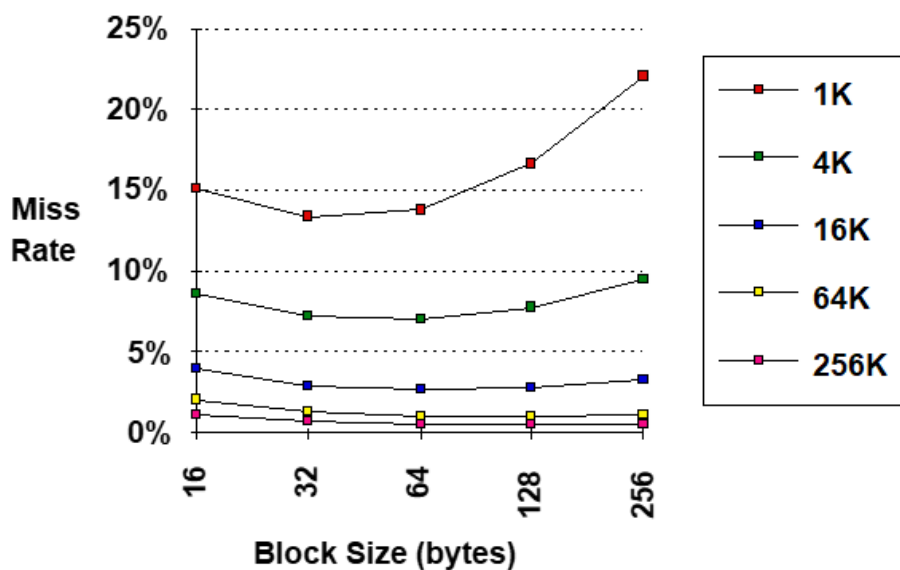


Computer Architecture W6D2

1. 降低失效率的方法

1.1 通过增大块的大小

1. Reduce Misses via Larger Block Size



1/24/01

CS252/Kubiatowicz
Lec 3.19

缓存总大小增加可以显著降低失效率，这是显然的。但需要注意的是在缓存总大小给定的情况下，失效率曲线却是先减后增的。原因大致如下：

1. 一方面，块的变大能增加空间局部性(space locality)，并且其也会减少强制失效
2. 另一方面，块的变大也导致了缓存中块数目减小，这有可能会增加冲突失效。值得注意的是，在缓存总大小较小时，这亦会导致更多的容量失效。

1.2 通过提高关联度

回顾我们之前讨论的 2:1 规则，这点变得很自然。但请注意：

- Beware: Execution time is only final measure!
 - Will Clock Cycle time increase?
 - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%

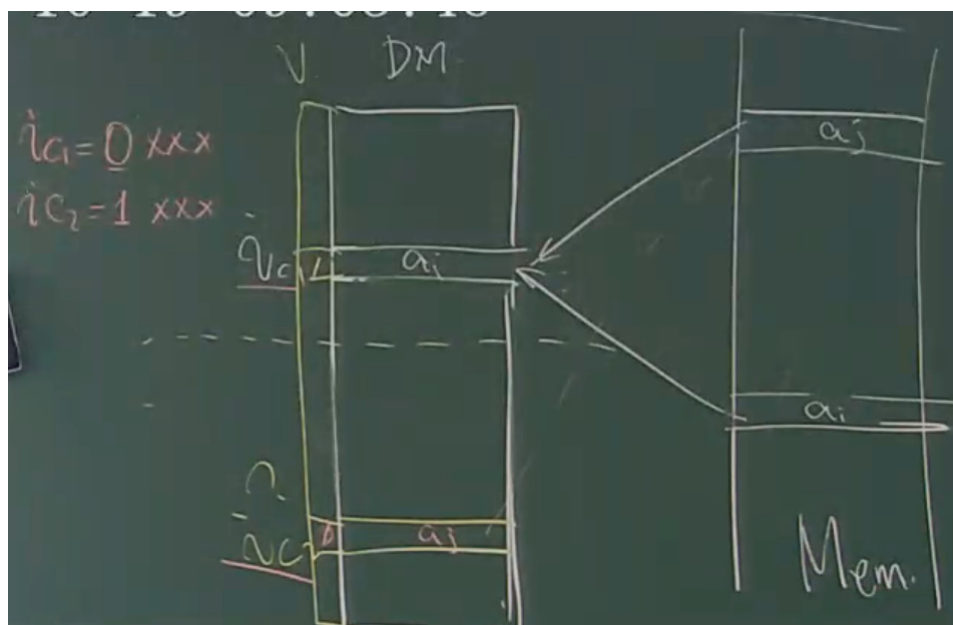
直观的想法：在关联度增加的时候 CCT 会增加，这导致有的时候尽管失效率降低了，我们的终极指标：AMAT 反而增加了。

1.3 利用“受害者缓存”

有时候，cache 中被替换的数据可能被立刻使用，又导致了更多的重新写入的时间开销（比方说开了个循环变量 a ，而 a 在循环中的两次使用之间间隔比较久）。那么我们就可以使用 victim cache。它是用来保存最近被替换的数据的，且一般采用全相连的结构。处理器内核可以同时读取 cache 和 victim cache。如果在 victim cache 中找到了，那么效率就和 cache 命中相同一样，接下来我们把 victim cache 中的数据写入 cache，相当于是进行了一次交换。

1.4 采用伪相联 (Pseudo-Associativity)

它结合了多路组相联的低不命中率和直接映像的命中速度。它通过索引最高位进行分割，首先按照直接映射的方式匹配表示，如果不成功匹配，则将最高位取反再次进行匹配。



这张图应该能非常好地解释我上面所说的内容。

1.5 硬件预取/软件预取

硬件预取可以预先将指令和数据读入快速存储器，比方说如果 miss 了，就把后面两个数据块都读进来，但是前提是不能影响程序的正常工作，即预取数据的时候处理器还是能继续进行或者硬件还能占有同样的带宽，否则反而可能是无用功。

软件预取是在说，直接把要 load 的数据放进寄存器，或者将它放进 cache。

```

fetch( &ip);
fetch( &a[0]);
fetch( &b[0]);

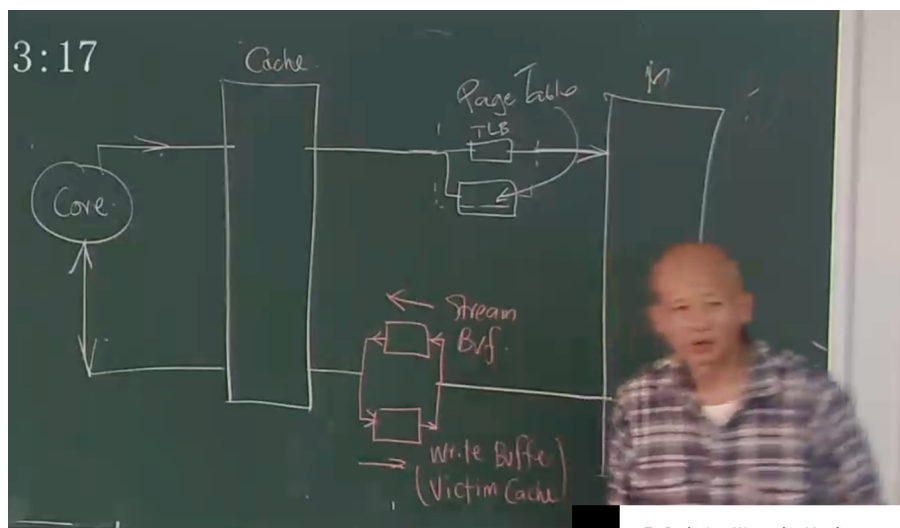
for (i = 0; i < N-4; i+=4){
    fetch( &a[i+4]);
    fetch( &b[i+4]);
    ip = ip + a[i] * b[i];
    ip = ip + a[i+1]*b[i+1];
    ip = ip + a[i+2]*b[i+2];
    ip = ip + a[i+3]*b[i+3];
}
for ( ; i < N; i++)
    ip = ip + a[i]*b[i];

```

(d)

知乎 @

这是一个比较好的例子，我们首先取第 0 个元素，避免 compulsory cache miss，然后在循环末尾，按照 cache line 的大小一次多读几个元素。但由于 prefetch 指令本身有时间开销，所以我们要做出一些权衡。



alpha 公司的策略是用 stream buffer 来存储指令预取的内容，而这对数据预取亦有作用：“1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%”，令人惊叹不已。

1.6 编译器优化

1.6.1 基本思想

- 对指令来说，在内存中重新安排各过程的位置，减少 conflict misses，或者预先填充指令来查看可能的冲突
- 对数据来说，有 merging arrays, loop interchange, loop fusion, blocking 等

1.6.2 合并数组


```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];
```

如是可以提高空间局部性，也可以减少 val 和 key 两个数组之间的冲突。

1.6.3 循环互换

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



容易看到，如果做了优化，那么可以在内循环中将 `x[i]` 整个读入缓存，是非常有利于效率的。

1.6.4 循环融合

```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {
        a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
```

1.6.5 分块

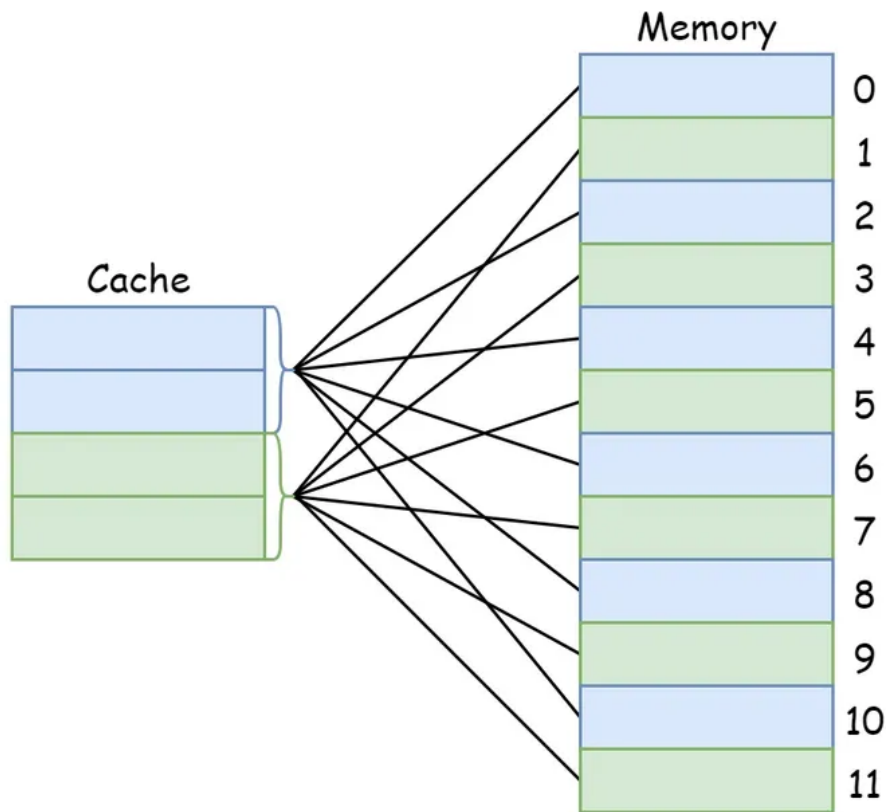
附录

cache 和 buffer 的区别

- 1、**Buffer**（缓冲区）是系统两端处理**速度平衡**（从长时间尺度上看）时使用的。它的引入是为了减小短期内突发I/O的影响，起到**流量整形**的作用。比如生产者——消费者问题，他们产生和消耗资源的速度大体接近，加一个buffer可以抵消掉资源刚产生/消耗时的突然变化。
- 2、**Cache**（缓存）则是系统两端处理**速度不匹配**时的一种**折衷策略**。因为CPU和memory之间的速度差异越来越大，所以人们充分利用数据的局部性（locality）特征，通过使用存储系统分级（memory hierarchy）的策略来减小这种差异带来的影响。
- 3、假定以后存储器访问变得跟CPU做计算一样快，cache就可以消失，但是buffer依然存在。比如从网络上下载东西，瞬时速率可能会有较大变化，但从长期来看却是稳定的，这样就能通过引入一个buffer使得OS接收数据的速率更稳定，进一步减少对磁盘的伤害。
- 4、TLB（Translation Lookaside Buffer，翻译后备缓冲器）名字起错了，其实它是一个cache。

组和路的区别

组是若干 cache line 的集合；路是每个组中 cache line 的数量。



2-ways associate by Rand

知乎 @Rand

Harvard Structure

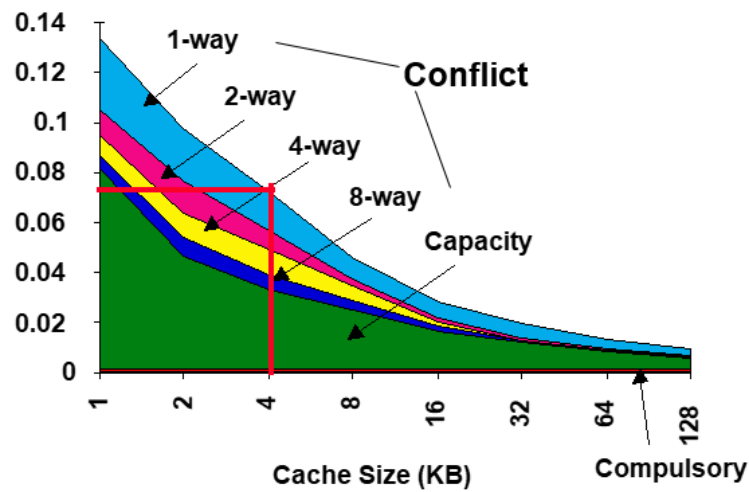
它的一大特点是分开的 cache（也即 I cache/D cache）。那么容易看到，这可以减少我们之前提到过的结构冒险，因为这样同一时钟周期中的取指令和取数据就不会冲突了。

那如果是 unified structure，那取数据的时候就需要 stall 一个额外的周期了（参照 Pipelining）

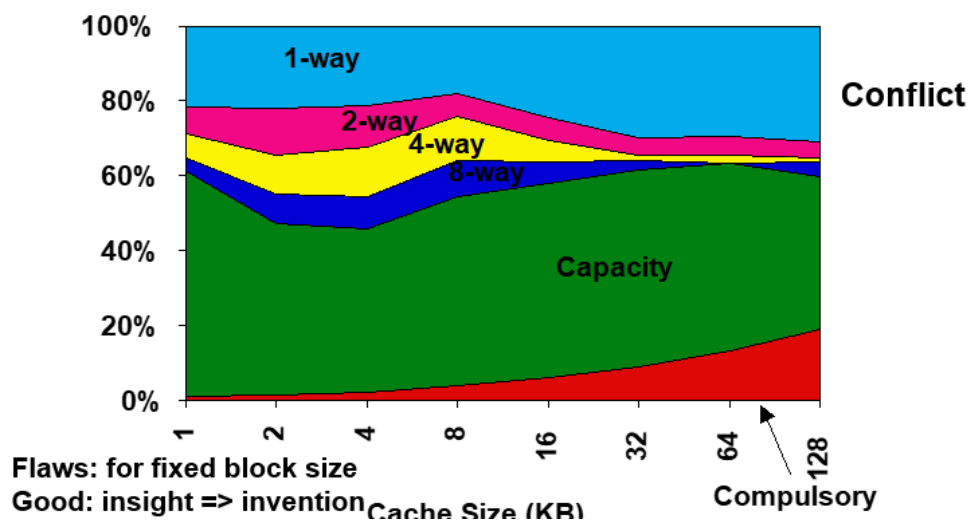
cache 的量化分析

2:1 Cache Rule

miss rate 1-way associative cache size X
= miss rate 2-way associative cache size X/2



3Cs Relative Miss Rate



4k 的时候，增加关联度能显著减少 miss rate。

64k 的时候（可以认为是因为 cache 显著变大了），则没什么显著效果。