高等计算机体系结构
第十三讲: 隐藏访存延迟/虚拟存储

栾钟治
北京航空航天大学 计算机学院 中德联合软件研究所
2021-05-21

1

提醒：作业

- 作业 5
  - 已发布，5月28日上课前截止提交
  - Cache和Memory
- 作业 6
  - 5月28日发布，6月11日截止
  - 预取和并行

2

实验2-5

- 近期发布，预计7月11日截止

3

阅读材料

- 分层存储体系结构

- Patterson & Hennessy's *Computer Organization and Design: The Hardware/Software Interface* （计算机组成与设计：软硬件接口）
  - 第五章：5.1-5.3, 5.4
- Maurice Wilkes早期关于cache的论文
  - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
- 推荐阅读
  - Denning, P. J. *Virtual Memory*. ACM Computing Surveys. 1970
  - Jacob, B., & Mudge, T. *Virtual Memory in Contemporary Microprocessors*. IEEE Micro. 1998

4

回顾：交叉存取

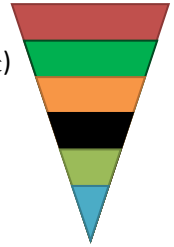
- 问题: 单片的存储阵列访问时间很长，并且无法并行执行多个访存
- 目标: 减小对存储阵列访问的延迟，并且能够并行执行多个访存
- 思路: 将存储阵列划分为多个可以(在同一个周期或连续的周期)独立访问的Bank
  - 每个 Bank 都比整个存储空间小
  - 可以重叠地访问不同的 Bank
- 需要解决的难题: 如何将数据映射到不同的 Bank? (如何在不同的Bank之间交叉存取数据?)

5

5

回顾：DRAM 子系统的组织

- 通道
- DIMM(双列直插式存储模块)
- Rank
- 芯片
- Bank
- 行/列



6

6

回顾：延迟组件

- CPU → 控制器的传输时间
- 控制器延迟
  - 控制器中排队和调度的延迟
  - 访存被转换为基本命令
- 控制器 → DRAM的传输时间
- DRAM Bank 延迟
  - 如果行已经打开，则简单的列选通，或者
  - 如果阵列已经预充电则行选通 + 列选通，或者
  - 预充电 + 行选通 + 列选通 (最坏的情况)
- DRAM → CPU 的传输时间 (通过控制器)

7

7

回顾：DRAM 刷新

- 刷新对性能的影响
- 集中式刷新: 所有行在前一行刷新完成后立即刷新
- 分散式刷新: 每存储周期刷新一行
- 分布式刷新: 每一行按照固定的间隔在不同时间刷新

8

8

回顾：减少刷新操作

- 思路: 确定不同行的保持时间，根据每行需要的刷新频率刷新每一行
- (注重成本的) 思路: 根据最小保持时间把行分组，再按照每组特定的刷新频率对组内行进行刷新
  - 比如, 64-128ms刷新的组, 128-256ms刷新的组, …
- 观察: 只有很少的行需要很高刷新频率 [64-128ms] → 只有很少的几组→ 用低的硬件开销实现刷新操作的大幅度减小
- Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

回顾：DRAM 控制器

- 确保DRAM操作正确(刷新和时序)
- 在遵循DRAM芯片的时序约束下响应DRAM的请求
  - 约束: 资源冲突 (Bank, 总线, 通道), 最小的写-读延迟
  - 将请求翻译成DRAM命令序列
- 缓冲和调度请求以提升性能
  - 重排序, 行缓冲, Bank/Rank/总线管理
- 管理DRAM的功耗和发热
  - 开/关DRAM芯片, 管理功率模式

回顾：DRAM 调度策略

- FCFS (先来先服务)
  - 最旧的请求最优先
- FR-FCFS (行缓冲优先)
  1. 行命中的优先
  2. 最旧的优先

  目的: 最大化行缓冲命中率 → 最大化DRAM吞吐量

  调度策略实际上是优先级的序

行缓冲管理策略

- 打开行
  - 一次访问之后保持该行打开
  + 下一次访问可能是同一行 → 行命中
  -- 下一次访问可能是不同的行 → 行冲突, 消耗能量
- 关闭行
  - 一次访问后关闭该行 (如果在请求缓冲中没有其它请求访问同一行)
  + 下一次访问可能是不同的行 → 避免一次行冲突
  -- 下一次访问可能是同一行 → 额外的激活延迟
- 自适应策略
  - 预测下一次对Bank的访问是否是同一行

打开 vs. 关闭行策略

策略	第一次访问	下一次访问	下一次访问需要的命令
打开行	行 0	行 0 (行命中)	读
打开行	行 0	行 1 (行冲突)	预充电 + 激活行 1 + 读
关闭行	行 0	行 0 – 在请求缓冲中 (行命中)	读
关闭行	行 0	行 0 – 不在请求缓冲中 (行关闭)	激活行 0 + 读 + 预充电
关闭行	行 0	行 1 (行关闭)	激活行1 + 读 + 预充电

13

为什么DRAM控制器很难设计?

- 需要遵从DRAM时序约束以保证正确性
  - DRAM中有很多时序约束 (50+)
  - tWTR: 写命令发射后发射读命令需要等待的最小周期数
  - tRC: 对于同一个Bank连续发射两条激活命令之间需要等待的最小周期数
  - ...
- 需要持续监视各种资源以防止冲突
  - 通道, Bank, Rank, 数据总线, 地址总线, 行缓冲
- 需要处理DRAM刷新
- 需要优化性能 (多约束条件下)
  - 重排序并不简单
  - 预测未来?

14

DRAM 时序约束

Latency	Symbol	DRAM cycles	Latency	Symbol	DRAM cycles
Precharge	^tRP	11	Activate to read/write	^tRCD	11
Read column address strobe	CL	11	Write column address strobe	CWL	8
Additive	AL	0	Activate to activate	^tRC	39
Activate to precharge	^tRAS	28	Read to precharge	^tRTP	6
Burst length	^tBL	4	Column address strobe to column address strobe	^tCCD	4
Activate to activate (different bank)	^tRRD	6	Four activate windows	^tFAW	24
Write to read	^tWTR	6	Write recovery	^tWR	12

Table 4. DDR3 1600 DRAM timing specifications

- From Lee et al., "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," HPS Technical Report, April 2010.

15

更多关于DRAM操作

- Kim et al., "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," ISCA 2012.
- Lee et al., "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," HPCA 2013.

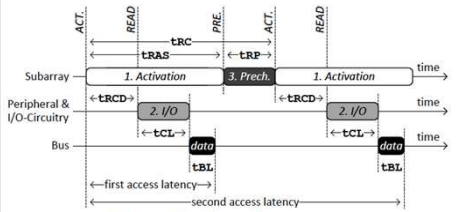


Figure 5. Three Phases of DRAM Access

Table 2. Timing Constraints (DDR3-1066) [43]

Phase	Commands	Name	Value
1	ACT → READ ACT → WRITE	tRCD	15ns
	ACT → PRE	tRAS	37.5ns
2	READ → *data* WRITE → *data*	tCL tCWL	15ns 11.25ns
	data burst	tBL	7.5ns
3	PRE → ACT	tRP	15ns
1 & 3	ACT → ACT	tRC (tRAS+tRP)	52.5ns

16

DRAM 电源管理

- DRAM芯片有多种电源管理模式
- 思路: 当某个芯片没有访问时进入节电模式

- 功率状态
  - 活跃 (最高功率)
  - 所有Bank空闲
  - 节电模式
  - 自刷新 (最低功率)

- 状态转换会产生延迟，在该延迟内芯片无法访问

17

17

新型的非易失性存储技术

18

非易失存储器

- 如果存储器是非易失的...
  - 不需要刷新...
  - 不会在掉电时丢失数据...

- 问题: 非易失存储器件一直以来都比DRAM慢很多
  - 比如硬盘... 甚至闪存...

- 机遇: 一些新兴的存储技术，非易失而且相对比较快
  - 同时，比DRAM可扩展性更好

- 提问: 是否可以采用这些新兴技术来实现主存储器？

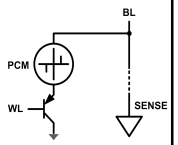
19

19

新兴的存储技术

- 一些新兴的电阻式存储技术似乎比DRAM具有更好的可扩展性 (并且它们是非易失的)

- 例如: 相变存储器
  - 通过材料的相变存储数据
  - 通过检测材料的阻抗读取数据
  - 预计尺寸可以达到9nm (2022 [ITRS])
  - 原型20nm (Raoux+, IBM JRD 2008)
  - 将比DRAM密度更高: 可存储多个bit/位元



- 当然, 新的技术会有一些缺陷
  - 它们能够代替DRAM吗?

20

20

电阻式存储器技术

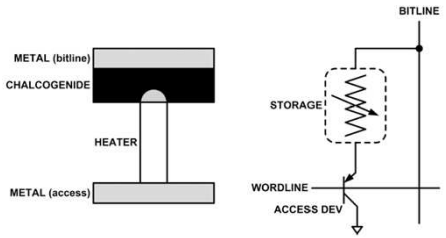
- PCM(相变存储器)
  - 通过注入电流使材料发生相变
  - 相变决定阻抗的不同
- STT-MRAM(自旋转矩磁随机存取存储器)
  - 通过注入电流改变磁极
  - 极性改变决定阻抗的不同
- Memristor(忆阻器)
  - 通过注入电流改变原子结构
  - 原子间的距离决定阻抗

21

21

什么是相变存储器?

- 相变材料(硫族化合物玻璃) 存在于两种状态中:
  - 非晶态: 低光反射率，高电阻率
  - 晶态: 高光反射率，低电阻率



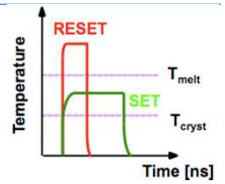
PCM是电阻式存储器: 高阻态 (0), 低阻态 (1)
PCM的位元可以在不同状态之间可靠、快速地切换

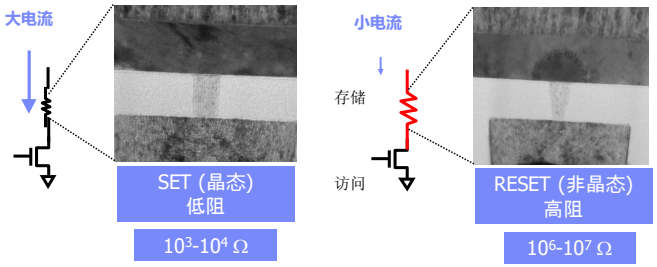
22

22

PCM 如何工作?

- 写: 通过电流注入改变物相
  - SET: 持续注入电流加热位元温度超过 T*cryst*
  - RESET: 位元加温超过T*melt* 并迅速冷却
- 读: 通过材料的阻抗判断物相
  - 非晶/晶态





23

23

相变存储器: 优点和缺点

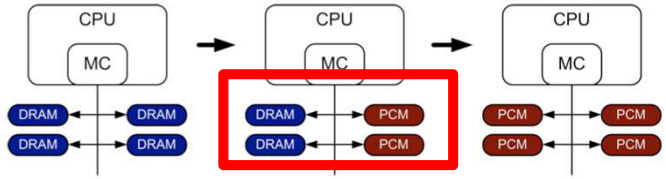
- 优于DRAM之处
  - 更好的工艺规模(容量和成本)
  - 非易失
  - 空闲时功率低 (无需刷新)
- 缺点
  - 延迟更高: ~4-15x DRAM (尤其是写入时)
  - 活跃状态能耗更高: ~2-50x DRAM (尤其是写入时)
  - 重复使用寿命较低 (位元寿命~10^8 次写入)
- 用PCM替换或者协助DRAM组成主存储器的挑战:
  - 减小PCM缺陷的影响
  - 找到合适的方式将PCM引入系统

24

24

基于PCM的主存储器 (I)

- 基于PCM的(主)存储器如何组织?



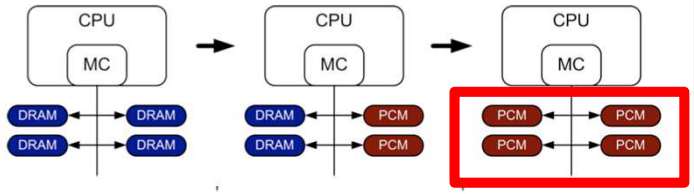
- 混合PCM+DRAM [Qureshi+ ISCA'09, Dhiman+ DAC'09]:
  - 在PCM和DRAM之间如何分区/迁移数据

25

25

基于PCM的主存储器(II)

- 基于PCM的(主)存储器如何组织?



- 纯 PCM的主存储器 [Lee et al., ISCA'09, Top Picks'10]:
  - 如何重新设计整个层次结构(包括核)以克服PCM的缺点

26

26

基于PCM的存储系统: 研究上的挑战

- 分区
  - 可以用DRAM做cache或者主存，或者灵活配置吗?
  - 各占多少比例? 需要多少控制器?
- 数据分配/移动 (能耗, 性能, 生命周期)
  - 谁来管理分配和移动?
  - 控制算法?
  - 如何预防因为材料老化导致的失效?
- Cache的层次设计, 内存控制器, OS
  - 消除PCM缺陷的影响, 利用PCM的优点
- PCM/DRAM芯片和模块设计
  - 重新考虑PCM/DRAM的新需求

27

27

初步的研究: 用PCM替换DRAM

- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.
  - 综述了2003-2008的原型系统 (比如 IEDM, VLSI, ISSCC)
  - 得出PCM"平均"F=90nm

Density
- ▷ 9 - 12F^2 using BJT
- ▷ 1.5× DRAM

Latency
- ▷ 50ns Rd, 150ns Wr
- ▷ 4×, 12× DRAM

Endurance
- ▷ 1E+08 writes
- ▷ 1E-08× DRAM

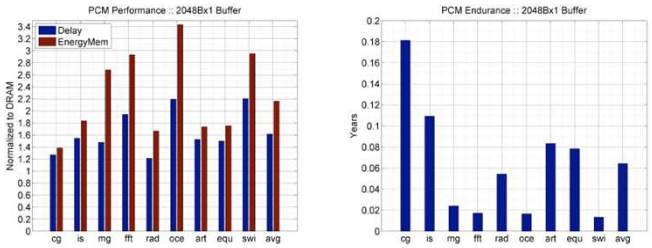
Energy
- ▷ 40μA Rd, 150μA Wr
- ▷ 2×, 43× DRAM

28

28

结果: 用PCM简单替换DRAM

- 在4核、4MB L2 cache的系统中用PCM替换DRAM
- PCM的组织与DRAM相同: 行缓冲, Bank…
- 1.6x延迟, 2.2x能耗, 500小时平均寿命



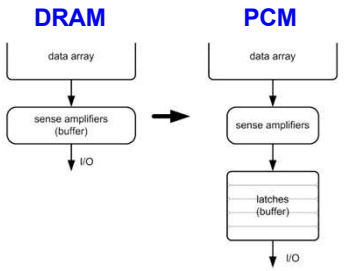
- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

29

29

设计PCM架构以减小缺陷的影响

- 思路1: 在每个PCM芯片中使用多个窄行缓冲
  → 减少阵列的读/写 → 更好的耐久性、延迟和能耗表现
- 思路2: 写入阵列时按cache line或字的粒度写
  → 减少不必要的损耗

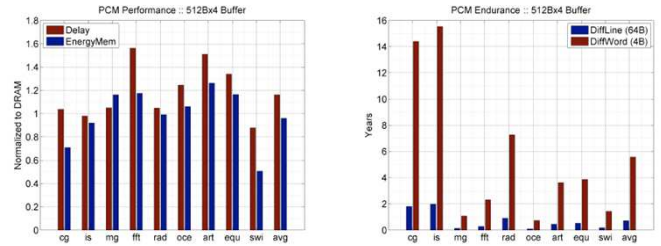


30

30

结果: 用PCM构建主存储器

- 1.2x延迟, 1.0x能耗, 5.6-year平均寿命
- 调整规模可以改善能耗、耐久性和密度等指标

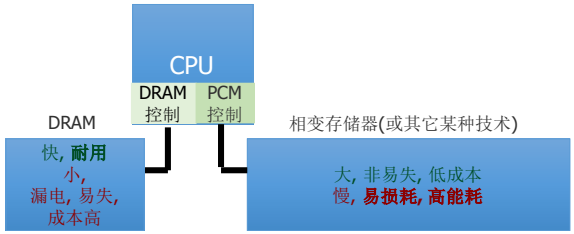


- 注1: 最坏情况下寿命很短 (无保障)
- 注2: 密集型应用的性能和能耗都会受到很大的影响
- 注3: 如何优化PCM的参数?

31

31

混合存储系统



硬件/软件管理数据分配和移动以获得多种技术的优势

Meza+, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

Yoon, Meza et al., "Row Buffer Locality Aware Caching Policies for Hybrid Memories," ICCD 2012 Best Paper Award.

32

一种选择: DRAM作为PCM的cache

- PCM作主存; DRAM缓存主存的行/块
  - 好处: DRAM缓存命中时减小延迟; 写过滤
- 内存控制器硬件管理DRAM cache
  - 好处: 没有系统软件的开销
- 三个需要解决的问题:
  - 哪些数据应该放到DRAM中?
  - 数据移动的粒度该如何选择?
  - 如何设计低成本的硬件管理的DRAM cache?
- 两个思路:
  - 感知局部性的数据放置[Yoon+ , ICCD 2012]
  - 低开销标签存储和动态粒度 [Meza+, IEEE CAL 2012]

33

访存延迟容忍

34

延迟容忍

- 乱序执行处理器通过并发执行独立的指令容忍多周期操作的延迟
  - 通过在保留站和重排序缓冲中缓冲指令来实现
  - 指令窗口: 需要硬件资源来缓冲所有已经译码但尚未提交/回收的指令
- 如果一条指令要花费500的时钟周期该怎么办?
  - 需要多大的指令窗口才能持续译码?
  - 乱序执行能够容忍多少周期的延迟?

35

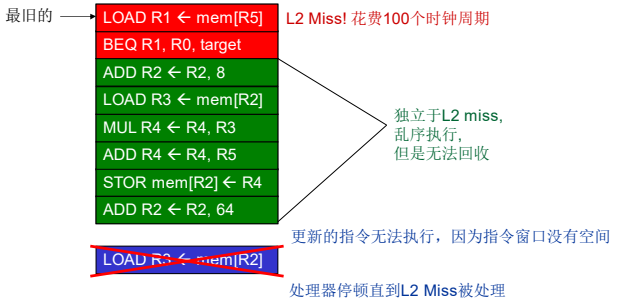
由长延迟指令导致的停顿

- 当一条长延迟指令没有结束, 它会阻碍指令回收
  - 因为需要保证精确异常
- 输入的指令填满指令窗口(重排序缓冲, 保留站)
- 一旦窗口填满, 处理器无法继续向窗口中放入新的指令
  - 称为满窗口停顿
- 满窗口停顿会阻止处理器向前推进正在执行的程序

36

满窗口停顿

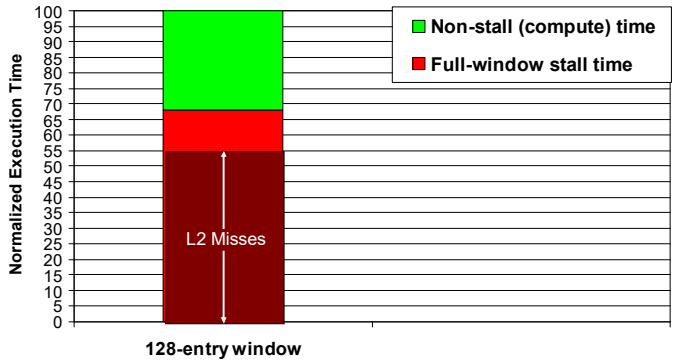
8-路指令窗口:



- L2 cache的缺失是导致满窗口停顿的最主要原因

37

Cache缺失引起很多的停顿



512KB L2 cache, 500个周期的DRAM延迟, 激进的基于流的预取
147种内存密集型benchmark在高端x86处理器上的实验数据的平均结果

38

如何容忍内存导致的停顿?

- 两种主要方法
  - 减少/消除停顿
  - 当停顿发生时容忍它的影响
- 四种基本技术
  - 高速缓存
  - 预取
  - 多线程
  - 乱序执行
- 有很多技术使这四种基本技术在容忍存储延迟时更加有效

39

内存延迟容忍技术

- 高速缓存 [最早提出Wilkes, 1965]
  - 使用最广泛, 简单, 有效, 但是效率不高, 被动
  - 不是所有的应用都表现出时间或者空间的局部性
- 预取 [最早提出 IBM 360/91, 1967]
  - 适用于普通的内存访问模式
  - 预取不规则的访存模式很困难, 不准确, 硬件开销大
- 多线程 [最早提出 CDC 6600, 1964]
  - 适用于多线程场景
  - 如何利用多线程的硬件提升单个线程的性能，仍需要研究
- 乱序执行 [最早提出 Tomasulo, 1967]
  - 容忍因无法预取而导致的不规则cache缺失
  - 需要大量的资源来容忍长的延迟

40

预取

41

预取

- 思路: 在程序需要使用之前取数据
- 为什么?
  - 访存延迟高, 如果可以足够早并且准确地预取将减小/消除延迟
  - 可以消除cache的强制缺失
  - 是否能消除所有的cache缺失? 容量, 冲突?
- 包括预测哪个地址会是未来需要的
  - 如果程序具有缺失地址的可预测模式

42

预取和正确性

- 预取时的错误预测是否会影响正确性?
- 不会, 从"预测错误"的地址预取的数据不会被用到
- 不需要做状态恢复
- 对比分支预测错误或者值预测错误

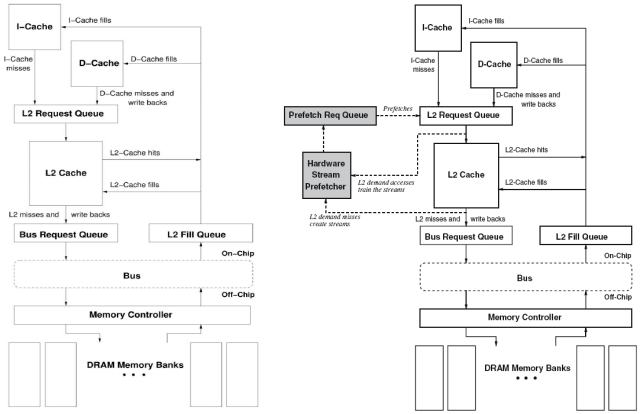
43

基础

- 现代系统中, 预取通常在cache块的粒度上实现
- 预取技术可以减小
  - 缺失率
  - 缺失延迟
- 预取可以在以下层面实现
  - 硬件
  - 编译器
  - 程序员

44

在存储系统如何加入硬件预取器



45

45

预取: 四个问题

- What
  - 预取什么地址
- When
  - 何时发起预取请求
- Where
  - 预取的数据放到哪儿
- How
  - 软件、硬件、基于执行、合作

46

46

预取的挑战: What

- 预取什么地址
  - 预取无用的数据会浪费资源
    - 存储带宽
    - Cache或预取缓冲的空间
    - 能耗
    - 可供有需要的请求或更准确的预取请求使用
  - 预取地址的准确预测是很重要的
    - 预取精度 = 有用的预取/ 发出的预取
- 我们怎么知道预取什么
  - 基于过去访问模式的预测
  - 利用编译器关于数据结构的知识
- 预取算法决定预取什么

47

47

预取的挑战: When

- 何时发起预取请求
  - 预取太早
    - 预取的数据可能在没被使用之前就被踢出暂存空间
  - 预取太迟
    - 可能无法隐藏全部的访存延迟
- 数据被预取的时机影响一个预取器的及时性指标
- 预取器可以具有更好的及时性
  - 更加的激进: 尽量保持领先处理器访问流的幅度 (硬件)
  - 在代码中更早的发起预取指令 (软件)

48

48

预取的挑战: Where (I)

- 预取的数据放到哪儿
  - 放到cache里
    - + 设计简单, 不需要单独的缓冲
    - -- 可能会将有用的数据踢出 → cache污染
  - 放到独立的预取缓冲中
    - + 有用的数据被保护起来，不受预取影响 → 没有cache污染
    - -- 存储系统设计更加复杂
      - 预取缓冲放在哪儿
      - 何时访问预取缓冲 (与cache访问并行还是串行)
      - 何时将数据从预取缓冲移动到cache
      - 如何规划预取缓冲的大小
      - 保持预取缓冲的一致性
- 很多现代系统将预取数据放入cache
  - Intel Pentium 4, Core2, AMD, IBM POWER4,5,6, …

49

49

预取的挑战: Where (II)

- 预取到哪个级别的cache?
  - 从内存到L2, 从内存到L1, 优点/缺点?
  - 从L2 到 L1? (在cache不同层次之间增加独立的预取器)
- 在cache里把预取的数据放到哪儿?
  - 预取的块和按需取的块同样对待吗?
  - 预取的块不知道是不是需要的
    - 采用 LRU策略时, 将按需取的块放置在MRU位置
- 需要调整替换策略以使它能够更优待按需取的块吗?
  - 比如, 将所有的预取块按某种方式放置在LRU位置?

50

50

预取的挑战: Where (III)

- 硬件预取器应该放在分层存储结构的什么位置?
  - 换句话说, 预取器应该看到什么样的访问模式?
  - L1 hit 和 miss
  - L1 miss
  - L2 miss
- 看到更复杂的访问模式:
  - + 可能有更好的预取精度和覆盖率
  - -- 预取器需要检验更多的请求 (带宽密集, 更多输入端口的预取器?)

51

51

预取的挑战: How

- 软件预取
  - ISA 提供预取指令
  - 程序员或编译器插入预取指令
  - 通常只对"常规的访问模式"有效
- 硬件预取
  - 硬件监控处理器的存取
  - 记录或者发现模式
  - 自动生成预取地址
- 基于执行的预取器
  - 执行一个"线程"为主程序预取数据
  - 可以通过软件/程序员或者硬件生成

52

52

软件预取(I)

- 思路: 编译器/程序员将预取指令插入代码中合适的位置

- Mowry et al., "Design and Evaluation of a Compiler Algorithm for Prefetching," ASPLOS 1992.

- 预取指令将数据预取放入cache
- 编译器或程序员能够向程序中插入这样的指令

53

X86 预取指令

PREFETCHh—Prefetch Data Into Caches

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 18 /1	PREFETCHT0 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T0 hint.
0F 18 /2	PREFETCHT1 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T1 hint.
0F 18 /3	PREFETCHT2 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T2 hint.
0F 18 /0	PREFETCHNTA *m8*	Valid	Valid	Move data from *m8* closer to the processor using NTA hint.

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
  - Pentium III processor—1st- or 2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.

NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.

- Pentium III processor—1st-level cache
- Pentium 4 and Intel Xeon processors—2nd-level cache

微体系结构相关的规范

不同的指令对应不同的cache级别

54

软件预取(II)

```
for (i=0; i<N; i++) {
  __prefetch(a[i+8]);
  __prefetch(b[i+8]);
  sum += a[i]*b[i];
}
```

```
while (p) {
  __prefetch(p→next);
  work(p→data);
  p = p→next;
}
```

```
while (p) {
  __prefetch(p→next→next→next);
  work(p→data);
  p = p→next;
}
```

哪一个更好?

- 适用非常规则的基于类似数组结构的访问模式，存在的问题:
  - -- 预取指令占用处理/执行带宽
  - 该多早开始预取? 很难决定
    - -- 预取距离依赖于硬件实现 (存储延迟, cache大小, 循环迭代之间的时间)
    - -- 在代码中回退太远会降低精度 (当中间有分支时尤其如此)
  - 需要ISA设置"特殊的"预取指令?
    - 并非如此. Alpha架构中向31号寄存器 load 被看作是预取 (r31==0)
    - PowerPC *dcbt* (data cache block touch) 指令
  - -- 对付基于指针的数据结构不太容易

55

软件预取(III)

- 编译器会往哪儿插入预取指令?
  - 预取每一个load访问?
    - 过度的带宽密集 (包括内存和执行带宽)
  - 分析代码并确定可能会缺失的load
    - 如果分析的输入集没有代表性会怎么样?
  - 预取应该在缺失之前多远的地方插入?
    - 分析并确定使用不同预取距离的可能性
      - 如果分析的输入集没有代表性会怎么样?
      - 通常需要插入的预取能覆盖100个时钟周期的主存延迟 → 降低精度

56

硬件预取(I)

- 思路: 采用特殊的硬件观察load/store的访问模式，基于过去的访问行为预取数据

- Tradeoff:
  - + 可以协调地成为系统实现的一部分
  - + 不会浪费指令执行带宽
  - -- 为了检测模式会使硬件更加复杂
  - - 在某些情况下软件可能更有效率

57

57

Next-line预取器

- 硬件预取最简单的形式: 总是预取一个按需访问(缺失)之后的N个cache行
  - Next-line 预取器 (也叫紧邻顺序预取器)
  - Tradeoff:
    - + 实现简单，无需复杂的模式检测
    - + 适用于顺序/流访问模式
    - -- 对于非规则模式会浪费带宽
    - -- 即使是规则的模式:
      - -如果访问的跨度是2，N=1，预取的精度如何?
      - - 程序从高地址向低地址遍历内存会怎么样?
      - - 预取“之前的”N个 cache 行?

58

58

跨度(步长)预取器

- 两种
  - 基于指令指针/程序计数器
  - 基于Cache块地址

- 基于指令:
  - Baer and Chen, “An effective on-chip preloading scheme to reduce data access penalty,” SC 1991.
  - 思路:
    - 记录一条load指令引用内存地址的距离(即load的跨度)以及该load指令引用的最后一个地址
    - 下一次取这条load指令时，预取上次引用的最后一个地址+跨度

59

59

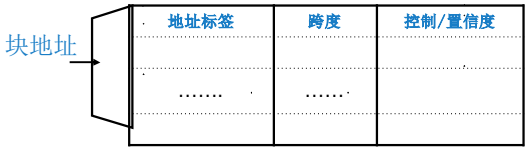
基于指令的跨度预取

- 有什么问题?
  - 提示: 预取可以提前多少? 预取能覆盖多大的缺失延迟?
  - Load再一次被取指时才启动预取就太迟了
    - Load被取指之后很快就会访问cache!
  - 解决方案:
    - 用预读PC索引预取器表
    - 提前预取 (最后地址 + N*跨度)
    - 生成多个预取

60

60

基于cache块地址的跨度预取



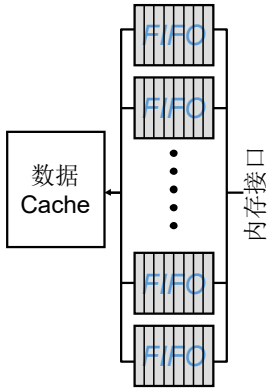
- 能够检测
  - A, A+N, A+2N, A+3N, …
  - 流缓冲是基于cache块地址的跨度预取的一个特例，N = 1
    - Jouppi的论文

61

61

流缓冲(Jouppi, ISCA 1990)

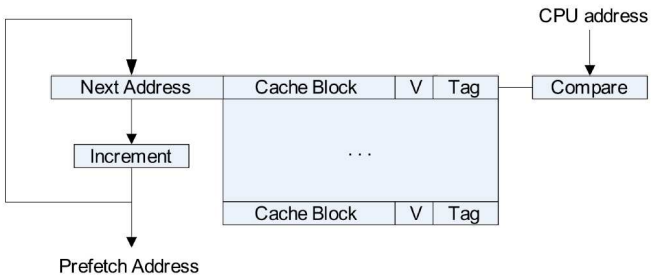
- 每个流缓冲保持一个顺序预取的cache行的流
- 当发生load缺失时，检查所有流缓冲头部是否有地址匹配
  - 如果命中, 从FIFO队列中弹出，更新cache中的数据
  - 如果不命中, 向新的缺失地址分配一个新的流缓冲(可能需要按照LRU策略回收一个流缓冲)
- 只要有空间并且总线不忙，流缓冲的FIFO队列会持续不断地放入后续的cache行



62

62

流缓冲设计



63

63

预取器性能(I)

- 精度 (有用的预取 / 发出的预取)
- 覆盖率 (预取的缺失 / 所有的缺失)
- 及时性 (准时的预取 / 有用的预取)

- 带宽消耗
  - 有/没有预取器时，存储带宽的消耗
  - 好消息: 可以利用空闲时的总线带宽

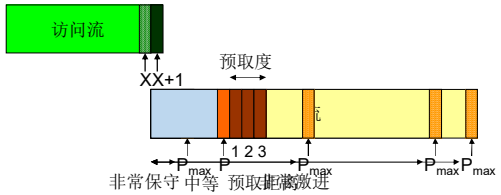
- Cache污染
  - 由于预取放在cache中导致的额外的按需访问缺失
  - 很难量化，但是会影响性能

64

64

预取器性能(II)

- 预取器的激进特性影响所有的性能指标
- 激进程度取决于预取器的类型
- 对于大多数硬件预取器:
  - 预取距离: 领先访问需求流的距离
  - 预取度: 每个按需访问预取的数量



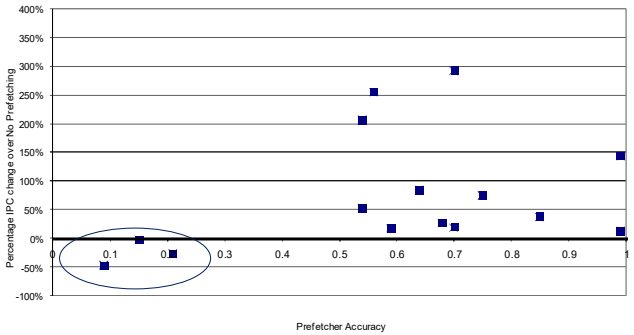
65

预取器性能(III)

- 这些指标如何相互影响?
- 非常激进
  - 大幅领先load访问流
  - 更好的隐藏访存延迟
  - 更多的投机
  - \+ 更高的覆盖率, 更好的及时性
  - -- 很可能精度较低, 带宽消耗较高, cache污染也较高
- 非常保守
  - 接近load访问流
  - 可能无法完全覆盖访存延迟
  - 减小可能的cache污染和带宽竞争
  - \+ 可能有较高的精度, 较低的带宽消耗, 较少的污染
  - -- 可能覆盖率较低，不够及时

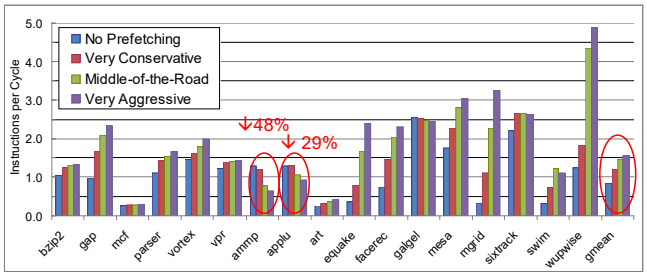
66

预取器性能(IV)



67

预取器性能(V)

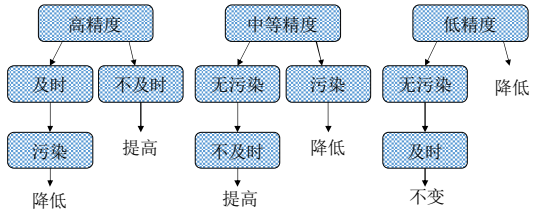


- Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers", HPCA 2007.

68

基于反馈的预取器调节(I)

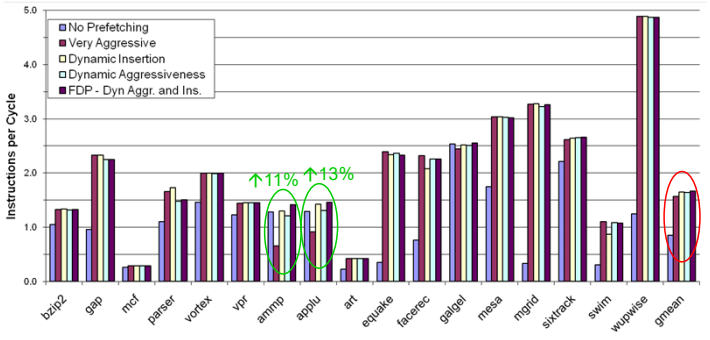
- 思路:
  - 动态监控预取器性能指标
  - 基于过去的性能状况调节预取器的激进程度
  - 基于过去的性能状况改变预取插入cache的位置



69

69

基于反馈的预取器调节(II)



- Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers", HPCA 2007.

70

70

如何预取不规则访问模式?

- 规则模式: 跨度, 流预取器
- 不规则访问模式
  - 间接数组访问
  - 链式数据结构
  - 多跨度(1,2,3,1,2,3,1,2,3,...)
  - 随机模式?
  - 针对所有模式的通用预取器?
- 基于相关性的预取器
- 基于内容的预取器
- 基于预计算或预执行的预取器

71

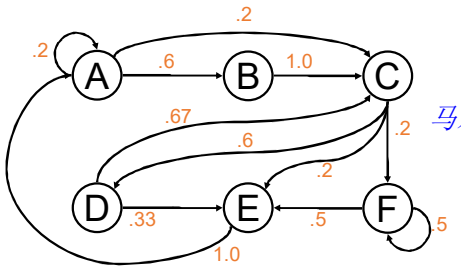
71

马尔科夫预取(I)

- 考察下列cache块地址访问的历史

  A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C

- 在引用某个特定地址 (比如 A 或 E)之后, 确实有某些地址看起来更有可能在接下来被引用



马尔科夫模型

72

72

马尔科夫预取(II)

- 思路:当看到地址A时记录可能的后续地址 (B, C, D)
  - 下一次当A被访问, 预取B, C, D
  - A被称作与B, C, D相关
- 预取*精度*通常比较低，所以预取N个后续地址以增加*覆盖率*
- 预取精度可以通过使用多个地址作为下一个地址键值的方法来改善: (A, B) → (C)

  (A,B) 与 C 相关

- Joseph and Grunwald, "Prefetching using Markov Predictors," ISCA 1997.

73

73

马尔科夫预取(III)

- 优点:
  - 可以应对任意访问模式
    - 链式数据结构
    - 流模式 (虽然不那么高效!)
- 缺点:
  - 需要一张很大的相关表以获得高覆盖率
    - 记录每个缺失地址及其后续的缺失地址是不可行的
  - 及时性低: 预读是受限的，因为对下一个访问/缺失的预取是在前一个之后开始的
  - 消耗很多的存储带宽
    - 特别是当马尔科夫模型概率 (相关性) 低的时候
  - 无法减少强制缺失

74

74

基于内容的预取(I)

- 一种针对指针值的特殊预取器
- Cooksey et al., "A stateless, content-directed data prefetching mechanism," ASPLOS 2002.
- 思路: 识别取回的cache块中的指针，发射针对它们的预取请求

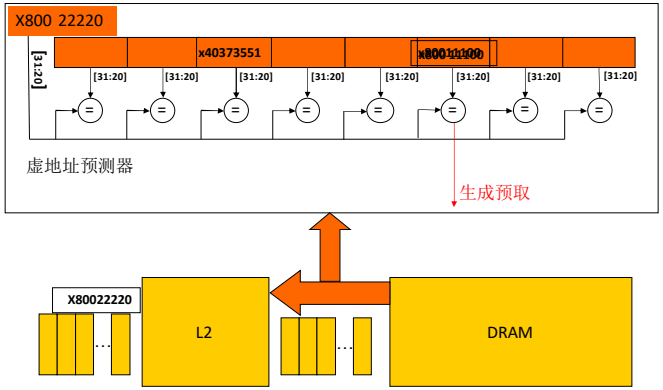
+ 无需记忆过去的地址!
+ 可以消除强制缺失 (从未见过的指针)
-- 不加选择地预取cache块中的所有指针

- 如何识别指针地址:
  - 按地址的尺寸比较cache块中的数据和cache块的地址 → 如果最高几位匹配，就是指针

75

75

基于内容的预取(II)



76

76

使基于内容的预取器更有效

- 硬件没有足够的关于指针的信息
- 软件有 (而且可以通过分析得到更多的信息)

- 思路:
  - 编译器分析并提供预取哪些指针地址可能有用的提示
  - 硬件使用这些提示仅仅预取可能有用的指针

- Ebrahimi et al., "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," HPCA 2009.

77

77

基于执行的预取器(I)

- 思路: 专门为预取数据而预执行程序(修剪)的一个片段
  - 只需要提取会导致cache缺失的片段

  - 投机线程: 预执行的程序片段可以被看作是一个"线程"

  - 投机线程可以执行在
- 独立的处理器/核
- 独立的硬件线程上下文 (回忆细粒度多线程)
- 相同线程上下文的空闲周期 (在cache缺失期间)

78

78

基于执行的预取器(II)

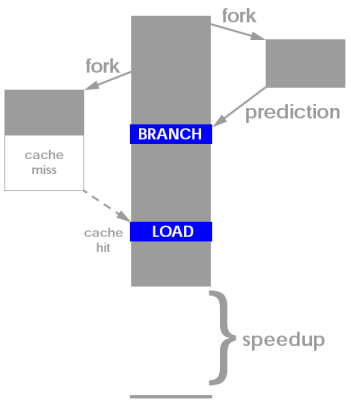
- 如何构建投机线程
  - 基于软件的修剪和指令生成
  - 基于硬件的修剪和指令生成
  - 使用原始程序 (而非重新构建), 但是
    - 不受停顿和正确性约束的快速执行

- 投机线程
  - 需要比主程序更早发现缺失
    - 避免等待/停顿，使计算效率下降
  - 为了达到这一目标
    - 仅执行地址生成计算、分支预测和值预测

79

79

基于线程的预执行



- Dubois and Song, "Assisted Execution," USC Tech Report 1998.

- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.

- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

80

80

基于线程的预执行需要解决的问题

- 在哪里执行预计算的线程?
  1. 独立的核 (与主线程竞争最小)
  2. 在同一个核上独立的线程上下文 (更多竞争)
  3. 相同的核, 相同的上下文
     - 当主线程停顿时
- 何时生成预计算线程?
  1. 在“问题”load之前插入生成的指令
     - 多远之前?
       - 太早: 可能还不需要预取
       - 太迟: 预取可能不及时
  2. 当主线程停顿时
- 何时终止预计算线程?
  1. 由预插入的CANCEL指令终止
  2. 基于有效性/竞争的反馈

81

81

阅读

- Luk, “Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,” ISCA 2001.
- Zilles and Sohi, “Execution-based Prediction Using Speculative Slices”, ISCA 2001.
- Zilles and Sohi, ”Understanding the backward slices of performance degrading instructions” , ISCA 2000.

82

82

虚拟存储

83

83

现代虚拟存储的两个部分

- 在多任务系统中，虚拟内存为每个进程提供了一个大的、私有的、统一的内存空间幻象
- 命名和保护
  - 每个进程都看到一个大的、连续的地址空间（为了方便）
  - 每个进程的内存都是私有的，即受保护不被其他进程访问（为了共享）
- 需求分页（针对层次结构）
  - 辅助存储容量(磁盘上的交换空间)
  - 主存储速度（DRAM）

84

84

共同的基准：地址翻译

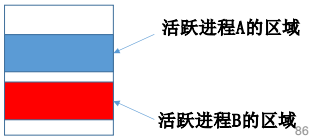
- 通过地址翻译实现大的、私有的、统一的抽象
  - 用户进程基于有效地址（EA）运行
  - 在每次内存引用时硬件将有效地址翻译成物理地址
- 通过地址翻译
  - 控制进程可以引用哪些物理位置(DRAM和/或交换磁盘)
  - 允许动态分配和重新定位物理存储(在DRAM和/或交换磁盘中)
  - 地址转换的硬件和策略由操作系统控制，受用户保护

85

85

内存保护的演化

- 早期不需要保护或翻译
  - 单一进程，每次单一用户
  - 直接使用物理地址(PA)访问所有位置
- 多任务
  - 每个进程被限制在非重叠、连续的物理存储区域(存储空间不是从0 地址开始的。。。)
  - 所有的内容都必须放在这个区域
  - 如何防止一个进程读取或破坏另一个进程的代码和数据？

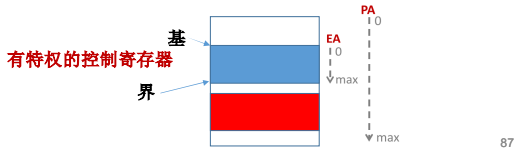


86

86

基和界

- 一个进程的私有存储区域被定义为
  - 基：该区域的首地址
  - 界：该区域的大小
- 用户进程发出从0到界的“有效”地址（EA，私有且统一）



87

87

基和界寄存器

- 翻译和保护机制针对每一次用户的内存访问检查硬件
  - 物理地址(PA)=有效地址(EA)+基
  - 如果有效地址>=界则产生冲突
- 每次切换用户进程，操作系统设置基和界寄存器
- 用户进程不能自行修改基和界寄存器

操作系统需要至少2个有受保护指令和状态的特权级

88

88

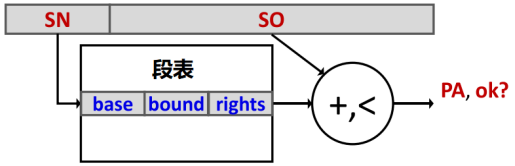
分段的地址空间

- 单由基和界确定存储区域的局限性
  - 可分配空间变得支离破碎，很难找到大的连续空间
  - 两个进程可以共享一些存储区域，但不能共享其他存储区域吗？
- 一组“基和界”是保护机制起作用的基本单元
  →给用户多个内存“段”
  - 每个段是连续的存储区域
  - 每个段由一对基和界定义
- 在最开始，代码段和数据段分离
  - 代码与数据使用2组基和界
  - 子进程可以共享代码段

  后来变得复杂:代码、数据、堆栈等

分段的地址翻译

- 有效地址被划分为段号和段偏移量
  - 段的最大尺寸受段偏移量限制
  - 界动态的设置段的大小
- 每进程一张段翻译表
  - 将段号映射到相应的基和界
  - 每个进程独立映射
  - 用于实施保护的特别结构



分成几个大段有利于隔离管理

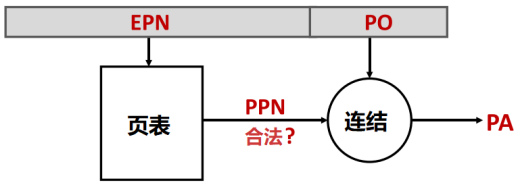
访问保护

- 段表项中的保护位表征每一段的访问权限
- 通常的选项有
  - 可读(R)?
  - 可写(W)?
  - 可执行(E)?
  - ……（可能是各种混杂的选项，比如可高速缓存）
- 比如
  - 正常的数据段→RW(!E)
  - 静态共享数据段→R(!W)(!E)
  - 代码段→R(!W)E
  - 非法段→(!R)(!W)(!E)

  访问冲突异常会引发操作系统介入

分页的地址空间

- 将物理地址和有效地址空间分成大小相等且尺寸固定的片段，称为页（页帧，最经典的大小是4KB）
- 物理地址和有效地址都可以解释为页号+偏移量
  - 页表将有效页号翻译成物理页号，偏移量是相同的
  - 物理地址=物理页号+页内偏移量



分成很多个页有利于分配管理