Cache

Memory Hierachy

processor-memory gap：高速处理器的性能常常被内存带宽和延时所限制

- latency：单次访问需要的时间，一般来说远大于时钟周期
- bandwidth：单位时间内访存次数

Little's Law：同时能传送的数据量是 bandwidth 和 latency 的乘积

理想的内存是：零延时，无限容量，便宜，无限带宽，然而这些要求是矛盾的

- 越大越慢：需要时间寻址
- 越快越贵
- 带宽越高越贵

故解决方法是将内存组织成层次结构，从上到下容量上升，速度变慢。保证大部分需要的数据都在比较快的层级

一般的层次：寄存器，cache，主存，外存

现在架构都是由硬件自动完成对不同层级间数据的移动，对于程序员来说是透明的

不需要知道 cache 就可以写出正确的程序，但是如果要程序更快，就需要考虑到 cache 友好性

Cache Basics

Caches

Momory Hierachy 能够 work 的关键就是要确保较快的内存中确实有处理器需要的数据，这是由内存的局部性保证的，局部性可分为时间局部性和空间局部性

- temporal：程序倾向于使用短时间内使用过的数据
- spacial：程序倾向于使用空间上相邻的一组内存
  - 循环时的指令
  - 数组

cache 的基本思想就是利用这两种局部性

- temporal：存储最近访问过的数据，并认为在不远的将来程序会再次访问这些数据

- spatial：将访问地址相邻的数据也一并存入，并认为邻近的数据不久后会被访问

cache 一般要和流水线紧密地结合在一起，最好能达到 1 cycle 的访问时延，使得指令不会因为访问 cache 而 stall，但是对于高速的流水线来说，cache 不能达到很大，解决方案是：分层 cache

Cache Terminology

block：被缓存的最小单元

frame：存储一个 block 的位置

hit：在 cache 中有需要的数据

miss：在 cache 中没有需要的数据

miss ratio：访问 cache 时的 miss 的比率

hit time：如果 hit，访问需要多少时间

miss penalty：如果 miss，在更低层的内存找到数据的时间

Hierarchical Latency Analysis

考虑基本的 load 过程：在处理器发出 load 请求时，比较需要的地址和 cache 里的 tag，如果 hit 则返回数据，否则从主存中寻找数据，然后选择一个 victim block 换出 cache，将新数据存入 cache 并返回

考虑第 i 层内存，本身的访问时间是 t_i ，而感知到的访问时间是 T_i ，对于某一层来说

- 如果 hit 了，访问时间是 t_i
- 如果 miss 了，访问时间是 $t_i + T_{i+1}$
- hit 几率 h_i ，miss 几率 m_i
- miss rate 仅仅指在第 $i-1$ 层 miss 的几率

则 AMAT (Average Memory Access Time) 为

$$T_i = t_i + m_i \times T_{i+1}$$

T_{i+1} 即为 miss penalty

希望达到的效果是 $T_1 \approx t_1$ ，为此需要

- 保证 m_i 尽可能小
  - 提升容量会降低 m_i ，但同样会导致 t_i 上升

    - 通过更好地管理，如替换（预测不需要的数据）和预取（预测将要需要的数据）
- 保证 T_{i+1} 尽可能小
    - 提升速度，提升价格
    - 引入中间层，如分层 cache

Classifying Caches

cache 可根据多种标准进行分类

- block placement：在 cache 中组织存放 block 的方法
- block identification：如何在 cache 中找到 block
- block replacement：在 miss 时如何换出 block
- write strategy：如果发生写，如何处理

block placement

可以分为三类

- fully associative：每个 block 可以放在任意位置
- X-way set associative：将 cache frame 分组，每个 block 对应固定的一个组，但是可以放在组内任意位置
- direct mapped：每个 block 对应一个固定的位置

block identification

对于 direct mapped，将主存地址分为三部分：tag，index，offset

通过 index 找到 cache 对应位置，比较 tag 确认是否是需要的 block，然后通过 offset 确定具体要读的数据

对于 x-way set associative，同样是将主存地址分为 tag，index，offset，通过 index 找到组，比较组内每行的 tag，通过 offset 确定要读的数据

对于 fully associative 则需要与每一行进行比较

associativity tradeoff

- 更高的 associativity 带来更高的 hit rate（block 存放的可能性多）
- 增加了访问时间
- 硬件成本上升，需要更多的比较器

而且增大 associativity 会有边际效应

block replacement

对于 direct mapped 来说没有选择，只能替换对应的那个

一般对于 associative cache 来说选择方法有

- random
- LRU (Least Recently Used)
  - 每次访问需要更新 LRU 状态
  - 对于小的 set 来说可以实现真 LRU
  - 4-way，8-way 等可以实现伪 LRU
- FIFO
- NMRU (Not Most Recently Used)，本质 FIFO，只不过不替换最近用过的

对于 2-way 可以实现 perfect LRU，用 1 个 bit 就能跟踪访问情况（把访问的置 1 另一个置 0）

更大的 set 一般不选择实现 perfect LRU（逻辑复杂）

- NMRU
- 层次 LRU，如将 4-way 的分成两组，记录 MRU 的组和每组内的 MRU way
- victim and next victim：只记录 victim 和下一个 victim

> 层次 LRU
>
> 将 set 分成多个 group，只记录 MRU group 和每个 group 的 MRU way
>
> 替换时选择一个 non-MRU group 的 non-MRU way
>
> victim/next victim
>
> 每个 set 中记录两个：V (Victim) 和 NV (Next Victim)，其余 block 都是 O (Ordinary)
>
> - cache miss 时，替换 V，将 NV 转为 V，随机选择一个 O 作为 NV
> - cache hit NV 时，随机选择 O 为 NV，原来的 NV 变为 O
> - cache hit V 时，NV 转换为 V，随机选择 O 为 NV，将 V 变为 O
> - cache hit O 时，不改变状态

write strategy

对于 cache hit 的情况，可以分为两种

- write through：同时写 cache 和主存
  - 简单，确保每一层的内存都是最新的数据，考虑一致性时简单
  - 更多带宽需求，没有合并写入
- write back：只写 cache，换出时才写内存
  - 换出前多次写 cache 也只需要一次写内存，减小带宽需求
  - 需要多一个控制位指示是否修改

对于 cache miss 的情况，同样可以分为两种情况

- no write allocate：只写主存
- write allocate：将 block 拉入 cache，写 cache

一般的组合是 write through + no write allocate 或是 write back + write allocate

Multi-level Caching

L1 cache 分为指令和数据（I cache and D cache）

- 主要以低访问延时为设计目标
- small, lower associativity
- 并行存入 tag 和数据

L2 cache 不区分指令数据

- 设计上考虑访问延时和 hit 率的平衡
- usually large, high associativity
- tag 和数据的存入是串行的

一般是串行访问各级 cache，即 L1 miss 了才向 L2 发出请求

High Performance Cache

Cache Performance

AMAT = hit time + miss rate $\times$ miss penalty

提高性能可选的方法有

- 降低 hit time
- 降低 miss rate
- 降低 miss penalty
- 并行技术
- 提升 cache 带宽

Reduce Hit Time

Reduce Hit Time

small and simple cache

cache hit time 会随着 capacity 和 associativity 的上升而增加

parallel access to tag and data

也可以并行地访问 tag 和 data，即使此时还不知道是否 hit

avoiding address translation

cache 应该以虚拟地址还是物理地址索引?

- 虚拟地址索引
  - 上下文切换要冲刷 cache
  - memory alias：同一个物理地址可能出现在 cache 的两个位置
- 物理地址索引
  - 需要转换地址
  - TLB 在 critical path

还有一种解决方案是 virtually-indexed physically tagged cache，即页号作为 tag，offset 拆分为 index 和 offset 用于索引 cache

如果 cache size $>$ page size $\times$ associativity，则 cache index 的高位在页号中

> 考虑 page size 为 2^n，cache size 为 2^m，且 associativity = 2^i，如果 cache size $>$ page size $\times$ associativity，则有 $2^m > 2^{n+i}, m > n + i$，设 cache line 大小为 2^k，则 cache offset 长度为 k，cache index 长度为 $m - i - k$，总的长度为 $m - i > n$，超过了 page offset 的位数

- 限制 cache size
- 当写一个 block 时，寻找所有可能的位置
- 限制 OS 的页面替换，保证 virtual index = physical index，即 page coloring

way prediction

用额外的 bit 来预测取 set 中的哪个 way，可以在 x-way associative cache 达到近似 direct mapped 的速度

Reduce I cache Hit Time

超标量处理器对 cache fetch 的带宽有要求，而 fetch bandwidth 限于 I cache bandwidth，BP 以及控制流，而 I cache 存放的是编译生成的顺序代码而非动态执行的代码

解决方法：缓存动态执行的指令序列，即 trace，然后根据 trace 来获取 cache block

Reduce Write Hit Time

写需要 2 个 cycle，一个用来匹配 tag 一个用来写

解决方案

- 同时读写的 RAM，能在 tag miss 的时候保存旧值
- CAM tag cache

- pipeline，将要写的数据缓存，当检查下一个 tag 时写数据

Reduce Miss Rate

cache miss 的原因可以分为三类

- Compulsory miss：第一次访问某个地址总是 miss，即使 cache 容量无限也不能避免
- Capacity miss：由于 cache 体积不足产生的 miss，如 working set 大于 cache 容量，产生于 fully associative cache
- Conflict miss：由于 collision 产生的 miss，产生于 x-way associative cache

除此之外还有 coherence miss，因为 cache 一致性产生的 miss

large cache size

减少 miss rate 可以通过增加 cache 容量

> 倍增 cache 容量，miss rate 下降约 $\sqrt{2}$

也可以通过增大 block size

- 优点
  - 减小 tag 体积
  - 更好利用空间局部性
- 缺点
  - 浪费带宽
  - block 数目减小导致 conflict 增加

large associativity

增大 associativity

> 容量为 N 的 direct mapped cache miss rate 等于容量为 $N/2$ 的 2-way cache

multiple level cache

分层 cache，这里有两种不同的 miss rate 定义

- local miss rate：当前 cache miss 次数除以对当前 cache 的访问次数
- global miss rate：当前 cache miss 次数除以 CPU 发出的访存次数

> Inclusion Policy
>
> Inclusive multilevel cache
>
> - Inner cache holds copies of data in outer cache

> - External coherence snoop access need only check outer cache
>
> Exclusive multilevel cache
>
> - Inner cache may hold data not in outer cache
> - Swap lines between inner/outer cache on miss

compiler optimization

通过编译器的优化减小 miss rate

- 重构代码使得代码 cache 友好
- 避免数据进入 cache，如替换前只会访问一次的数据，需要软件告知硬件 no-allocate
- 将不会再用到的数据 kill 掉，如流式数据，只有空间局部性没有时间局部性

最简单的例子就是行优先存储的矩阵，`A[i][j], A[i][j + 1]` 内存中相邻，而 `A[i][j], A[i + 1][j]` 在内存中距离很远

Reduce Miss Penalty

victim cache

使用 victim cache，即用一个 fully associative buffer 来存储被换出的 block

- 优点
  - 避免两个 cache line 交替换入换出的情况
  - 减少 conflict miss
- 缺点
  - 如果 victim cache 和 L2 cache 串行访问，会增大 miss latency
  - 复杂

cirtical word first & early restart

critical word first 即最先从内存获取需要的数据，同一 block 中的其他数据之后按顺序填入 cache

early restart 即数据按序填入 cache，需要的数据到达 cache 了直接启动 stall 的处理器

write buffer

使用 write buffer，将换出的 block 存在 buffer 中

- 处理器不在 write 上 stall，且读 miss 可以先于 write 直接访问主存

- 当 read miss 时，检查 write buffer 中的地址，如果不匹配则读主存，否则读 write buffer 中的数据

write buffer 好处是可以 merge，即多个写可以 merge 到同一个 entry

Increase Cache Bandwidth

non-blocking cache

使用非阻塞的 cache

- 允许在有等待处理的 miss 的情况下访问 cache
- 允许并行访问 cache
- MLP (Memory-Level Parallelism)

处理多个未完成的访存可以使用 MSHR (Miss Status Handling Registers) 或是 MAF (Miss Address File)

MSHR 中条目包括有效位，地址，是否 issue 到内存系统等

当发生 miss 时，检查 MSHR 中有没有同一个 block 的条目

- 如果有，则在其中再添加一个 load/store 条目
- 否则分配一个新的 MSHR
- 如果没有可用的条目，stall

当从下一级内存返回一个 subblock 时

- 检查有哪个指令在等待该 subblock，将其转发到对应 function unit，从 MSHR 中去除该 load/store 条目
- 将数据写入 cache
- 如果是最后一个 subblock，删去 MSHR 条目

Memory-Level Parallelism

MLP 即同时处理多个访存要求

一般的 cache 替换目标都是尽量降低 miss 次数，这一出发点是基于减少 miss 次数可以减少 memory stall time 的前提，而 MLP 打破了这个前提

- 消除一个单独的 miss 对性能提升大于消除一个并行的 miss
- 消除一个高延时的 miss 对性能的提升大于消除一个低延时 miss

multiporting & banking

还可以通过增加内存的读写口来增加带宽

如果是真实的多读写口，会导致芯片面积增加，hit time 增加

- virtual multiporting：时分单口，内存频率高于 CPU 频率，这样一个 cycle 可以访问不止一次，在 Alpha 21264 中使用
- multiple cache copies：两份 cache，有独立的两个读地址线和数据输出线，可以并行处理 load，同时共享一条写地址线和数据输入线，一次 store 更新两份 cache，在 Alpha 21164 中使用

或是通过 cache banking，将地址空间划分为不相交的 bank，对每个 bank 独立读写

- 不增加存储区域面积
- 会有 conflict：多个访存指向同一 bank
- 需要输入输出 interconnect

banking 是一种通用的解决方案，为了解决单个内存空间访问延时过大和并行访问的问题，将地址空间划分为多个独立的 bank，可以并行或串行访问，对不同 bank 的访问在时间上可以重叠

问题在于如何划分