模式来扫描3d数组a?（这样，代码就会有很好的空间局部性）

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (k = 0; k < N; k++)
       for (i = 0; i < M; i++)
          for (j = 0; j < N; j++)
             sum += a[k][i][j];
    return sum;
}
```

例子4: 矩阵乘法（扩展）

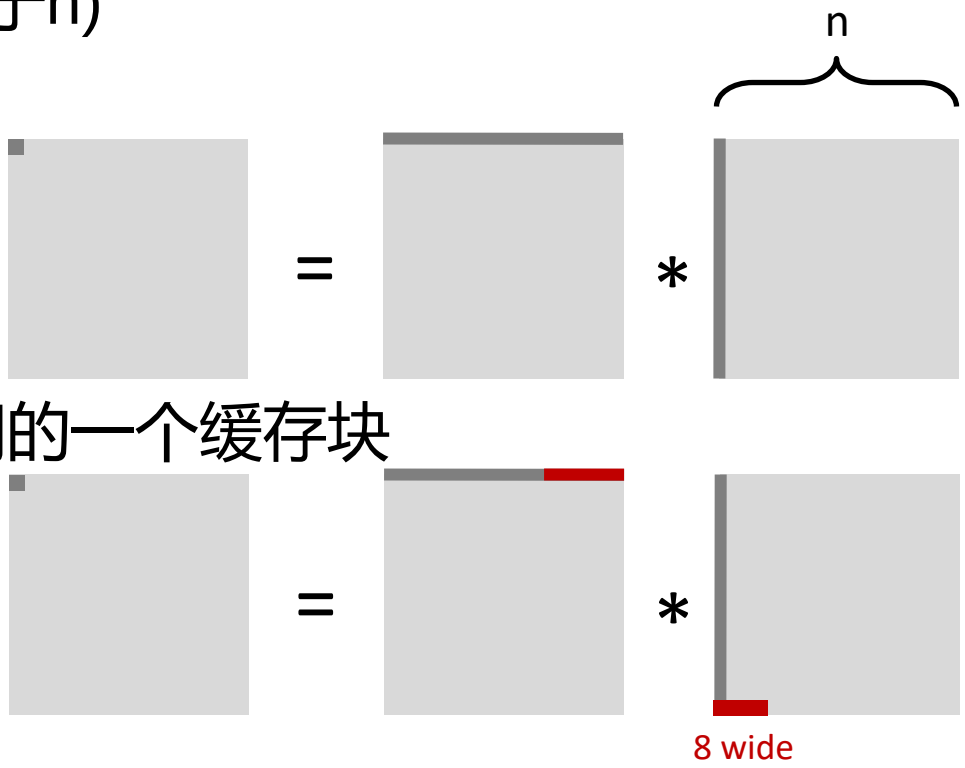
```
c = (double *) calloc(sizeof(double), n*n);

/* 矩阵a 乘 矩阵b，形状为n x n  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
}
```



缓存失效分析

- 假设：(仅考虑内存循环, A&B)
  - 矩阵中的元素都是double类型
  - 缓存块可以存放 8个double类型
  - 缓存容量C << n (远小于n)

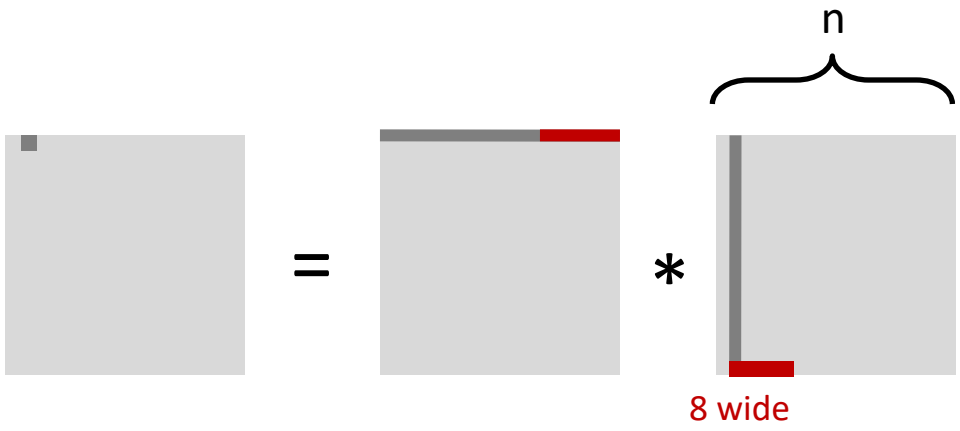
- 第一次迭代:
  - 失效：n/8 + n = 9n/8
  - 红色表示最后一次访问的一个缓存块
    - 访问A的一行：n/8
    - 访问B的一列：n



缓存失效分析

- 假设: (仅考虑内存循环, A&B)
  - 矩阵中的元素都是double类型
  - 缓存块可以存放 8个double类型
  - 缓存容量C << n (远小于n)

- 第二次迭代:
  - 同样，失效为:
    n/8 + n = 9n/8

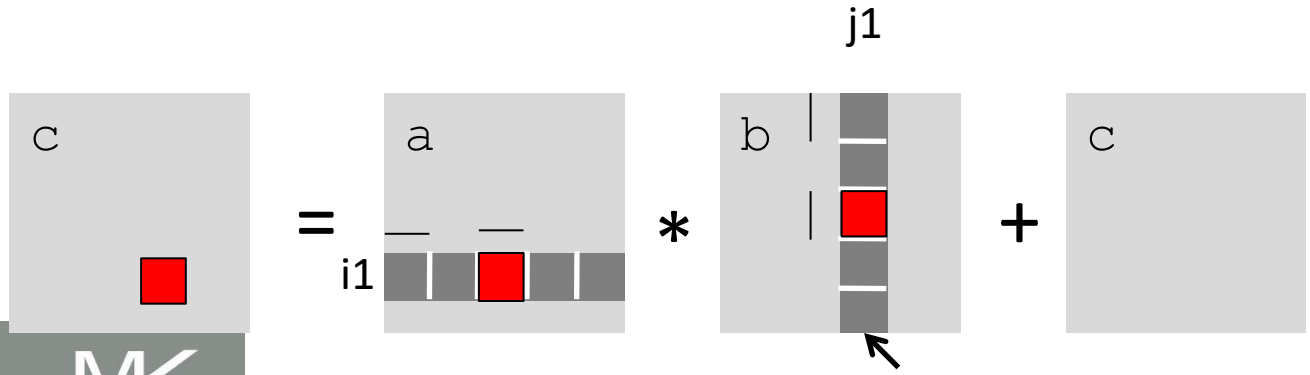


- 总的失效为:
  - 9n/8 * n^2 = (9/8) * n^3 (而不是 2*n^3, 为什么?)

分块的矩阵乘法

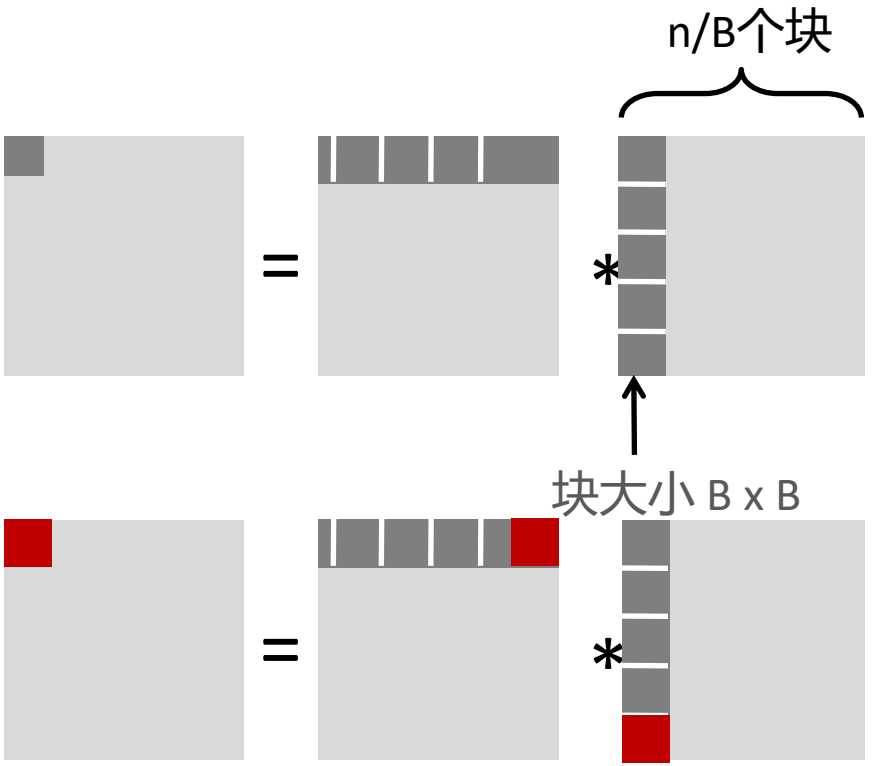
```
c = (double *) calloc(sizeof(double), n*n);

/* 矩阵a 乘 矩阵b，形状为n x n  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* 每次计算形状为B x B 的小矩阵的乘法 */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1][j1] += a[i1][k1]*b[k1][j1];
}
```

matmult/bmm.c



缓存失效分析

- 假设: (仅考虑内存循环, A&B)
  - 缓存块可以存放 8个double类型
  - 缓存容量C << n (远小于n)
  - 但是 **$3B^2 < C$，即缓存可以存下3个小矩阵**
- 第一次迭代(块级别) :
  - 内层1循环小矩阵a的1行和b的1列
    - 失效：B/8 + B/8 = B/4
  - 内层3循环累计B^2次
    - $B/4 * B^2 = B^3/4$
  - 内层4循环累计n/B次
    - $B^3/4 * n/B = nB^2/4$
  - 总的失效
    - $nB^2/4 * (n/B)^2 = n^3/4$



缓存分块的小结

- 不分块: (9/8) * n^3
- 分块: (1/4) * n^3
  - B 为缓存块大小的倍数，才有效
  - 要满足$3B^2 < C$!

- 戏剧性的性能差距的原因:
  - 矩阵乘法固有的数据局部性:
    - 输入数据的规模: $3n^2$, 计算规模 $2n^3$
    - 每个数组元素访问 O(n) 次!
  - 但是需要合理地编写程序

能否通过重排循环进行优化?

- 重排多层循环?

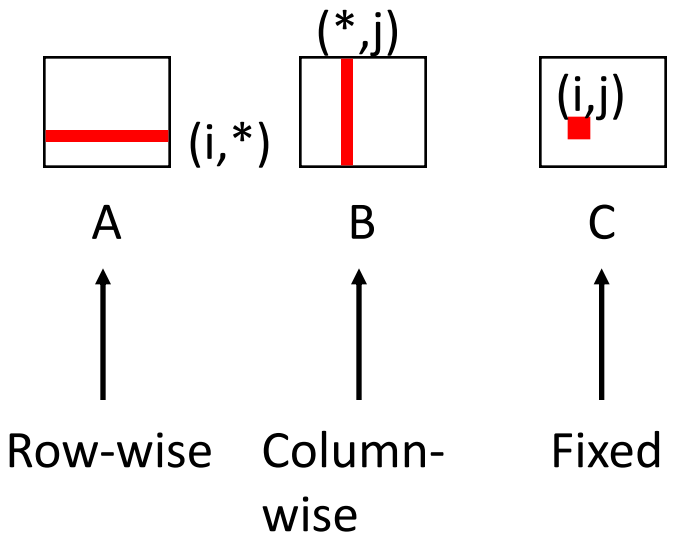
矩阵乘法 (ijk)

- 重排多级循环?

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

内层循环:



内层循环每次迭代的失效率:

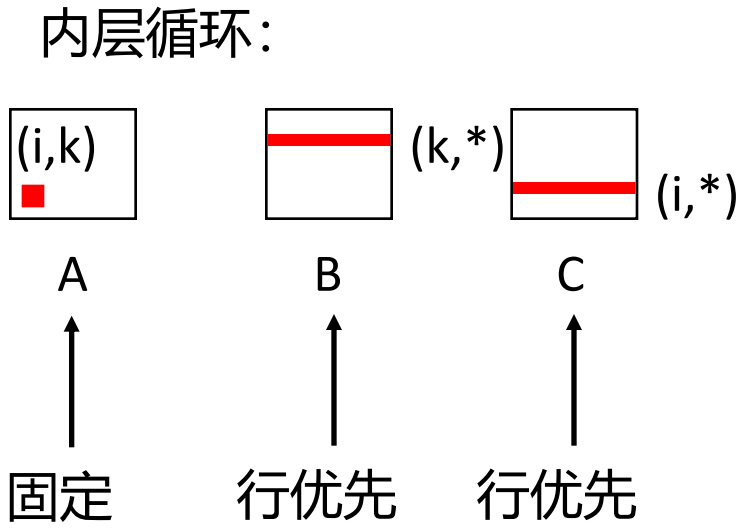
A	B	C
0.125	1.0	0.0

#失效: $(n/8+n) * n^2 = 9/8 * n^3$

循环重排后的矩阵乘法

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c



内层循环每次迭代的失效率:

A	B	C
0.0	0.125	0.125

$(n/8+n/8) * n^2=1/4 * n^3$

不如cache blocking：$1/(4B) * n^3$

还没有更糟的reordering？

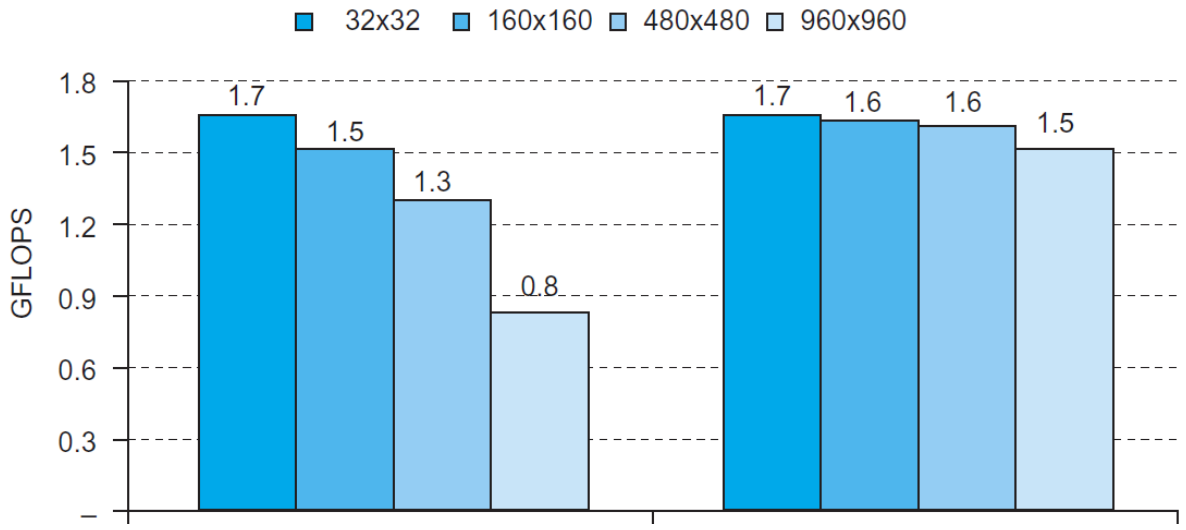
软件优化的小结

- 缓存对性能有显著的影响
- 程序员可以编写缓存优化的程序，来提升性能
  - 专注内层循环：那是计算和访存操作集中的位置
  - 最大化空间局部性：以步长为1顺序访问数据
    - 循环重排
  - 最大化时间局部性：在数据加载到缓存后，尽可能频繁地访问
    - cache blocking (also called loop tiling) 缓存分块

缓存分块的矩阵乘法

```
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5  for (int i = si; i < si+BLOCKSIZE; ++i)
6   for (int j = sj; j < sj+BLOCKSIZE; ++j)
7   {
8    double cij = C[i+j*n];/* cij = C[i][j] */
9    for( int k = sk; k < sk+BLOCKSIZE; k++ )
10    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11   C[i+j*n] = cij;/* C[i][j] = cij */
12  }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17   for ( int si = 0; si < n; si += BLOCKSIZE )
18    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19     do_block(n, si, sj, sk, A, B, C);
20 }
```

分块矩阵乘法的访问模式



Unoptimized Blocked

应用：Example from paddlepaddle

- https://github.com/PaddlePaddle/CINN

```
#include "cinn/cinn.h"
using namespace cinn;
// Declare constants
Expr M(10), N(20), K(30);
// Declare the inputs
auto A = Placeholder<float>("A", {M, K});
auto B = Placeholder<float>("B", {K, N});
auto k1 = Var(K.as_int32(), "k1");
auto C = Compute(
{M, N}, [&](Var i, Var j) { return ReduceSum(A(i, k1) * B(k1, j), {k1}); }, "C");
Target target = common::DefaultHostTarget();
int block_size = 32;
// The stages holds all the schedules for each tensors.
auto stages = CreateStages({C});
// Blocking optimization by loop tiling stragety.
auto [i_outer, i_inner, j_outer, j_inner] = stages[C]->Tile(0, 1, bn, bn);
auto [k_outer, k_inner] = stages[C]->Split("k0", 4);
stages[C]->Reorder({i_outer, j_outer, k_outer, k_inner, i_inner, j_inner});
// Generate C source code:
Module::Builder builder("module_block", target);
auto func = Lower("matmul_block", stages, {A, B, C});
builder.AddFunction(func);
CodeGenCX86 compiler(target, CodeGenCX86::Feature::AVX512);
Outputs outputs;
outputs = outputs.c_header("./test02_matmul_block.h").c_source("./test02_matmul_block.cc");
compiler.Compile(builder.Build(), outputs);
```

应用：华为列出的难题 and 曙光DCU竞赛

HUAWEI

.com

技术挑战

Matmul作为一种常用算子，其内核实现和分块算法在任何形状/数据类型/格式上都需要具备良好的执行性能。此外，为防止设备执行时出现阻塞，分块算法需在极短时间内执行完成。

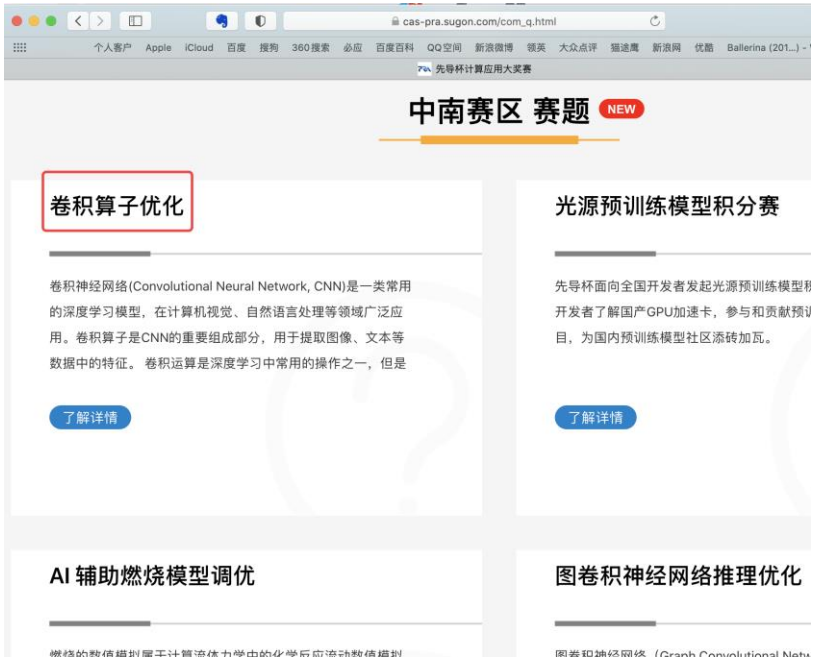
当前结果

当前内核实现和分块算法的性能如下：

- 分块算法的执行时间可以限制在5 μs左右；
- 针对某些形状，执行性能有明显的下降；
- 昇腾910B内核的平均cube利用率约70%，英伟达A100 GPU平均利用率约85%。

技术诉求

- 分块算法的执行时间应小于5 μs；
- 内核执行性能随形状的变化呈线性（或近似线性）变化；
- 内核的平均cube利用率达90%以上；
- 支持多种数据类型（如Int8/FP16/FP32等）。



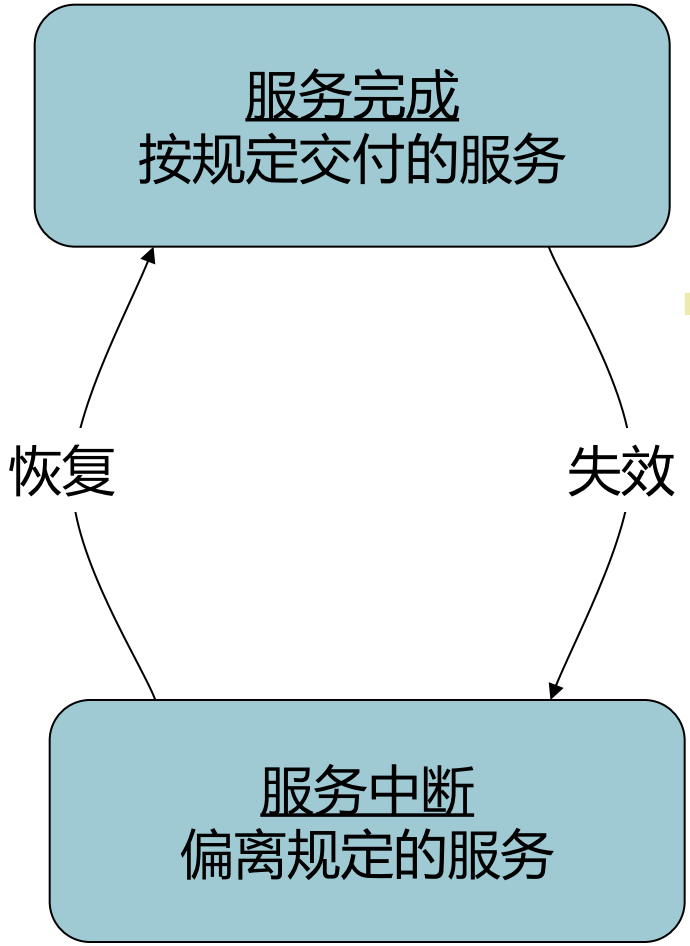
总结：缓存的性能

- 缓存性能很重要
- 性能优化方法
  - 硬件
    - 利用相联度降低失效率
    - 利用多级缓存降低失效惩罚
    - 通过**小容量L1**缓存减少命中时间
    - **缓存块大小**的考虑：失效率↓，失效惩罚↑
  - 软件优化
    - 分块
    - 重排
    - 分块 & 重排?
  - prefetch?

Outline

- **内存概述**
- **缓存基础**
- **缓存的性能**
- **可靠的内存技术**
- 虚拟内存
- 一个通用框架

可靠性



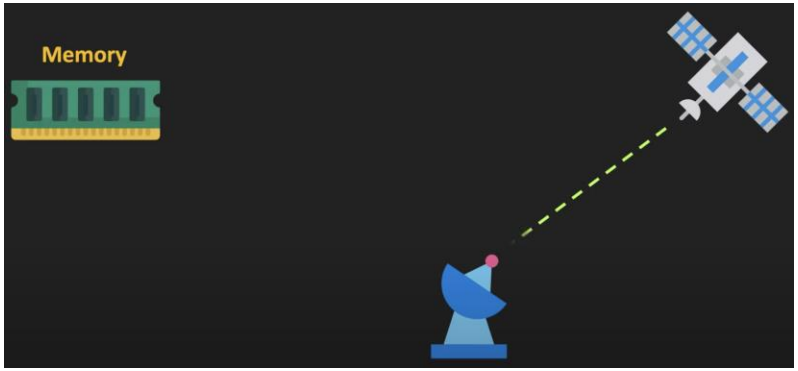
- 故障: 组件的失效
  - 可能会也可能不会，导致整个系统的失效

可靠性的度量

- 正常运行时间——平均故障时间 (MTTF)
- 服务中断时间——平均修复时间 (MTTR)
- 平均的故障间隔时间
  - MTBF = MTTF + MTTR
- 可用性 = MTTF / (MTTF + MTTR)
  - 可用性=正常运行时间 / (正常运行时间+服务中断时间)
- 提升可用性
  - 提高 MTTF: 故障避免、故障容忍、故障预测
  - 降低 MTTR: 先进的诊断&修复的工具与流程

时间/空间带来可靠性问题

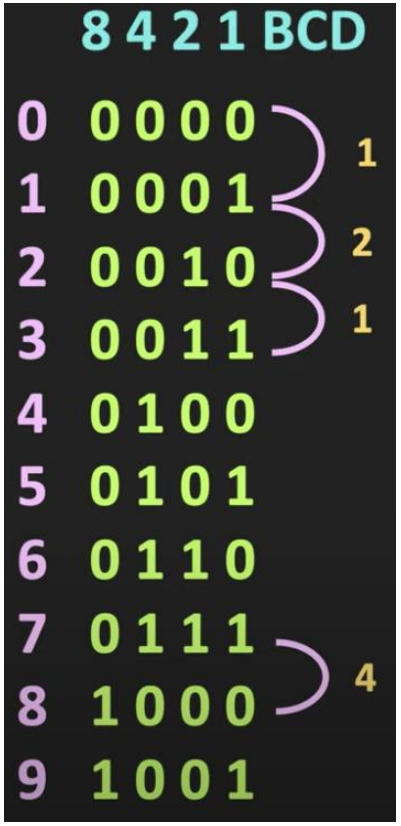
- 存储中的可靠性（时间）
  - 在时间上传递
- 通信中的可靠性（空间）
  - 在空间上传递



- 利用冗余改善可靠性
  - 3倍重复编码：
    - 将要传递的信息重复3次，根据多数投票确定结果
      - 假设：错2位的概率 << 错1位的概率

海明距离，最小海明距离

- 定义
  - 两个0/1序列中，对应位置上bit值不同的数目



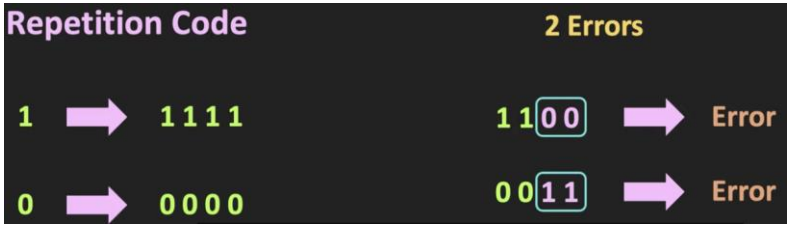
  - 3倍重复编码的海明距离是多少？

最小海明距离与检错、纠错能力

- 3倍重复编码的检错能力
  - 最小海明距离为3
  - 可以纠正1位错误

  - 可以检测2位错误

- 如何提高能力，比如纠正2位错误?
  - 4倍重复编码?
  - 5倍重复编码?

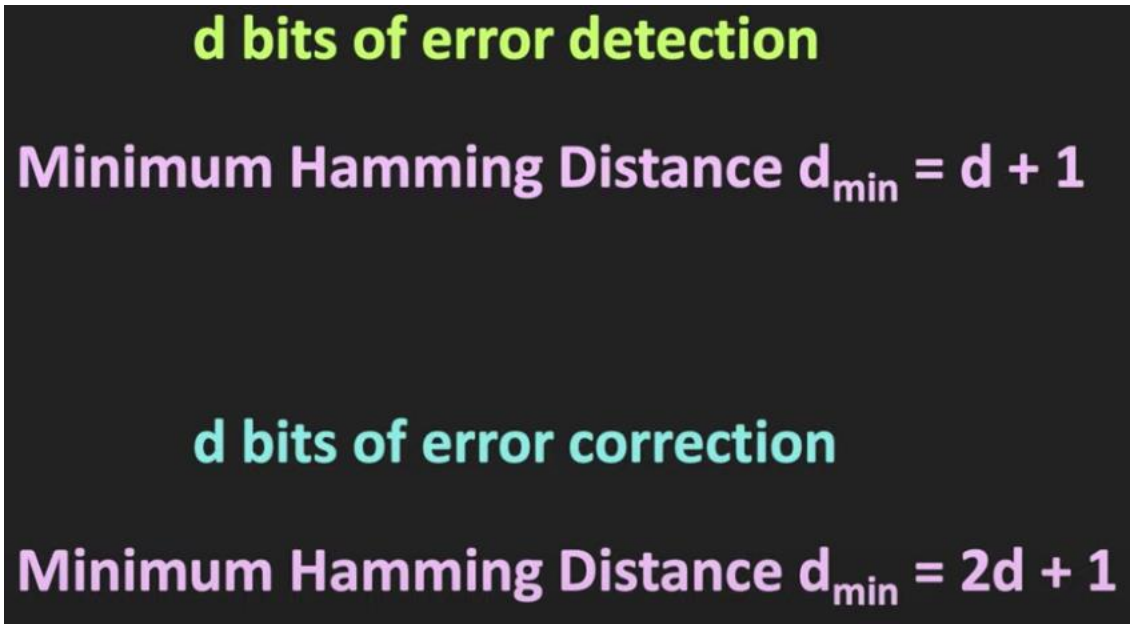


最小海明距离与检错、纠错能力

- 最小海明距离 = 2
  - 提供单bit错误的检错能力 provides single bit
  - 比如，奇偶校验码 parity code
- 最小海明距离 = 3
  - 提供单bit错误的纠错能力
  - 提供2-bit错误的检错能力

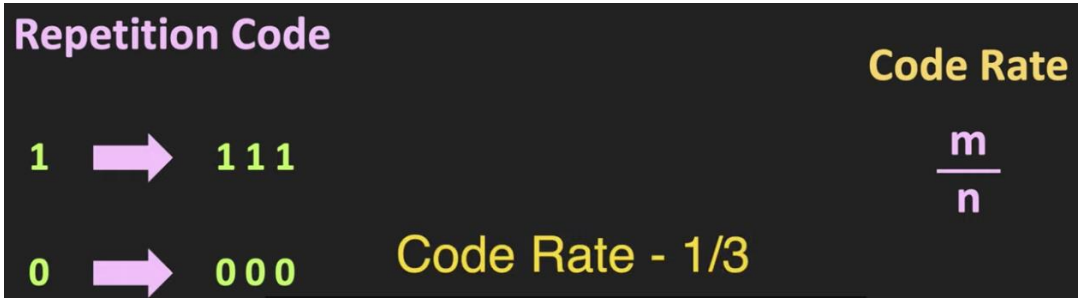
最小海明距离与检错、纠错能力

- 最小海明距离为d
  - 可以实现d-1位检错
  - 可以实现(d-1)/2位纠错



但是，编码效率很重要

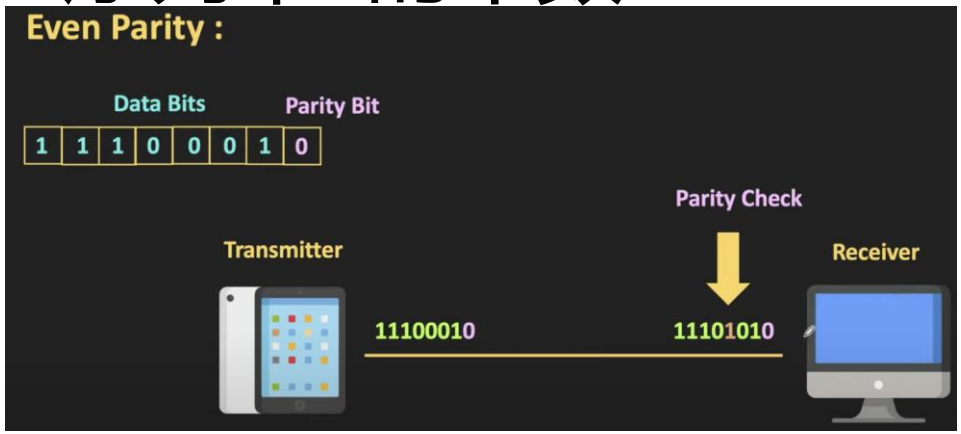
- 编码效率的定义
  - 有效信息bits/整体信息bits



- 编码效率太低
  - 降低了通信带宽
  - 浪费了存储资源

奇偶校验编码—海明距离为2？

- 利用一位bit，计算bit序列中1的个数
  - 奇数 -> 校验位为1
  - 偶数 -> 校验位为0
  - 实现：异或逻辑
- 过程
  - 写时生成校验位
  - 读时检查校验位
- 能力
  - 检测奇数个错误
  - 不能纠正



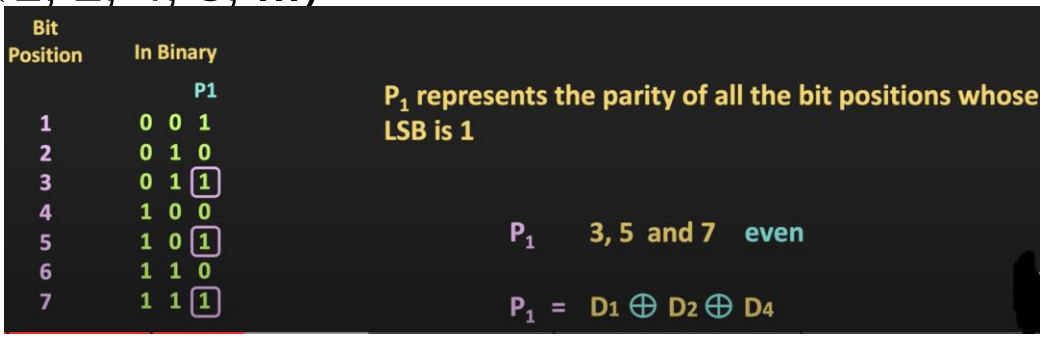
SEC(Single Error Correction)编码——(7,4)

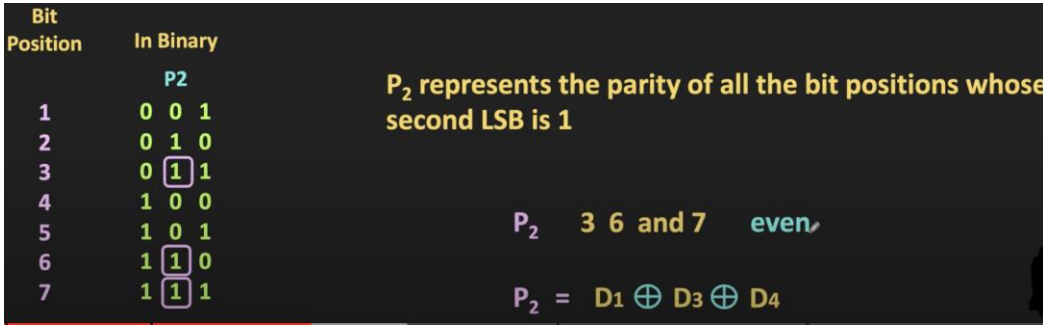
- 编码方案
  - p表示校验位
  - d表示数据位
  - p_k是位置编号第k位为1的bit值的偶校验
    - 校验位的位置为2^k（1, 2, 4, 8, ...）

	p1	p2	d1	p3	d2	d3	d4
位置编号	1	2	3	4	5	6	7

- p_3怎么编码？

$$d_2 \oplus d_3 \oplus d_4$$





SEC的编码

- 为了计算海明码:
  - 左边起，从1开始为bit位编号
  - 恰好处于2的幂的位置，用来存储奇偶**校验位**
    - 如下图中的**p1**、**p2**、**p4**、**p8**
  - 每个校验位只检查特定的数据位
    - 比如，p4 (100)只检查位置（1xx)

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
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

SEC的解码

- Value of parity bits indicates which bits are in error
- 所有校验位组成的值，指示了出错的bit位
  - 使用编码过程中的排序
  - 比如
    - 校验位为 0000，表示没有错误
    - 校验位为1010，表示第10th 位的bit被翻转了
      - 1010_2 等于 10_{10}

练习

- 给定数据1010，如何进行（7，4）编码？

练习

- 给定数据1010，如何进行（7，4）编码？
  - p1 = (3,5,7) = d1◎d2◎d4 = 1
  - p2 = (3, 6, 7) = d1◎ d3◎ d4 = 0
  - p3 = (5, 6, 7) = d2◎ d3◎ d4 = 1

位置编号	1	2	3	4	5	6	7
	p1	p2	**d1**	**p3**	**d2**	**d3**	**d4**
位置存储值	1	0	1	1	0	1	0

- 如何检测和纠正错误？

练习

- 如何检测和纠正错误
  - 如果收到的信息中，d2变化了，如何检测和纠正？
    - p1校验：p1◎ d1◎d2◎d4 = 1
    - p2校验：p2◎ d1◎ d3◎ d4 = 0
    - p3校验： p3◎ d2◎ d3◎ d4 = 1
    - 错误的位置为5，即d2

位置编号	1	2	3	4	5	6	7
	p1	p2	d1	p3	d2	d3	d4
位置存储值	1	0	1	1	1	1	0

  - 如果没发生错误，校验值是多少？
  - 如果发生2个错误？
    - 比如d1和d2同时出错？

SEC编码的编码效率

- SEC编码(7, 4)
  - 编码效率:
    - 数据位的数目：d=4
    - 校验位的数目：p=3
    - 总的数目：n=p+d = 7
    - 编码效率：d/n = 4/7
- 分析
  - n位可能出错，且可能都不出错，总共有n+1个状态
  - p可以表示2^p个状态，故2^p>=p+d+1

SEC编码(7,4)

- 更多的SEC编码

	p1	p2	d1	p3	d2	d3	d4	p4	d5	d6
位置	1	2	3	4	5	6	7	8	9	10

(7,4) Hamming Code

(15,11) Hamming Code

(31,26) Hamming Code

$(2^k - 1 , 2^k - 1 - k)$

k – parity bits

$2^k - 1 = n$ (total bits)

$2^k - 1 - k = m$ (message bits)

- 编码效率?

扩展的海明编码：SEC/DEC Code

- 为整个序列添加额外的校验位
  - 称为 p_n
- 这样，海明距离 = 4
  - 可以纠错1个bit位，检错2个bit位
- 解码:
  - 假设 H = SEC 校验位
    - 如果H 偶数, p_n 偶数, 则没有错误
    - 如果H 奇数, p_n 偶数, 则发生了可纠正的、单bit错误
    - 如果H 偶数, p_n 奇数, 则在p_n 位发生了错误（可纠正)
    - 如果H 奇数, p_n 偶数, 则发生了2个bit的错误 (不可纠正)
- 注意：ECC DRAM 使用 SEC/DEC，用8位校验码来保护64位

参考资料

- More on ECC
  - https://www.youtube.com/watch?v=t4kiy4Dsx5Y
- More on coding theory
  - A book

算法题

- 1000瓶药，确定其中混入一瓶毒药
  - 允许用老鼠实验，服用一滴，3天后就知道效果
    - 毒药，死
    - 非毒药，正常
  - 3天后就必须找出毒药，只有一次实验周期
  - 问：至少需要多少老鼠?