Chapter 05 优化程序性能

- **内存别名使用(memory aliasing)**
- **每个元素的周期数(Cycles Per Element, CPE)**
- **延迟界限(Latency Bound)**: 当一系列操作必须严格按照顺序执行时, 会遇到的界限.

  吞吐量界限(Throughput Bound): 运用了所有并行能力得到的程序性能的终极限制.
- 功能单元的性能刻画:
  - **延迟(Latency)**: 完成运算所需要的总时间.
  - **发射时间(Issue Time)**: 两个连续的同类型运算之间, 需要间隔的最小时钟周期数.
  - **容量(Capacity)**: 能够执行该运算的功能单元的数量.
- **最大吞吐量:** 若发射时间为 I , 容量为 C , 则最大吞吐量为 C/I , 即每 I 个周期发射 C 条指令.
- **关键路径**
- $k \times 1$ **循环展开**: 每次循环, `i += k`
- $k \times k$ **循环展开**: 每次循环, `i += k` , 并设置 k 个累计变量 acc_0~acc_{k-1} . 前提是这个运算满足结合律和交换律(或者说你不在乎那些潜在的偏差)

  对延迟为 L , 容量为 C 的操作而言, 为了达到 `吞吐量界限` , 要求循环展开因子 $k \geq L \cdot C$

  > 比如浮点乘法有C=2个单元, 延迟L=5, 那么一条流水线里有5条乘法指令. 为了两次迭代中数据不相关, 需要5个积累变量. 而我们有两条流水线, 所以是 5 * 2 = 10 个积累变量.
- $k \times 1a$ **循环展开**: `acc = acc OP (data[i] OP data[i+1])`
- **寄存器溢出**: 不要把k设置得过大, 因为考虑到x86只有16个寄存器, 如果积累变量太多, 比如有20个, 那么就会有一些被存储在内存中(栈中), 访存将耗费更多的时间, 得不偿失.

Chapter 06 存储器层次结构

- **局部性(locality)**: 具有良好局部性的程序更多地倾向于从储存器层次结构的较高层次访问数据.
  - 空间局部性: 针对自己附近的内存位置

    步长为k的引用模式: k越小, 空间局部性越好
  - 时间局部性: 针对自己本身

    对于取指令而言, 循环体具有好的时间&空间局部性. 循环体越小, 迭代次数越多, 局部性越好.

- **随机访问寄存器(Random-Access-Memory, RAM)**
  - **静态RAM(SRAM):**
    - 更快, 更贵.
    - 作为高速缓存(Cache).
    - 双稳态, 对干扰不敏感.
    - 不需要不断刷新.(当然, 断电还是会没...)
  - **动态RAM(DRAM):**
    - 作为内存, 帧缓冲区(显存).
    - 对干扰非常敏感.
    - 需要不断刷新: 内存系统必须周期性读出&重写, 来刷新内存每一位.
  - **传统的DRAM:**
    - **`DRAM芯片`** 被分成 d 个 `超单元(supercell)`, 每个超单元由 w 个DRAM单元组成.
      从而一个 $d \times w$ 的DRAM总共存储了 dw 位信息.
      其中, d进一步分解为 $d = r \times c$, 因为这些超单元被组织成了 $r \times c$ 矩阵
    - 信息通过 `引脚(pin)` 流入和流出芯片. 每个引脚携带1位信号.
    - 由于超单元被组织成了矩阵, 所以访问(读/写w位)某个超单元时,
      需要依次地先后提供 `行地址(RAS,RowAccessStrobe)` 和 `列地址(CAS)`.
      这是因为在真实的设计中, 二者共享同一组 `addr地址引脚`.
      收到行地址后, DRAM会把**一整行**都复制到 `内部行缓冲区`;
      收到列地址后, DRAM从行缓冲区中复制出对应的超单元(包含 w 位信息).
      这种设计的优点是**减少了引脚的数目**, 但缺点是由于必须分两步发送地址, **增加了访问时间**.
    - **`DRAM芯片`** 聚合封装构成了 **`内存模块`**.
      例如将8个$8M \times 8$ 的DRAM芯片组织起来, 分别编号为 `DRAM0` ~ `DRAM7`
      当要取出内存地址A处的一个字(8Byte)时, `内存控制器` 将地址A转化为超单元地址(i,j)并广播给每一个 `DRAM芯片`.
      作为相应, 每个DRAM芯片输出它的 $w = 8$ 位内容. `内存模块电路` 收集这8个输出并合成一个64位字, 最终返回给内存控制器.
    - 由于 `地址引脚` 数目决定了超单元矩阵的长宽, 且行和列共享地址引脚, 所以应该尽可能将矩阵组织成方阵(或长宽比为2:1的长方形), 以节省资源.
- **只读存储器(Read-Only Memory, ROM)**
  存储在ROM中的程序通常被称为 `固件(firmware)`, 当一个计算机系统通电之后, 他会运行存储在ROM中的固件.
  一些系统在固件中提供了基本的输入输出函数, 比如PC的 `BIOS(基本输入/输出例程)`.
- **CPU和主存的数据互通**

- **机械硬盘(Disk, 磁盘):**
  - 基本构造: `盘片(platter)` -> `表面(surface)` -> `磁道(track)` -> `扇区(sector)`

其中, 扇区之间以 `间隙(gap)` 分隔, 间隙中不存储 `数据位`, 而是存储用来**标识扇区**的 `格式化位`.

`柱面(cylinder)`: 所有盘片表面上, 到主轴中心的距离相等的磁道的集合. 也就是某个圆柱和所有的表面的交集(即k个圆, k=表面的数量).

- 决定 **`容量`** 的因素: `记录密度(bit/inch)`, `磁道密度(tracks/inch)`, `面密度(=记录密度x磁道密度)(bit/inch^2)`
- 决定 **`访问时间(AccessTime)`** 的因素:
  - `寻道时间`: 平均3~9ms, 最慢20ms
  - `旋转时间`: $T_{max\_rotate} = \frac{1}{RPSec} = \frac{60}{RPMin}$, $T_{average\_rotate} = \frac{T_{max}}{2}$
  - `传送时间`: 由于磁盘以扇区大小的块来读/写数据, 传送时间 $T_{max\_trans} = \frac{T_{max\_rotate}}{\text{平均每条磁道的扇区数}}$
- **逻辑磁盘块**: 磁盘将它们的构造呈现为一个简单的视图: 包含B个扇区大小的逻辑块的序列. 磁盘控制器负责维护 `逻辑块号` 和 `物理磁盘扇区` 之间的映射关系.
- **DMA(Direct Memory Access)**: 设想一个1GHz的处理器时钟周期为1ns, 而读取硬盘需要16ms, 在这段时间可以执行1600万条指令. 所以CPU当然不能干等着, 会造成浪费. 当磁盘控制器收到CPU的读命令后, 它将逻辑块号翻译成扇区地址, 然后读取该扇区的内容, 并将这些内容直接传送到主存. 这个过程中不再需要CPU的干涉. 称为 `直接内存访问(DMA)`.

- **固态硬盘(SSD)**
  - SSD由B个 `块Block` 组成, 每个 `Block` 下有若干个 `页(Page)`. 数据以 `页` 为单位进行读写.
  - **擦除**: 把一个 `block` 的所有页的所有bit都设置为1. 注意不能单独擦除某一个 `页`!
  - **写入**: 写入必须针对一个已经被擦除的页. 写入操作通过把部分bit设置为0而其余bit保持1不变, 来写入信息.
    所以无法重复写入一个已被写入且未被擦除的page, 因为其中有一部分已经是0了!
    随机写要比顺序写慢, 因为擦除Block需要时间(1ms左右);
    而且如果向一个已经有数据的Page_p写入, 则需要把对应的**整个block**都复制到一个全新的位置, 这样才能对页p顺利执行写入.

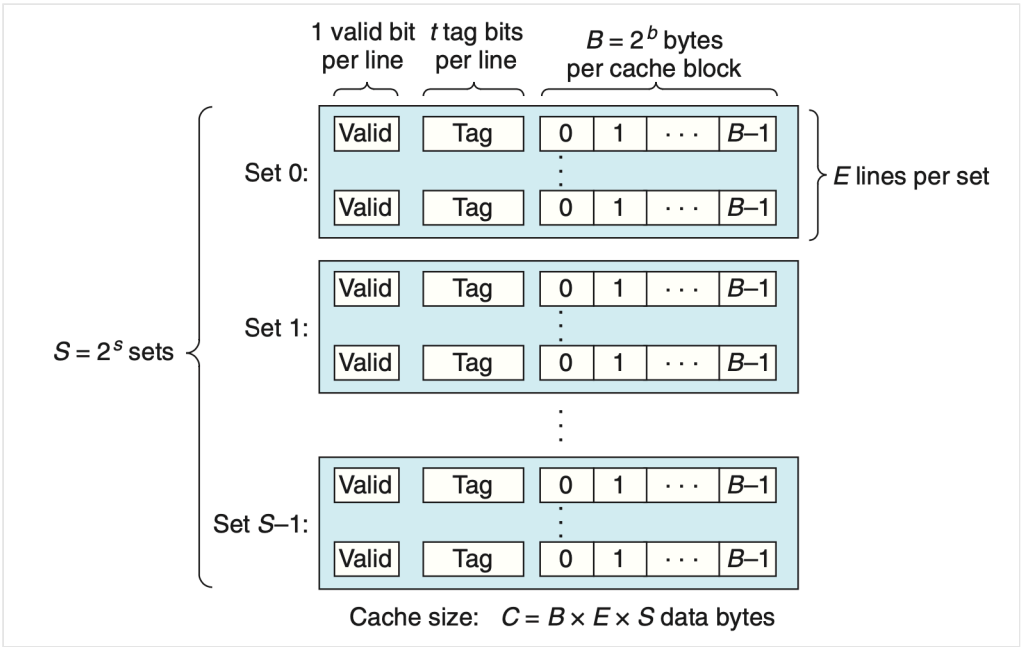
- **Cache**:
  - **缓存的通用原理:**
    将第 $k+1$ 层的数据的一个子集拷贝到第 k 层, 作为副本(缓存). 每次传输的 `传送单元` 以一整个Block为单位.
  - **命中(Hit):**
    当程序需要第 $k+1$ 层的数据时, 先去询问第 k 层, 如果找到了, 称为缓存命中.
  - **不命中(Miss):**
    如果没有找到, 那就只好去 $k+1$ 层找包含数据d的那个块, 取出并存放到 k 层.
    如果 k 层已经满了, 就会涉及到 `覆盖(替换/驱逐)` 现存的一个缓存块.
    决定该覆盖哪个是由 `替换策略` 来决定的.
  - **不命中的类型:**

    - 冷不命中(Cold Miss): 起初cache还是空的. 这种不命中
    - 冲突不命中(Conflict Miss): 虽然第 k 层容量仍然充沛, 但是两个块被映射到了 Cache 的同一个块. 造成重复替换.
    - 容量不命中(Capacity Miss): 因为第 k 层作为缓存, 当然存不下整个 k+1 层, 所以当 `工作块` 太大时, 就会造成这种Miss.
  - **放置策略(Placement-Policy):** 当发生不命中后, 新搬上来的块放在哪里?
    - 对于靠近CPU的缓存, 可以用昂贵的硬件实现"任何块可以缓存到任何位置". 速度很快, 但定位代价和硬件成本高昂.
    - 其余的缓存通常使用更严格的放置策略, 比如第 $k+1$ 层的块 i 必须放在第 k 层的 $\mathrm{i} \% 4$ 位置. 这可能造成 `冲突不命中` .
  - **替换策略(Replacement-Policy)**:
    - 比如: 随机替换策略, 最近最少被使用(LRU)替换策略, ...

- **通用的高速缓存组织结构**

  考虑一个计算机系统, 存储器地址有 m 位, 即具有 $M = 2^m$ 个不同的地址,

  则这台机器的高速缓存将被以如下方式组织:



即Cache包含 $S = 2^s$ 个 `高速缓存组(cache set)`

每个 `高速缓存组` 包含 E 个 `高速缓存行(cache line)`

每个 `高速缓存行` 由 3 部分组成:

- 1 个 `有效位(valid bit)`
- t 个 `标记位(tag bit)`
- 大小为 $B = 2^b$ 字节的 `数据块(block)`

当我们要访问地址Addr处的一个字时, 我们对Addr进行分割解析: `t位标记位|s位组索引|b位块偏移` :

`s位组索引` 被解析为一个无符号整数, 表示这个字(如果在cache中)必须存储在 `哪个` 高速缓存 `组` ;

一旦定位到了高速缓存 `组` , `t位标记位` 就可以确定(通过硬件硬搜)在这组的哪一行包含了这个字(注意每行高速缓存行都有一个t-bit标记位)

当且仅当 `标记位` 相互匹配, 且 `有效位` 被设置为"有效"时, 我们才认为组中的这一行包含这个字.

如果确认了包含, 那么 `块偏移` (单位为字节)给出了所要的字在这一行的数据块中的位置.

> `标记位` : 考虑到内存中的很多块都会映射到Cache的同一个组(Set),
> 所以需要用Addr的高Bit们(t位标记位)确认当前Cache的这个Block是不是你需要的那个位置的数据.
> `有效位` : 考虑到计算机中只有0/1, 没有第三个数表示"NULL", 所以只好用一个bit来标记这个地方到底是不是合法数据.

- 最简单的**直接映射高速缓存(Direct-Mapped Cache)**:
  - **组选择**: 考察中间s位, 看成无符号数, 作为组索引.
  - **行匹配**: 由于只有1行, 所以匹配当且仅当设置了有效位, 且标记位(共t位)匹配.
  - **字选择**: 把高速缓存行的数据块看做一个数组, 然后根据块偏移(标识了字节偏移量)访问相应字.
  - **不命中时的行替换**: 由于是直接映射高速缓存, 每个高速缓存组只包含一行, 所以替换策略很简单: 用新取出的行替换当前的行.
  - **冲突不命中**: 当程序访问大小为2的幂的数组时, 很有可能发生冲突不命中.
    比如Cache的总大小为 2^C , 那么比如对任何 2^{C+k} 的两个数组的交替访问就会频繁造成冲突不命中. 例如:

```
float dotProduct(float x[8], float y[8]){
    float sum = 0.0;
  for(int i = 0; i < 8; i++){
      sum += x[i] * y[i]; // 频繁造成冲突不命中!
  }
  return sum;
}
```

- **E路组相联高速缓存**:
  - **组选择**: 同上.
  - **行匹配**: 大致同上, 只不过之前只需要确认唯一的一行是否匹配. 而这里需要检查E行中的每一行.
    而匹配的过程是把 `t位标记位` 和 `有效位` 看做 `key` , 进行比对, 然后返回块的内容 `value` .
    这就很像是进行(key, value)搜索. 所以才叫做 `相联存储器` (回忆C++中的关联容器).
  - **字选择**: 同上.

- **不命中时的行替换**: 如果有空行, 当然直接利用空行. 否则会采用更复杂的策略. 基本基于局部性原理. 比如 `一段时间内最不常使用原则` / `距离最后一次访问的时间最久原则` 等等. 这些策略的实现需要额外的时间和硬件, 不过相比于不命中带来的巨大开销, 尤其是对于那些远离CPU的缓存, 这一切还是值得的.

- **全相联高速缓存**:
  - **组选择**: 只包含一个组, 即组0. 所以对于传入的内存地址Addr, 默认只能选择组0.
  - **行匹配**: 和组相连高速缓存一样.
  - **字选择**: 同上.

  全相联的确做到了"任何数据可以放在任何位置",

  然而由于需要并行地搜索许多相匹配的标记, 所以造价昂贵, 且只适合做很小的高速缓存.

- 三种类型的 `性能比较` :

  `首先指出， 只有当容量相同时， 对这三种缓存的讨论才有意义.`

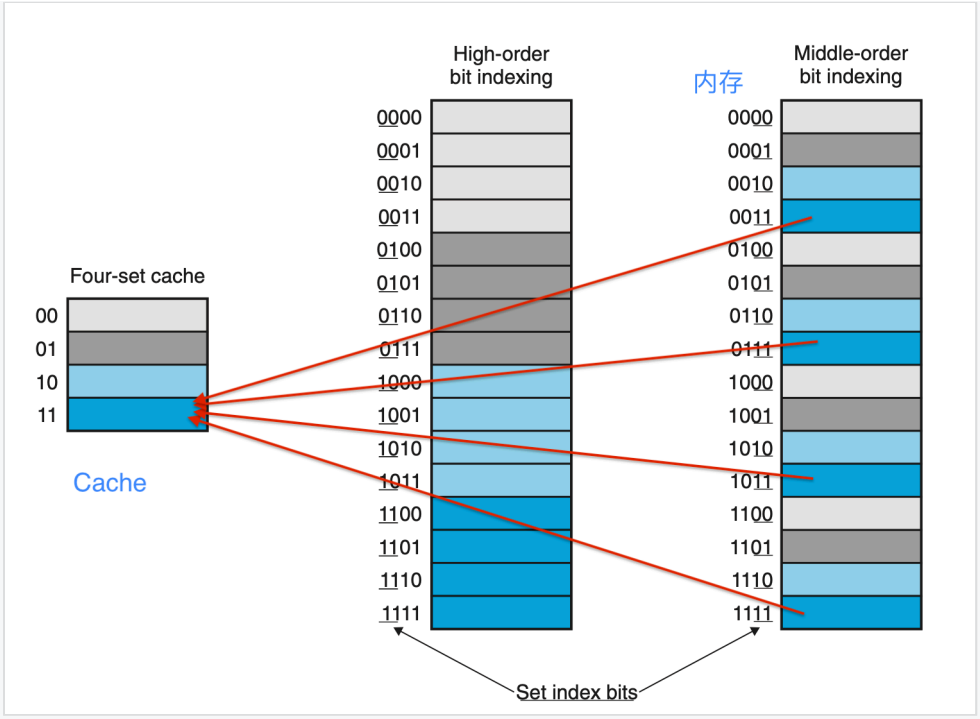
  `直接映射高速缓存` 是很朴素的想法, 先用中间的bit来确认在组号, 然后看这个组是不是我真的想要的那个组(靠高位bit判断)

  不过这种朴素的想法会面临一个问题: 当两个频繁使用的内存块, 恰巧对应相同的Cache组号时. 那就会来回替换, 称作**抖动**.

  于是对每一个组(set)设立多个行, 构成 `E路组相联高速缓存` . 此时每一组里不再只有一行, 而是多行. 同样, 先用中间的bit确认组号. 这时候, 这个组里有很多行, 所以要 `逐一比对` 是不是我想要的那个行. 而由于有多个行, 比如是2行, 那就可以轻松解决2个内存块可能争抢同一个Cache组的尴尬.

  至于 `全相联高速缓存` , 不再进行分 `组` 了, 大家都是 `组0` , 这就意味着内存中的任何数据都由 `组0` 负责. 真正意义上实现了"任何数据可以缓存到Cache的任何位置". 考虑到 `相联度` 很高, 一方面大大降低了出现抖动的可能性, 另一方面由于涉及同时搜索非常多的行, 很难使之速度很快, 命中时间偏长.

  > cpy关于分 `组(Set)` 的意义的想法:
  > 这里红色的箭头可以理解为"被...负责". 比如这里分成了S=4个 `组` , 于是整个内存也就被间隔着分成了4种颜色, 每个颜色对应Cache中的一个 `组` .



- `名词辨析` : 块(Block), 行(Line), 组(Set)
  - 块: 一个固定大小的信息包(B个Bytes), 在 k 层和 k+1 层来回传送 .
  - 行: 对块的封装, 即包含 `有效位(v)` , `标记位(t)` , 和 `块(数据块)(DataBlock)` .
  - 组: 一个行/多个行的集合.

- **写命中(write hit)**: 即我们要写的字已经在缓存中了
  - **直写(write through)**: 立即将刚才更新过的高速缓存块(第k+1层)写回到紧接着的低一层(第k层)去.
  - **写回(write back)**: 尽可能地推迟更新, 直到**替换算法**要驱逐这个被更新过的块时, 再执行向低一层的写入. 这需要高速缓存为每一个高速缓存行维护一个额外的 `修改位(dirty bit)`
- **写不命中**:
  - **写分配(write allocate)**: 加载相应的块到高速缓存中, 然后更新这个高速缓存块.
  - **非写分配(not-write-allocate)**: 避开高速缓存, 直接把这个字写到低一层中去.
- 通常, **直写**和**非写分配**搭配; **写回**和**写分配**搭配.

  虽然具体采用何种策略通常是对程序员不可见的,

  但我们可以假设采用的"写回(Write Back)+写分配(Write Allocate)"的模型. 因为这种模型在试图**利用局部性**.

- 一个更接近真实的高速缓存层次结构:
  - 现代处理器的 `L1Cache` 包括独立的 `i-cache` 和 `d-cache` , 即 `指令高速缓存` 和 `数据高速缓存` .

    这样做的原因是, 比如指令内存通常是只读的, 所以针对两种缓存可以采用不同的 `超参数` 来优化, 让整体效率更高.

End Of Note, 313 Lines.