Main Memory

Main Memory

主存是所有计算系统中重要的组成部分，必须可扩展

主存是处理器和外存的桥梁

DRAM 和 memory controller 已经无法满足所有要求

Graphic Data RAMs：GDDR & HBM

Major Trends

对主存的需求：容量，带宽，QoS

主存的能耗也成为了一个设计要点（DRAM 即使不使用也要耗电）

DRAM 的推进也逐渐走到尽头

对容量的需求主要来自：多核系统（比起单核更大的工作集），数据密集型应用，云计算等

但是主存容量的增长速度跟不上核数的增长速度

DRAM vs. SRAM

DRAM 的特点是

- 访问慢（有电容）
- 高密度（1 transistor 1 capacitor）
- 低成本
- 需要刷新

SRAM 的特点是

- 访问快（无电容）
- 低密度（6 transistor）
- 高成本
- 不需要刷新

DRAM 将数据存储在电容中，因此制程难以进一步扩展

DRAM Read

一般将内存组织成二维阵列

读内存的过程为

- 解码行地址，驱动 word line（word line 用于打开三极管）
- 读出一行数据（从 bit line 读出），缓存至 row buffer
- 用放大器将读出信号放大
- 解码列地址，读出需要的数据送至输出线
- 重新充能 bit line，为下一次读做准备

访问延时主要是由读取行（以及放大）的时间决定的

Destructive Read：读取之后原本电容里的值就消失了，必须在下一次读取前再次从 row buffer 写回电容

row buffer 中的内容就称为一个 DRAM row

- 每次读取一整行进入 buffer
- 读取其中某一个 block
- 读入 row buffer 很慢，但是从 row buffer 读出是比较快的，因此最好同一行的多次读写组织在一起（称为 row buffer hit）

> Page Mode DRAM
>
> - 每个地址都是一个二元组 `<row, column>`
> - 访问 closed row 时需要
>   - activate：将行的内容读入 row buffer
>   - read/write：读写其中数据
>   - precharge：关闭 row，重新充能 bit line，为下次读取做准备
> - 访问 open row 不需要 activate，直接读写即可

采用 page mode 的 DRAM，如果当前读取的地址是一个 open row，则可以根据列地址直接读出，而不用再次激活行

Fairness in Sharing

DRAM 是共享资源，需要被多个处理器访问，访问公平性的定义有很多种

- Max-min fairness
- Proportional fairness

DRAM Bank & Rank

访问一个单独的内存数组需要比较长的时间，且不支持并行访问

因此可以将内存分成多个独立不重叠的 bank，可以各自访问，对不同 bank 的访问在时间上可以重叠

考虑一个 128 M $\times$ 8 bit 的 DRAM chip

- 27 根地址线：17 行地址 + bank 选择，10 列地址
- 8 个 bank，3 bit 用于选择 bank
- 每个 bank 中 2^{14} 行，2^{18} 列（8 bit $\times$ 1024），每个 bank 有独立的 row buffer

rank：每个 chip 只能提供 8 bit 的输出，为了增大带宽，将多个 chip 组合在一起成为一个 rank，同步控制

- 对一条指令响应
- 共享数据线和地址总线，但是每个 chip 提供不同的数据
- e. g. 8 个 chip 可以提供 64 bit 的数据位宽

一个控制器可以接多个 rank，由控制器来片选

多通道：使用 channel，每个 channel 对应一个独立的控制器，多个 channel 可以独立工作

DRAM Subsystem

Channel：一个内存控制器连接一个 channel，多个控制器就有多个 channel，channel 间完全独立，互不干扰

DIMM (Dual In-line Memory Module)：也就是一般而言的内存条，一个 channel 可以连接多个 DIMM

Rank：多个 DRAM 芯片的组合，一般一个 DIMM 两个 Rank，正面反面各一个，用选择器选择读取的 rank

Chip：一个 rank 上多个 chip，多个 chip 的位宽相加组成了 rank 的位宽，所有 chip 共享控制线

Bank：chip 内有多个 bank，每个 bank 有独立的 row buffer，各 bank 间可以并行工作，用地址线的某几位来选择 bank

Row & Column：一个 bank 是一个二维阵列，读取时读取一行然后从行中读取需要的数据

Latency Components of DRAM Operation

一次 DRAM 的操作延时由以下部分组成

- 信号从 CPU 到 controller 的时间
- controller 延时
    - queuing & scheduling
    - 将访存指令转换成基本的 IO 指令

    - 将信号传送到 DRAM
- DRAM bank
    - 如果是 open row，只需要 CAS (Column Address Strobe)
    - 如果是 closed row，需要 RAS (Row Address Strobe) + CAS
    - 如果没有 precharge，则是 PRE + RAS + CAS
- 将数据从 DRAM 传送到 controller 的总线延时
- 数据从 controller 到 CPU

如果多次访问 DRAM 的不同 bank，访存请求之间可以 overlap，虽然没有降低延时但是增加了 throughput

Multiple Banks and Channels

多 bank 可以使得并发访问 DRAM

如果是多 channel 可以达到同样的效果，且由于 channel 有自己独立的数据总线，故提升了总的带宽

并发度高需要减少 bank 或 channel 的 conflict：多次访存读同一个 bank/channel

需要良好设计地址中的哪几位决定 bank/channel

- 低位的 bit 有更大的熵
- 可以使用 hash function

multiple channels tradeoff

- 优点
    - 提升带宽
    - 更多并发的访问
- 缺点
    - 成本高：更多线，芯片需要更多引脚

Address Mapping

考虑一个单通道系统，内存总线 8B，2GB 内存，8 bank，16k row & 2k columns per bank，cache block 大小为 64B

Row interleaving：地址线从高到低为

14 bit 行地址，3 bit bank 选择，11 bit 列地址，3 bit 总线地址

这样一整行存在连续的 bank 中，读取连续的 cache block 是流水线的形式（在同一 bank 中）

Cache block interleaving：地址线从高到低为

14 bit 行地址，8 bit 高位列地址，3 bit bank 选择，3 bit 低位列地址，3 bit 总线地址

连续的 cache block 存放在连续的 bank 中，可以并行读取

Bank Conflict

多个访存操作访问同一个 bank，conflict 的发生情况取决于访存模式

DRAM 可以将地址随机映射到 bank（如将固定位置的两个 3-bit 异或得到 bank 选择地址），减少 conflict 的发生几率

Memory Controller

确保 DRAM 能正确工作（在满足时间限制的情况下）

缓存并调度访存请求

管理能耗

DRAM controller 可以出现在多种位置

- 放在芯片组
  - 更灵活，可以在系统中支持不同种类的 DRAM
  - CPU 的能量密度降低
- 放在 CPU 里
  - 减少访存的时延
  - 更高的带宽

DRAM Refresh

DRAM 中的内容会随时间消失，需要定期读取并重新写回

- Burst Refresh：将 DRAM 整个全部刷新一遍
- Distributed Refresh：逐行刷新，避免内存长时间被阻塞
- Self Refresh：由 DRAM 自己刷新，关闭 controller，低能耗

refresh overhead：随着 DRAM 容量上升，用于刷新的时间会越来越多

而不同行的数据保留时间不同，故减少刷新次数的一个方法就是获得不同行的保留时间然后按其需要的频率进行刷新。可以按照保留时间分组，按固定频率刷新不同的组

- Profiling
  - 在行内写入数据
  - 停止刷新

    - 观察到数据失效要多久
- binning
    - 使用 Bloom Filter

> Bloom Filter：判断一个元素是否在集合中的一个概率模型，有三种操作
>
> - insert
> - test
> - remove all elements
>
> 用一个向量表示一个 set，insert 和 test 都是通过几个 hash function，insert 将得出的结果的几位置 1，test 只需检查对应位数是否都是 1，没有 false negative 因为只要插入了对应位数一定都是 1，会有 false positive
>
> 其优缺点为
>
> - 优点
>   - 减小存储集合成员关系的容量
>   - insert 和 test 快速
>   - 没有 false negative，即如果 bloom filter 说这个元素不在集合中，一定是真的
>   - 可以在时间，存储效率，false positive rate 之间做 tradeoff
> - 缺点
>   - false positive

对于刷新 DRAM 行来说 false positive 完全不是问题，只是某几行刷新地更频繁，而没有 false negative 保证了正确性，数据不会丢失。且 bloom filter 不会溢出

DRAM Scheduling Policies

DRAM 是共享资源，多个处理器访问时需要调度，一般来说调度是在 command 层面的，且行（read/write）的优先级高于列（activate/precharge）的

- FCFS (First Come First Served)：按请求到达顺序处理
- FR-FCFS (First Ready FCFS)：优先 row hit 以及最早到达的请求

一般调度是基于优先级，而优先级可以基于如下因素决定

- request age
- row buffer hit/miss rate
- request type (precharge/read/write)
- requestor type (load miss/ store miss)
- request criticality
- interference caused to other cores
- ...

row buffer 也有两种不同的管理方式

- open：在访问后保持 row open
- close：在访问后关闭 row

一般采用适应性的策略，即预测下一次访存是不是访问同一行

Prefetching

Latency

乱序执行的处理器对于长延时的指令容忍度更高，因为可以在延时时执行其他独立的指令

- 需要在 RS 和 ROB 中缓存指令
- 能缓存的指令数量即为 instruction window

而当延时过长时就需要 instruction window 非常大，延时过长的指令会阻塞住系统

- 阻塞了指令的 retire：为了精确中断
- instruction window 会填满
- window 满了就只能 stall 了

Memory Latency

主存的延时相对于 CPU 来说是很长的（即使减少了延时，仍然不在一个量级），一般的解决方法都是

- 减少甚至消除访问造成的 stall
- 如果发生 stall，减轻其影响

有四种基本方法

- caching：简单，但是低效且被动，不是所有程序都有局部性
- prefetching：对于规则的访问模式效果好
- multithreading：多个线程的情况下效果好
- OoO execution：可以容忍不规则的 cache miss，但是硬件成本昂贵

MLP (Memory-Level Parallelism) 的思想就是一次处理多个 cache miss 情况，这样处理器只用 stall 一次，即将不同 miss 的延时 overlap，而 OoO，multithreading，prefetching 都会产生复数的 cache miss

Prefetching

基本思路：在需要数据之前就将其拉入 cache

- 访存延时高，提前访存可以减少延时

- 能消除 compulsory cache miss

需要预测 prefetch 的地址，即要程序访存可预测，但是预测错了仅仅是不使用的数据，不需要恢复状态。现代系统中通常是通过 cache 每次都读入整个 block 实现的

prefetch 可以由硬件，编译器甚至程序员实现

需要考虑 3W1H

- What address to prefetch
- When to initiate a prefetch request
- Where to place the prefetched data
- How?

What address to prefetch

如果预取了不需要的数据会造成资源的浪费，包括内存带宽，cache 容量，以及能耗

可以根据过去的访存模式进行预测，也可以通过编译器对代码的分析实现

When to initiate a prefetch request

如果过早 prefetch，可能数据使用前就被换出

如果过晚 prefetch，可能仍有较大的延时

可以在硬件软件两方面考虑

- 硬件上提前于正常访存流
- 软件上直接将访存代码提前

Where to place the prefetched data

可以放在 cache

- 设计简单
- 可能会换出有用的数据

可以放在分离的 prefetch buffer

- 不影响 cache 中有用的数据
- 内存系统更复杂
    - prefetch buffer 放在哪
    - 和 cache 是串行访问还是并行访问
    - 何时将数据从 prefetch buffer 转移到 cache
    - 如何决定其大小
    - 如何确保一致性

大部分现代系统都是放在 cache 里。放在 L2？还是 L1？或是 L2 到 L1 还有一个 prefetcher

还要考虑放在 cache 的哪里，是当作正常 block 处理还是放在 LRU 位置

需要考虑 prefetcher 根据怎样的模式确定地址，L1 miss？L2 miss？

- 更多的访存请求带来更多精度
- 更多访存请求需要更多带宽

How

软件

- ISA 提供 prefetch 指令
- 程序员/编译器手动插入
- 对规则访问有效

硬件

- 硬件监控处理器的访存请求
- 从中寻找模式
- 预测生成 prefetch 地址

可以是基于一个“线程”来专门 prefetch

Software Prefetch

由程序员/编译器插入预取指令

- 预取的指令也占据了指令带宽
- 难以确定何时 prefetch，因为这与底层硬件有关
- 是否需要显式的 prefetch 指令
- 如果是基于指针的数据结构，难以 prefetch

何时预取也需要考虑，每个 load 都预取显然是不现实的，因此需要 profiling，插入时机不对会影响性能

Hardware Prefetch

由硬件观测访存模式并生成 prefetch

- 可以整合进系统实现
- 不需要额外指令
- 但需要复杂的结构来检测模式

最简单的方法就是每次访存后 prefetch 之后紧邻的 N block，实现简单但效果并不好（即使是规则访问，如果步长为 2 而 N = 1）

stride prefetcher 可以分为两种

- 基于 PC：记录 load 访问地址间的距离，以及上一个 load 访问的地址，如果遇到了相同的 load，预取上一个 load 地址 + 步长处的数据
  - 不会超前太多，fetch load 之后很快就会访问 cache
  - 需要初始化，然后下一次访问同样的 load 才能预测
  - 可以使用向前看的 PC，提前 prefetch
- 基于 cache block address：记录 cache block 的地址和步长
  - stream buffer：步长为 1 的特殊情况，每个 stream buffer 缓存 cache line 之后连续的 block，当 cache miss 后检查各个 stream buffer 的头，如果 hit 则将数据放入 cache，否则分配一个新的 stream buffer，可能需要基于 LRU 替换掉一个 stream buffer

Performance

衡量 prefetcher 的性能有多种指标

- accuracy: used / sent
- coverage: prefetched / all miss
- timeliness: on-time / used
- bandwidth consumption
- cache pollution: extra miss due to prefetch

激进的 prefetcher 会有较好的时间表现，较高的覆盖率，但是准确度低，带宽需求高

保守的 prefetcher 相反，精度较高且带宽需求低，对 cache 影响小，但时间表现和覆盖率不好

Memory Management

一般会分为正交的几个功能

- translation：虚地址到物理地址的映射
- protection：访问权限控制
- virtual memory：使用外存扩展内存空间

但一般现代系统用一个单独的基于页的内存管理系统实现上述功能

Bare machine

在裸机中只有物理地址

最早机器中同时只能运行一道程序，程序可以无限制地访问所有内存和 IO 设备

后来库开始流行，其中的代码是位置无关的，需要 linker 或 loader 将库模块分配到实际的地址

随着软件的发展，推动了动态地址转换的发展

- 多道程序以及 DMA 来管理 IO 设备提升 IO 性能
- 位置无关代码易于管理，需要一个基地址寄存器
- 各自独立的程序不能影响对方，需要边界寄存器
- 多程序需要一个管理的程序（OS）来维护上下文切换

Simple Base and Bound Translation

程序地址空间通过 base register 转换，通过 bound register 控制访问

只有在 CPU 内核态才能访问这两个寄存器

之后将程序的代码和数据分开，使用各自独立的 base register 和 bound register，便于管理（如代码段显然是只读的）

Paged Memory System

将地址分为 page number 和 offset

页表将虚拟页映射到物理页

页的出现使得可以用非连续的物理内存实现连续的虚拟内存

页也使得分配变得简单，因为大小是固定的，程序动态使用的内存可以轻松地增长或缩小

但是会有内碎片的问题（相比外碎片来说不是问题）

扁平的页表在内存中占据了过多的空间，故使用分级页表减小内存消耗

Address Translation and Protection

地址转换需要快速，因为每条涉及内存的指令都要转换

TLB (Translation-Lookaside Buffer) 减少多级页表中对内存的访问次数

- 如果 hit 则单个周期就能解决
- 如果 miss 则访问内存并填写 TLB

TLB 一般都是 fully associative 的（页面大，因此页面一般不连续，如数据区，内核栈，用户栈地址往往差很多，采用其他方式冲突几率高），替换策略采用随机或是 FIFO

TLB miss 可以由软件或硬件解决

- 软件：MIPS，Alpha
  - TLB miss 产生异常，OS 查询页表并填写 TLB
  - 在 OoO 处理器上会格外昂贵，因为需要冲刷流水线，转到处理程序
- 硬件：SPARC，x86，RISC-V
  - MMU 查询填写 TLB
  - 如果发现缺页（无论是数据还是内存中页表），MMU 直接产生 page fault

Page Fault

如果发生缺页

- 分配缺失的页
- 从磁盘中读取，然后更新页表
- CPU 可以调度其他进程
- 如果没有可用物理页，选择一个页替换出去
- 一般使用软件实现的伪 LRU 替换算法

page fault 处理时间长，一般完全由 OS 进行软件层的处理，处理时需要实地址模式

需要注意 TLB 和页表的一致性