高等计算机体系结构
第十二讲: 虚拟存储和隐藏访存延迟

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
2020-06-05

1

提醒：作业

- 作业 5
  - 已截止
  - Cache和Memory
- 作业 6
  - 今晚发布，6月19日截止
  - 预取和并行

2

实验2-5

- 5月10日发布，预计7月10日截止

3

阅读材料

- 虚拟存储和并行
  - 必读
    - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
    - Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979
    - 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：5.4，5.8
  - 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
    - Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966
    - Hill, Jouppi, Sohi, "Multiprocessors and Multicomputers," pp. 551-560 in Readings in Computer Architecture.
    - Hill, Jouppi, Sohi, "Dataflow and Multithreading," pp. 309-314 in Readings in Computer Architecture.
    - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

4

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

5

5

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.

6

6

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

7

7

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

8

8

回顾：多核系统中的内存干扰和调度

- FR-FCFS (行缓冲优先)
  - 最大化DRAM吞吐量
  - 确保向前推进
- 多核环境下进程间干扰不受控导致性能不可预测
  - 感知QoS的内存控制
- 其它处理干扰的方法
  - 目标: 减少/控制干扰
    - 优先级或请求调度
    - 数据映射到Bank/通道/Rank
    - 核/源调节
    - 应用/线程调度

9

9

回顾：新型的非易失性存储技术

- 问题: 非易失存储器件一直以来都比DRAM慢很多
- 机遇: 一些新兴的存储技术，非易失而且相对比较快
- 提问: 是否可以采用这些新兴技术来实现主存储器？
- 新兴的电阻式存储器技术
  - PCM(相变存储器)
  - STT-MRAM(自旋转矩磁随机存取存储器)
  - Memristor(忆阻器)
- 基于PCM的主存储器
  - 如何组织：纯，混合
  - 设计上的问题

10

10

回顾：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）
- 通过地址翻译
  - 实现大的、私有的、统一的抽象
  - 控制进程可以引用哪些物理位置，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

11

11

回顾：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
- 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
  - 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
- 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

12

12

回顾：分段和分页

- 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

- 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）
  - 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

13

13

回顾：碎片

- 按段划分导致的外部碎片
  - 有足够大的连续区域，但是存在大量未分配的DRAM空间
  - 内存分页则消除了外部碎片
- 页的内部碎片
  - 分配整个页，页内未使用的字节会被浪费掉
  - 较小的页尺寸能减少内部碎片
  - 现代ISA正在向更大的页尺寸变化（MB）

14

14

回顾：访问保护

- 表项中的保护位表征访问权限
- 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）

15

15

回顾：请求页面调度（缺页中断）

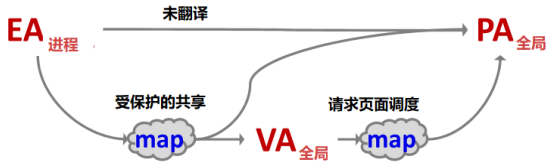
- 使用主存和“交换”磁盘作为*自动管理*的内存层级
  类似于缓存和主存
- 和Cache一样的基本问题
  (1)在DRAM的什么位置“缓存”页面?
  (2)如何在DRAM中找到一个页面?
  (3)什么时候把一页放进DRAM?
  (4)将哪个页面从DRAM置换到磁盘，释放DRAM用于新页面?
- 关键概念差异：交换vs.缓存
  - DRAM不保存磁盘上内容的副本
  - 一页既在DRAM中也在磁盘上–
  - 地址始终没有绑定到一个位置
- 规模和时间尺度的差异导致完全不同的实现选择

16

16

回顾：虚拟存储和“虚存”

- 有效地址(EA)：由每个进程空间中的用户指令发出(保护)
- 物理地址(PA)：对应于DRAM或交换磁盘上的实际存储位置
- 虚拟地址(VA)：指系统范围内的大的线性地址空间中的位置；并非虚拟地址空间中的所有位置都有物理支持(按页调度)



17

17

回顾：页表的大小

- 页表保存着从虚拟页号到物理页号的映射
- 每个进程有一个页表，页表可能很大

- 不需要跟踪整个虚拟地址空间
- 好的页表设计应该随物理存储大小线性扩展而不是虚拟地址空间
- 表不能太复杂

今天主要有两种使用模式: 分层页表和哈希页表

18

18

回顾：快表-TLB

- 用户的每次访存都需要翻译
- 用“cache”保存最近使用过的翻译
- 与cache和BTB类似的“标签”查找结构
  - 相同的设计考虑
    - C：L1 指令TLB应覆盖与L1 cache相同的空间
    - B：访问一页后，访问下一页的可能性有多大？
    - a：相联度最小化冲突？
  - TLB的表项：

标签：地址tag(来自VA), ASID
页表项(PTE)：PPN和保护位
其它：有效位、脏位等

19

19

虚存和cache交互

20

地址翻译与cache

- 什么时候需要做地址翻译?
  - 访问L1 cache之前还是之后?
- 换句话说, cache是虚拟编址还是物理编址?
  - 虚拟 vs 物理 cache
- 即使有TLB，翻译也需要时间
- 简单地讲，最佳情况下的内存访问时间是
  TLB命中时间+ cache命中时间
- 为什么不使用虚拟地址访问cache：只在cache miss 要访问DRAM时才做地址翻译
  如果TLB命中时间>> cache命中时间就有意义
- 大约在1990年，SUN SPARC 指令集的虚拟cache
  - 处理器速度足够快，片外SRAM访问需要多个周期
  - 芯片尺寸足够大可以设计片内L1 cache
  - MMU和TLB在不同的芯片上

21

21

虚拟cache中的同义词和同音异义词

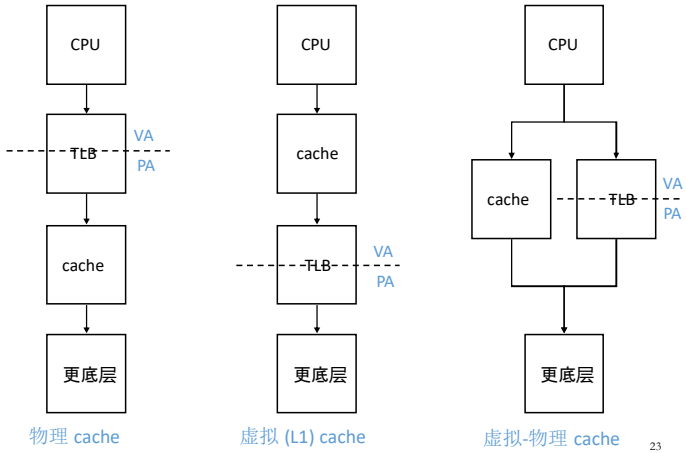
- 同音异义：相同的声音不同的含义
  - 相同的EA(不同进程中)→不同的PA（同一个虚拟地址可能映射到两个不同的物理地址）
    - 虚拟地址可能在不同的进程中
  - 在上下文之间刷虚拟cache，或者在cache标签中加入ASID
- 同义词：不同的声音相同的含义（不同的虚拟地址可能映射到同一个物理地址）
  - 不同的EA(相同或不同的进程)→相同的PA
    - 不同的页可能共享同一进程内或者跨进程的物理帧
    - 原因: 共享库, 共享数据, 同一个进程内的写时拷贝页, …
  - PA可以由不同的EA cache两次
  - 写一个cache的副本不会反映在另一个cache的副本中

要确保一次仅在cache中有一个这样的EA

22

22

Cache-VM 如何交互

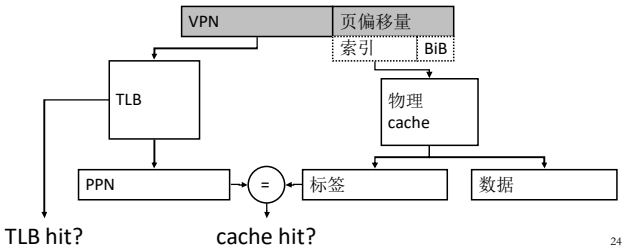


23

23

虚拟-索引,物理-标签

- 如果 Cache ≤ (页大小 × 相联度), cache索引位只来自页的偏移量部分 (在虚拟地址和物理地址中相同)
- 如果片内有cache和TLB
  - 用虚拟地址同时索引cache和TLB
  - cache检查标签(物理的)并比对TLB的输出给出结果

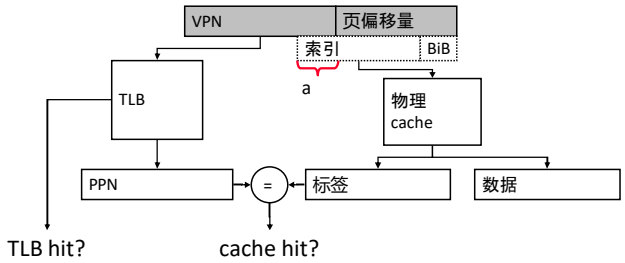


24

24

虚拟-索引,物理-标签

- 如果 Cache > (页大小 × 相联度), cache索引位将包含 VPN ⇒ “同义词”会引发问题
  - 同一个物理地址可能存在于两个不同的位置
- 如何解决?



25

25

解决“同义词”问题的一些方法

- 限制cache大小 (页大小x相联度)
  - 只从页偏移量获得索引
- 写一个块时，搜索所有可能包含相同物理块的标记, 更新/置为无效
  - Alpha 21264, MIPS R10K
- 在操作系统中限制页的放置
  - 确保虚拟地址的索引 = 物理地址的索引
  - 称为页着色
  - SPARC

26

26

深入思考的问题

- 在哪一层cache我们需要考虑“同义词”和“同音异义词”的问题?
- 系统软件的页映射算法对分层存储结构的哪一层会产生影响?
- 页着色有哪些潜在的优点和缺点?

27

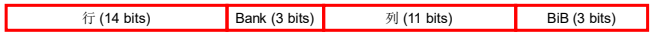
27

虚存和内存交互

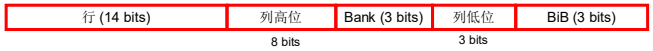
28

内存地址映射(单通道)

- 8字节内存总线的单通道系统
  - 2GB 内存, 8个 Bank, 每个 Bank 有 16K 行 x 2K 列
- 行交叉存取
  - 内存中连续的行在连续的Bank中



- Cache block交叉存取
  - 连续的cache block地址在连续的Bank中
  - 64字节的cache blocks



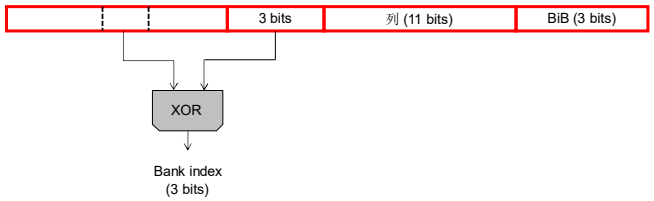
  - 访问连续的 cache block 可以并行
  - 随机访问? 跨步访问?

29

29

Bank 随机映射

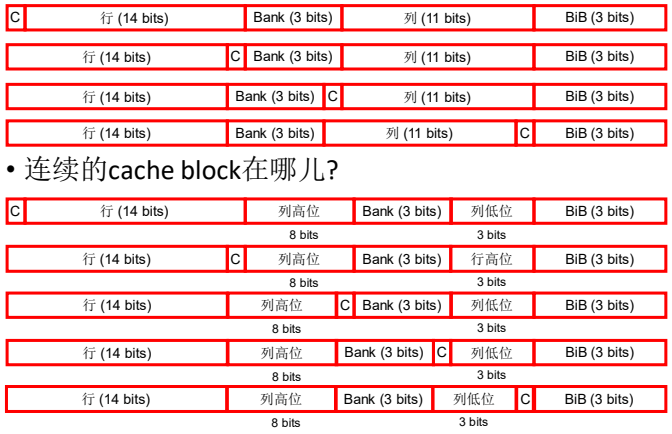
- DRAM控制器可以随机映射地址到Bank，这样就不太可能出现Bank冲突了



30

30

内存地址映射(多通道)



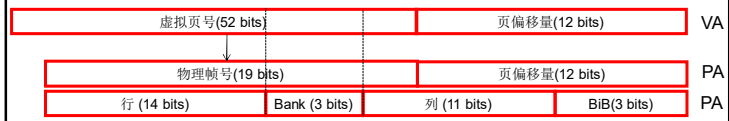
- 连续的cache block在哪儿?

31

31

虚拟存储– DRAM 的交互

- 操作系统会影响DRAM中的地址映射



- 操作系统能够控制虚页映射到哪一个bank/channel/rank
- 可以通过页着色最小化bank的冲突
- 或者最小化应用之间的干扰

32

32

虚拟存储和DMA的交互

- VA中连续的块
  - 在PA中不保证连续
  - 可能根本不在内存中
- 软件解决方案
  - 在DMA之前，内核从用户缓冲区复制到固定的连续缓冲区，或者
  - 用户为零拷贝DMA分配特殊的固定连续页
- 更智能的DMA引擎可依照一个命令“链表”移动不连续块
- 虚拟编址I/O总线(带I/O MMU)

访存延迟容忍

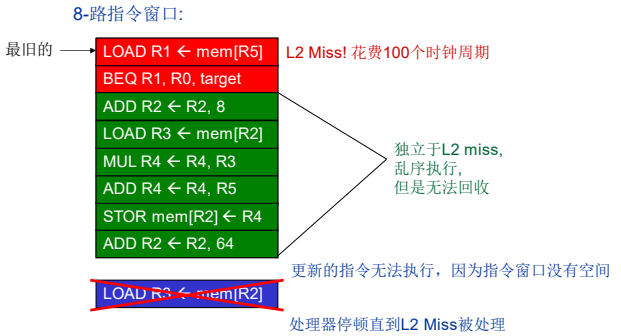
延迟容忍

- 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?

由长延迟指令导致的停顿

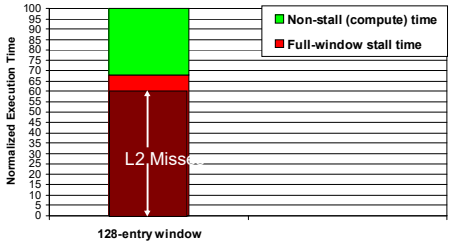
- 当一条长延迟指令没有结束, 它会阻碍指令回收
  - 因为需要保证精确异常
- 输入的指令填满指令窗口(重排序缓冲, 保留站)
- 一旦窗口填满, 处理器无法继续向窗口中放入新的指令
  - 称为满窗口停顿
- 满窗口停顿会阻止处理器向前推进正在执行的程序

满窗口停顿



- L2 cache的缺失是导致满窗口停顿的最主要原因

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

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

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

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

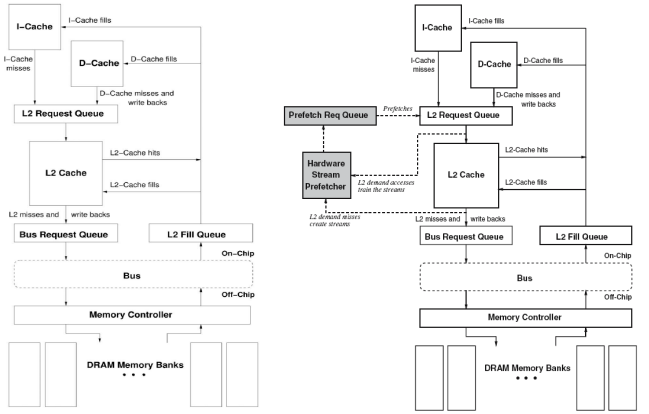
43

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

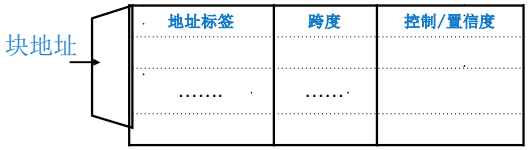
基于指令的跨度预取

- 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



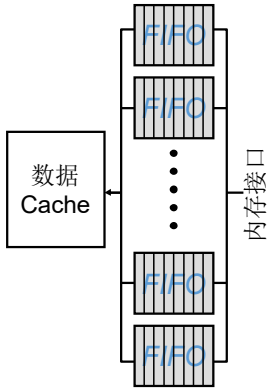
- 能够检测
  - A, A+N, A+2N, A+3N, …
  - 流缓冲是基于cache块地址的跨度预取的一个特例，N = 1
    - Jouppi的论文

61

61

流缓冲(Jouppi, ISCA 1990)

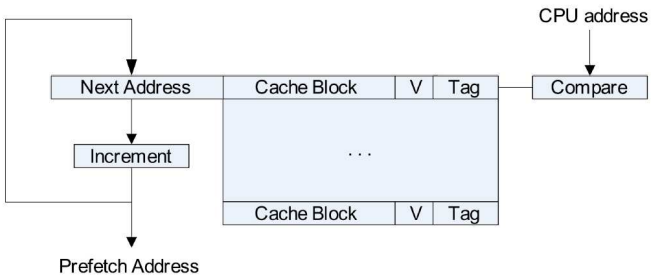
- 每个流缓冲保持一个顺序预取的cache行的流
- 当发生load缺失时，检查所有流缓冲头部是否有地址匹配
  - 如果命中, 从FIFO队列中弹出，更新cache中的数据
  - 如果不命中, 向新的缺失地址分配一个新的流缓冲(可能需要按照LRU策略回收一个流缓冲)
- 只要有空间并且总线不忙，流缓冲的FIFO队列会持续不断地放入后续的cache行



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

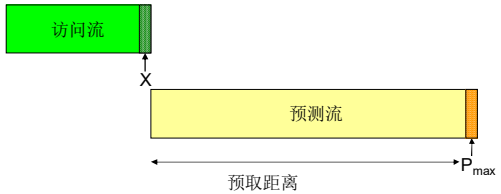
- Cache污染
  - 由于预取放在cache中导致的额外的按需访问缺失
  - 很难量化，但是会影响性能

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

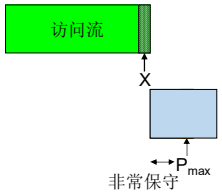


65

65

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

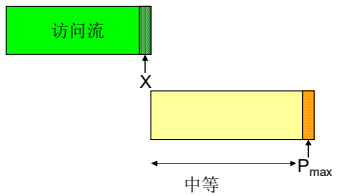


66

66

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

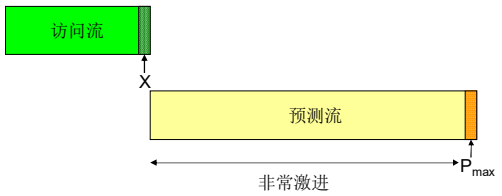


67

67

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

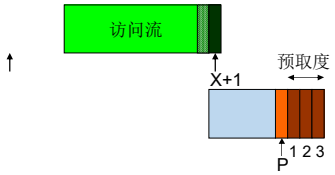


68

68

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



69

69

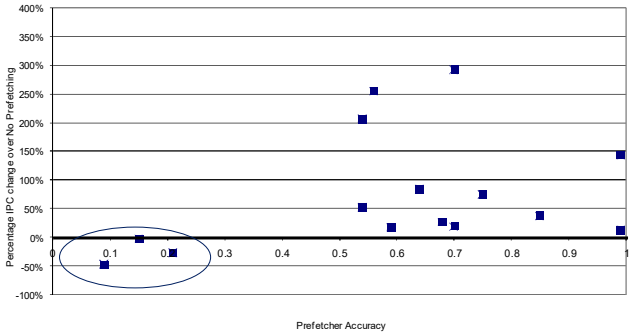
预取器性能(III)

- 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

70

70

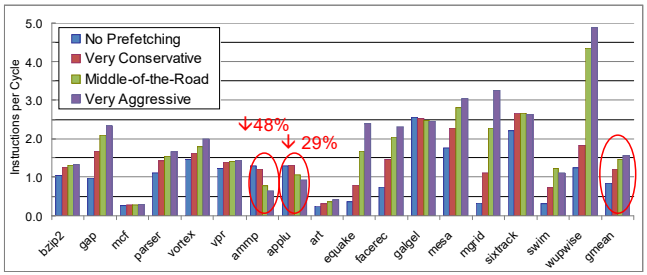
预取器性能(IV)



71

71

预取器性能(V)



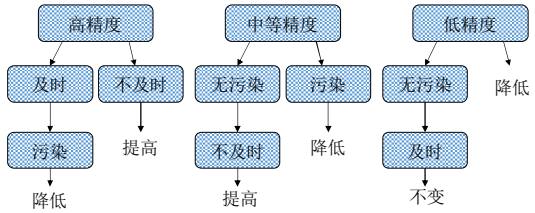
- Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers", HPCA 2007.

72

72

基于反馈的预取器调节(I)

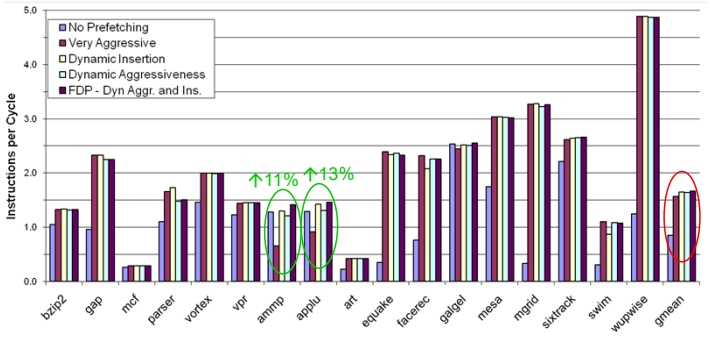
- 思路:
  - 动态监控预取器性能指标
  - 基于过去的性能状况调节预取器的激进程度
  - 基于过去的性能状况改变预取插入cache的位置



73

73

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.

74

74

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

75

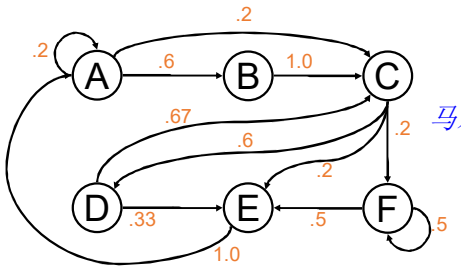
75

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

76

76

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.

77

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

78

基于内容的预取(I)

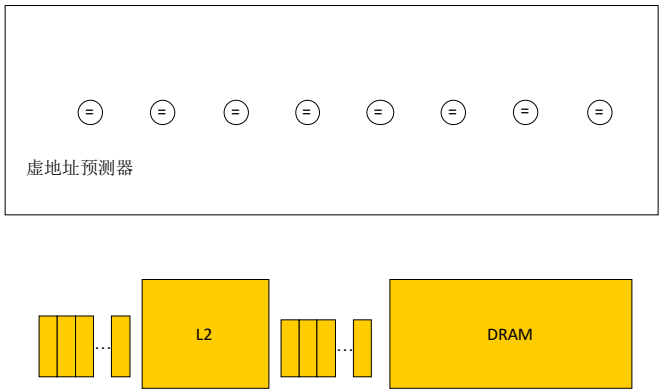
- 一种针对指针值的特殊预取器
- Cooksey et al., "A stateless, content-directed data prefetching mechanism," ASPLOS 2002.
- 思路: 识别取回的cache块中的指针，发射针对它们的预取请求

+ 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

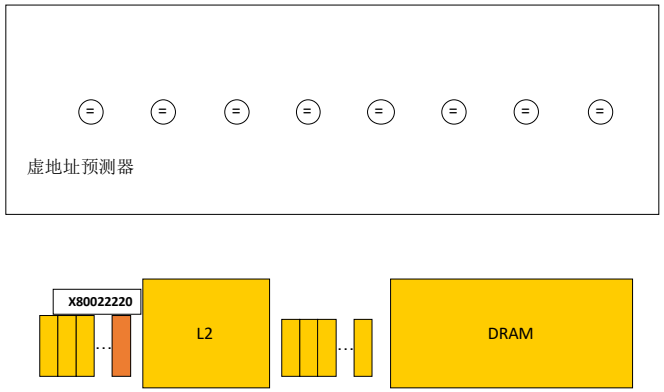
79

基于内容的预取(II)



80

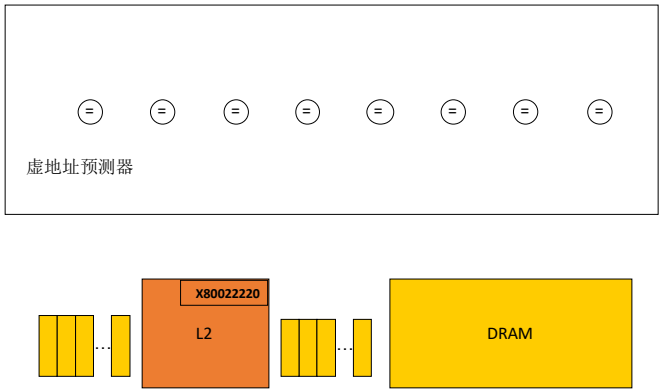
基于内容的预取(II)



81

81

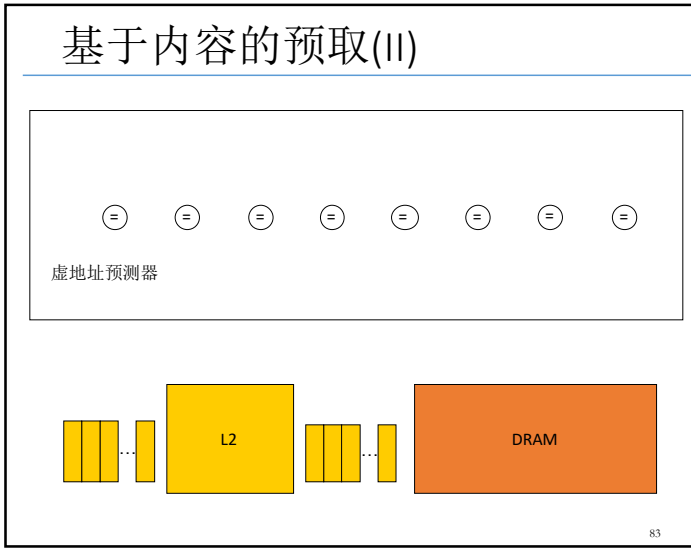
基于内容的预取(II)



82

82

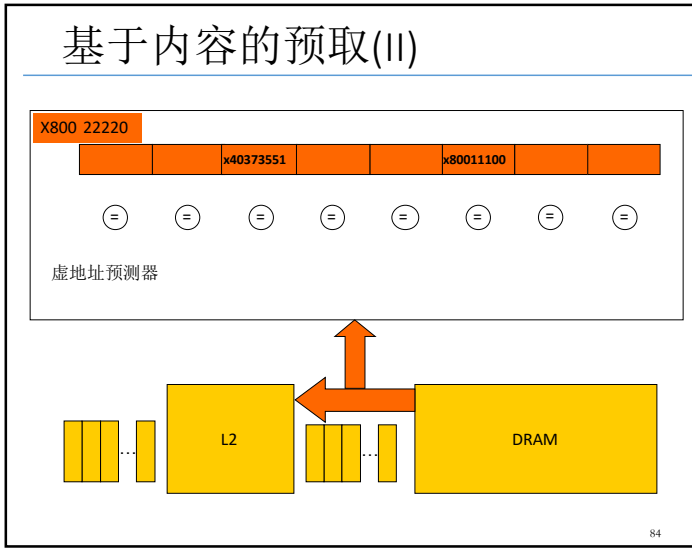
基于内容的预取(II)



83

83

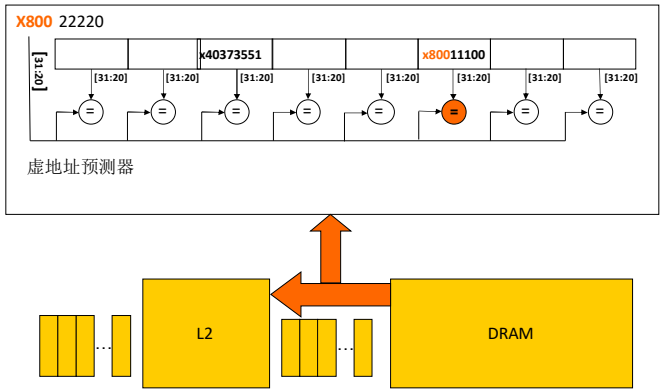
基于内容的预取(II)



84

84

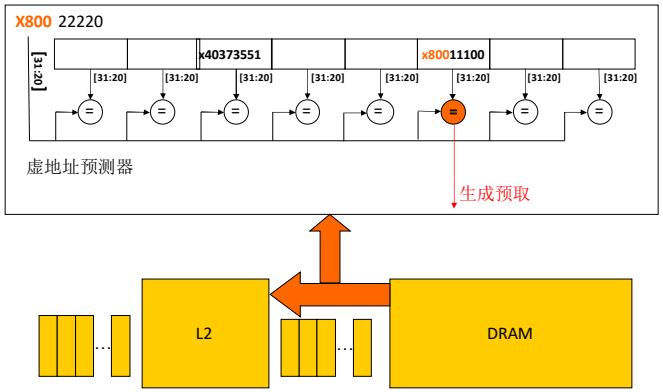
基于内容的预取(II)



85

85

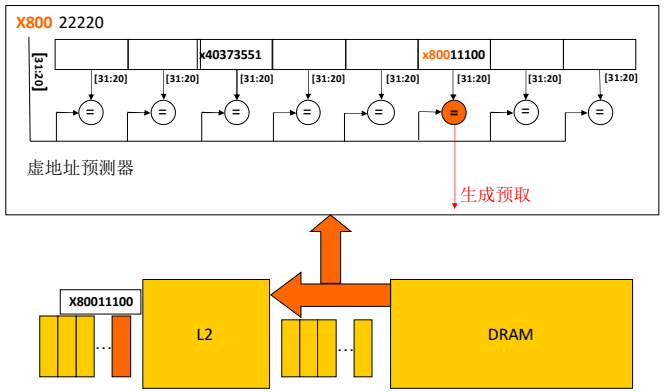
基于内容的预取(II)



86

86

基于内容的预取(II)



87

87

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.

88

88

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)

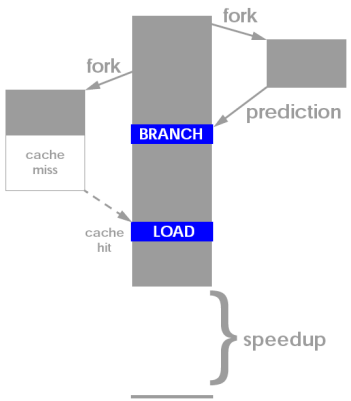
89

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

90

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.

91

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

92

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.

93