高等计算机体系结构
第十二讲: 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

1

访存延迟容忍

2

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?

3

3

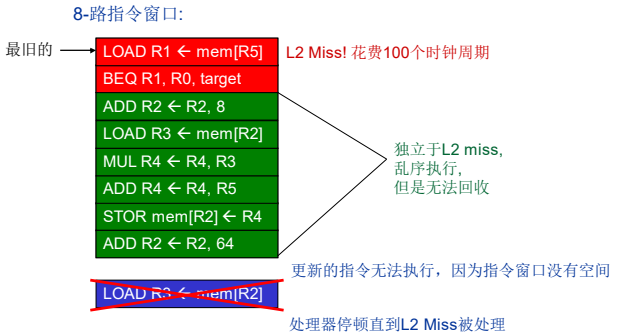
由长延迟指令导致的停顿

- 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

4

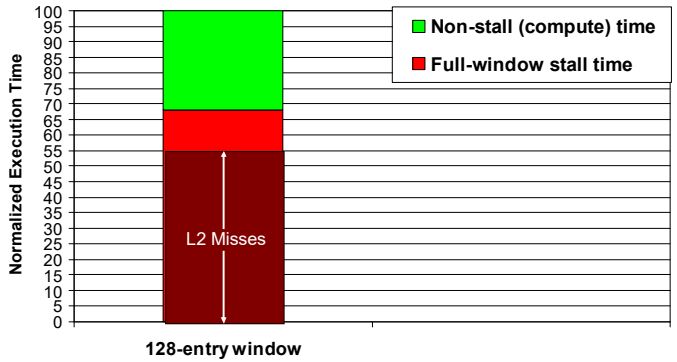
4

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

9

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

10

10

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

11

11

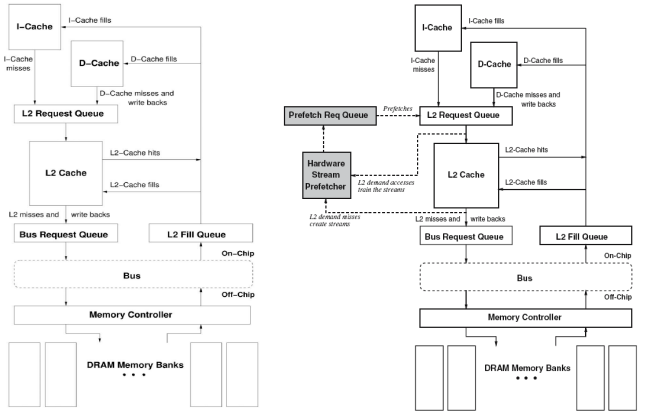
基础

- 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

12

12

在存储系统如何加入硬件预取器



13

13

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

14

14

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

15

15

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)

16

16

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, …

17

17

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?

18

18

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?)

19

19

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

20

20

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

21

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

22

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

23

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

24

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

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?

跨度(步长)预取器

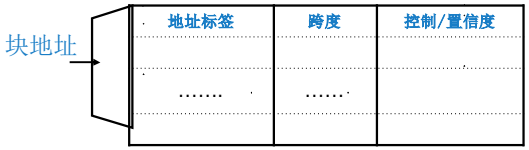
- 两种
  - 基于指令指针/程序计数器
  - 基于Cache块地址

- 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

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

基于cache块地址的跨度预取



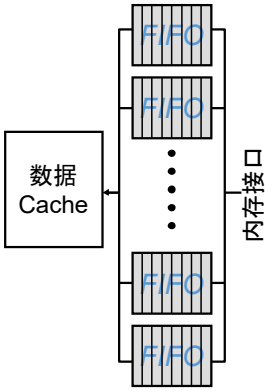
- 能够检测
  - A, A+N, A+2N, A+3N, …
  - 流缓冲是基于cache块地址的跨度预取的一个特例，N = 1
    - Jouppi的论文

29

29

流缓冲(Jouppi, ISCA 1990)

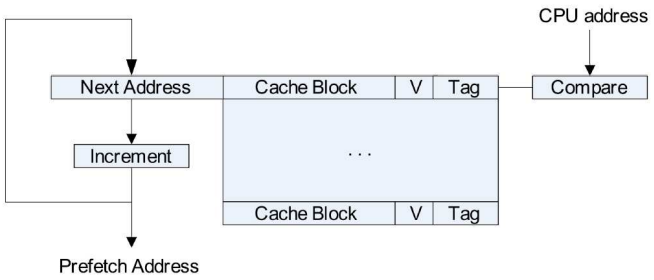
- 每个流缓冲保持一个顺序预取的cache行的流
- 当发生load缺失时，检查所有流缓冲头部是否有地址匹配
  - 如果命中, 从FIFO队列中弹出，更新cache中的数据
  - 如果不命中, 向新的缺失地址分配一个新的流缓冲(可能需要按照LRU策略回收一个流缓冲)
- 只要有空间并且总线不忙，流缓冲的FIFO队列会持续不断地放入后续的cache行



30

30

流缓冲设计



31

31

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

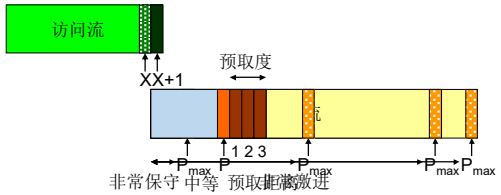
- Cache污染
  - 由于预取放在cache中导致的额外的按需访问缺失
  - 很难量化，但是会影响性能

32

32

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



33

33

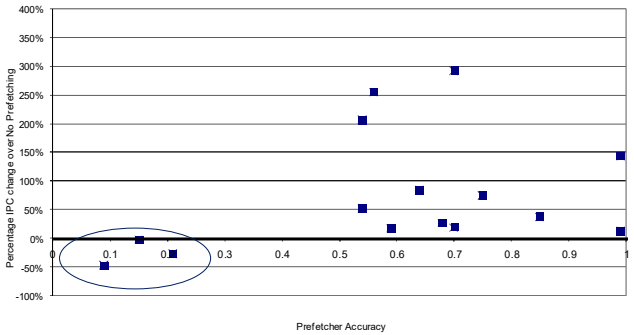
预取器性能(III)

- 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

34

34

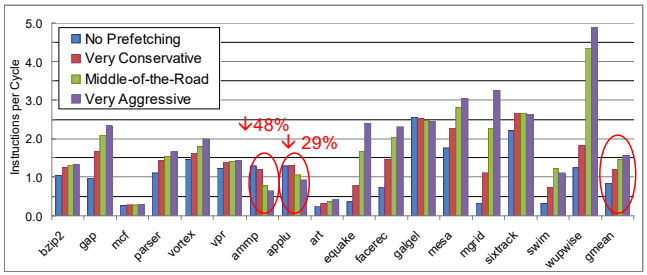
预取器性能(IV)



35

35

预取器性能(V)



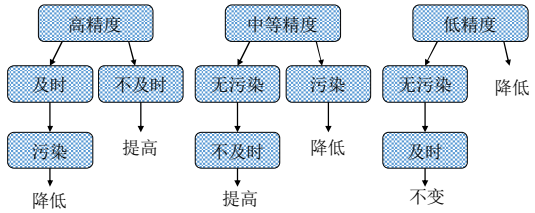
- Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers", HPCA 2007.

36

36

基于反馈的预取器调节(I)

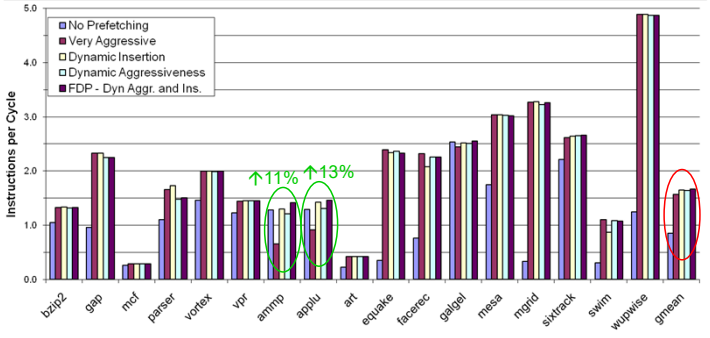
- 思路:
  - 动态监控预取器性能指标
  - 基于过去的性能状况调节预取器的激进程度
  - 基于过去的性能状况改变预取插入cache的位置



37

37

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.

38

38

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

39

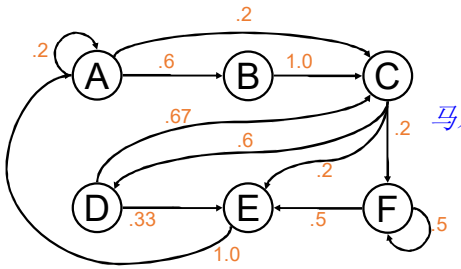
39

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

40

40

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.

41

41

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

42

42

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

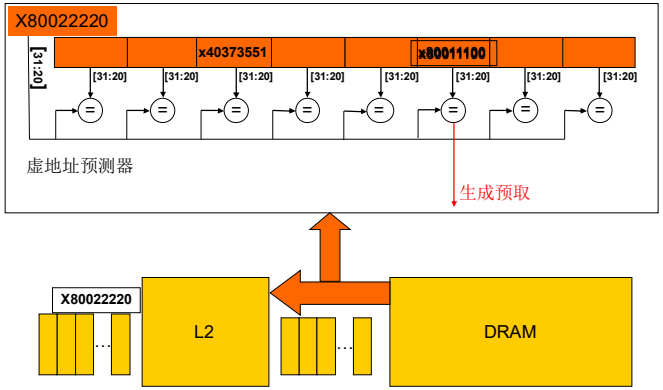
+ 无需记忆过去的地址!
+ 可以消除强制缺失 (从未见过的指针)
-- 不加选择地预取cache块中的所有指针

- 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

43

43

基于内容的预取(II)



44

44

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.

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"

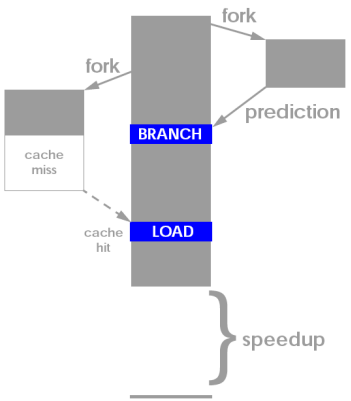
  - 投机线程可以执行在
- 独立的处理器/核
- 独立的硬件线程上下文 (回忆细粒度多线程)
- 相同线程上下文的空闲周期 (在cache缺失期间)

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

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.

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

49

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.

50

系统能耗分析：概念和基本方法

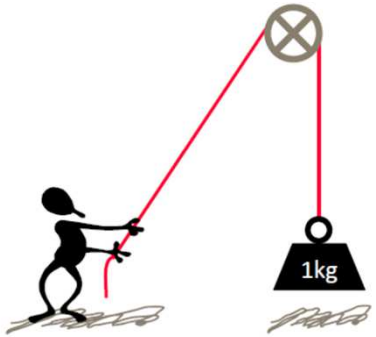
51

什么是能耗?

52

力：牛顿=千克·米/秒^2

- 9.8牛顿可以支持1千克质量的物体对抗重力
- 抓着绳子不消耗能量，不管重量有多重



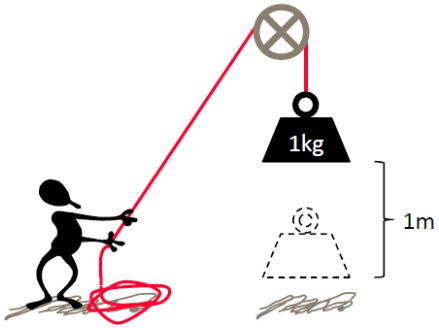
53

53

能量：焦耳=牛顿·米

- 9.8焦耳可以克服重力将1千克质量的物体提升1米

变化前后之间的静态概念



54

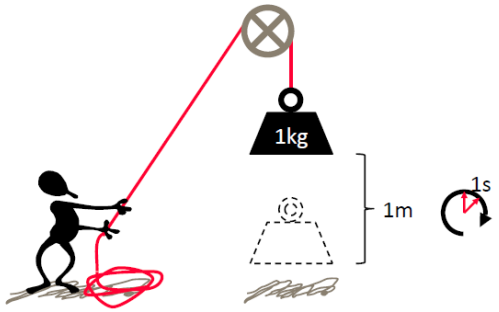
54

功率：瓦特=焦耳/秒

- 9.8瓦可以在1秒内克服重力将1千克质量物体提升1米

变化率的动态概念

- 9.8瓦可以是1千克/10米/10秒、10千克/1米/10秒等等



55

55

电子学中的能量与功率

- CMOS逻辑转换涉及到电容的充电和放电
- 当“电荷”从电源(VDD)流向地(GND)时，能量(焦耳)以阻抗产生的热耗散掉
  - 每次操作需要一定量的能量，例如，加法、寄存器读/写、对节点充放电
  - 能量∝计算量(功)

此外，只要保持通电就会有“漏”电流！！

- 功率(瓦特=焦耳/秒)是能量耗散率
  - 运算数/秒越高，焦耳/秒就越大
  - 功率∝性能

如果性能∝频率，那么一切将变得简单

功率=(½CV^2)·f

56

56

功和时间

- ***W***
  - 表示一个任务的“工作量”的标量（变量）
- ***T=W/C_{perf}***
  - 表示一个任务的执行时间
  - C_{perf} 是表示做功速率的标量（常量），即“单位时间做的功”

57

57

能量和功率

- $E_{switch}=c_{switch}W$
  - 与任务相关的“开关”能量
  - c_{switch}是表示做单位功产生的能量的标量（常量）
- $E_{static}=c_{static}T=c_{static}W/c_{perf}$
  - 保持给芯片供电产生的“泄漏”能量
  - c_{static}是所谓的“漏电功率”
- $E_{total}=E_{switch}+E_{static}=c_{switch}W+c_{static}W/c_{perf}=(c_{switch}+c_{static}/c_{perf})\cdot W$
  - 在高性能处理器中，静态功率可以接近50%
- $P_{total}=E_{total}/T=(c_{switch}W+c_{static}T)/T=c_{switch}c_{perf}+c_{static}$

58

58

简而言之，

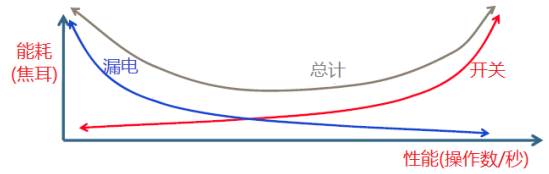
- 对于一个做功为*W*的任务
  - $T = W / c_{perf}$
    做功（工作量）越少的任务执行得越快
  - $E = E_{switch} + E_{static} = (c_{switch} + c_{static} / c_{perf})\cdot W$
    做功（工作量）越少的任务消耗的能量越少
  - $P = P_{switch} + P_{static} = c_{switch}c_{perf} + c_{static}$
    功率与任务不相关
- 现实是
  - W不是标量
  - c既不是标量也不是常数
  - ½CV^2和½CV^2f本身是非常“粗略”的近似值

59

59

关于静态功率的特别说明

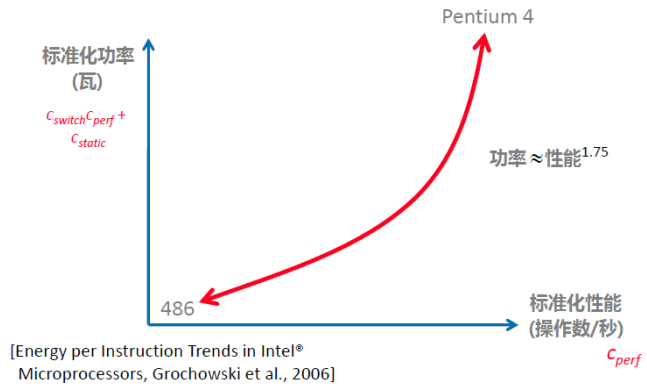
- $E_{total} = (c_{switch} + c_{static} / c_{perf})\cdot W = \underbrace{c_{switch}\cdot W}_{\text{开关}} + \underbrace{c_{static}\cdot W / c_{perf}}_{\text{静态}}$
- 执行得更慢(调慢时钟)会降低功率消耗，但会由于漏电的存在而增加能量消耗
- 执行得更快(必须改进设计)需要让c_{switch}和c_{static}有超线性的增长



60

60

注意：C_{switch},C_{static} ,C_{perf}互相依赖



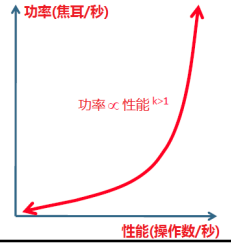
[Energy per Instruction Trends in Intel® Microprocessors, Grochowski et al., 2006]

61

61

暗示：功率和性能是密不可分的

- 如果不关心性能，很容易将功耗降至最低
- 可以预期功率超线性增加将提高性能
  - 较慢的设计更简单
  - 较低的频率需要较低的电压
  - “低挂果”优先
- 推论：较低的性能会产生较低的焦耳/操作
- 总之，越慢越节能



62

62

为什么能耗对今天的计算机体系结构如此重要？

63

63

技术缩放的入门知识

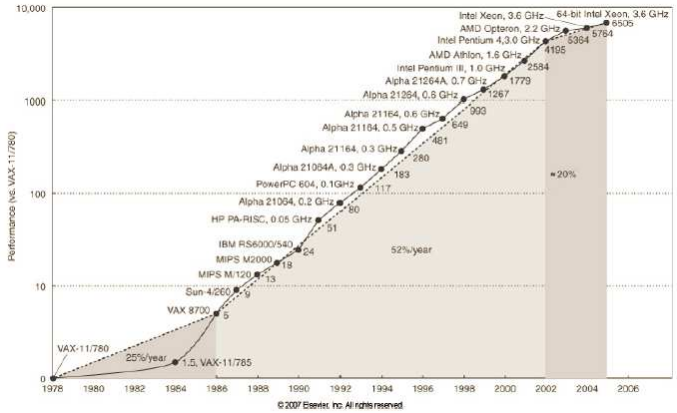
- 预计的缩放发生在离散的“工艺节点”中，每个节点线性缩放为大约先前的0.7x

  90nm, 65nm, 45nm, 32nm, 22nm, 15nm, 7nm, . . .
- 如果设计不变，尺寸线性减小0.7x(也称为“门收缩”)理想情况下会导致
  - 芯片面积= 0.5x
  - 延迟= 0.7x，频率=1.43x
  - 电容= 0.7倍
  - Vdd=0.7x(恒定磁场)或Vdd=1x(恒定电压)
  - 功率= C×V²×f = 0.5x(恒定磁场)
  - 功率= 1x(恒定电压)
- 如果面积不变
  - 晶体管数量= 2x
  - 功率= 1x(恒定磁场)，功率= 2x(恒定电压)

64

64

摩尔定律的一种表现形式



65

65

摩尔定律→性能

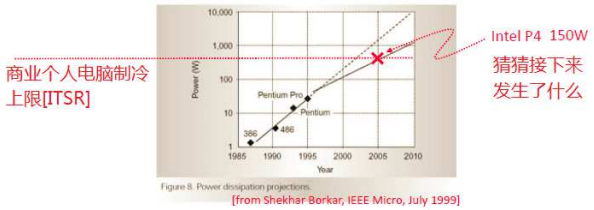
- 根据缩放理论的说法，我们应该得到
  @恒定复杂性：以1x晶体管数获得1.43x频率
  → 1.43x性能，0.5x功耗
  @最大复杂度：以2x晶体管数获得1.43x频率
  → 恒定功率下2.8x性能
- 实际上，我们得到了(高性能CPU)
  - 2x晶体管数
  - 2x频率(注意：比缩放理论快)
  - 总的来说，我们以约2x功率获得约2x性能

66

66

性能效率低下

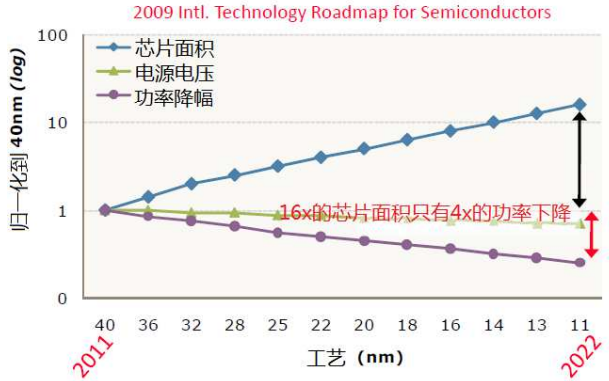
- 为了在单线程微处理器上达到“预期”的性能目标
  - 通过增加流水线深度越来越难提高频率
  - 使用2x晶体管构建更复杂的微体系结构(cache、分支预测、超标量、乱序执行)，以使更快/更深的流水线不会停顿
- 性能效率低下的后果是



67

67

登纳德缩放定律率先失效



必须以更少的焦耳/秒完成更多的操作数/秒

68

68

频率和电压缩放

- 每次转换的开关能量为½CV²(寄生电容建模)
- 每秒f次转换的开关功率为½CV²f
- 降低功率，就能降低时钟
- 如果时钟变慢了，可以通过降低电源电压来降低晶体管的速度
  - V→V'，因此½CV² → ½CV'²

漏电流/功率也由于更低的电压V'而超线性降低!!!

69

69

频率缩放

- 若$W/c_{perf} < T_{bound}$，可以通过一个因素的周期性缩放来降低性能

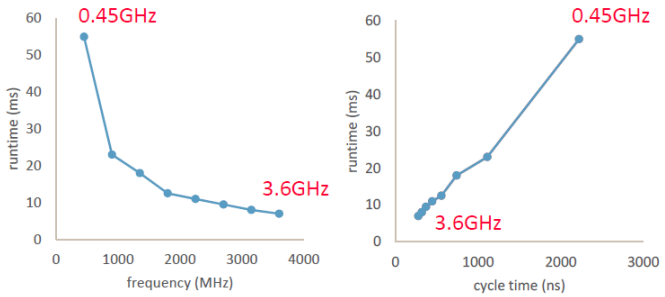
  $(W/c_{perf})/T_{bound} < s_{freq} < 1$

  当满足 $c_{perf}' = c_{perf}\, s_{freq}$
- $T' = W/(c_{perf}\, s_{freq})$
  - $1/s_{freq}$ 使执行时间变长
- $E' = (c_{switch} + c_{static}/(c_{perf}\, s_{freq}))\cdot W$
  - 更长的执行时间导致更高的(泄漏)能量
- $P' = c_{switch} c_{perf}\, s_{freq} + c_{static}$
  - 更长的执行时间导致更低的开关功耗

70

70

Intel P4 660 频率缩放: FFT_{64K}

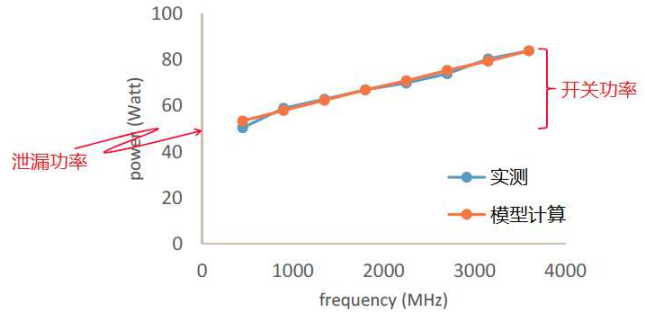


circa 2005, 90nm

71

71

Intel P4 660 频率缩放: FFT_{64K}



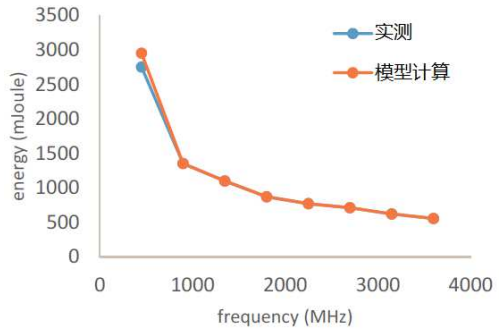
c_{perf} = 145操作数/秒; c_{switch}=0.24焦耳/操作; c_{static}=49.4瓦 (归一化到W=1)

circa 2005, 90nm

72

72

Intel P4 660 频率缩放: FFT_{64K}



c_{perf}= 145操作数/秒; c_{switch}=0.24焦耳/操作; c_{static}=49.4瓦 (归一化到W=1)

circa 2005, 90nm

73

73

频率和电压缩放

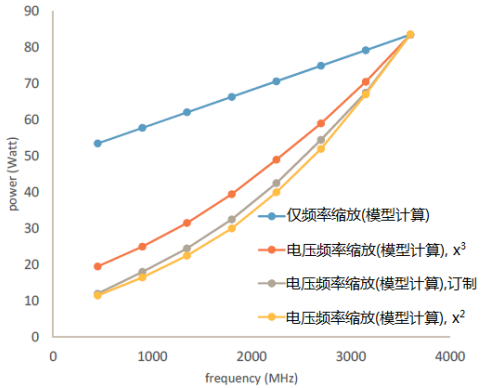
- 频率按s_{freq}缩放使得电压可以按相应的因子$s_{voltage}$缩放
- E∝V^2，因此
  - $c_{switch}'' = c_{switch} s_{voltage}^2$
  - $c_{static}'' = c_{static} s_{voltage} e^{2.7(s\_voltage-1)} \approx ?$
    简单起见，$c_{static} s_{voltage}^3$
- $T'' = W / (c_{perf} s_{freq})$
  - 执行时间增加1/ s_{freq}
- $E'' = (c_{switch} s_{voltage}^2 + c_{static} s_{voltage}^3 / c_{perf} s_{freq}) \cdot W$
  - 降低开关和静态能耗
- $P'' = c_{switch} s_{voltage}^2 c_{perf} s_{freq} + c_{static} s_{voltage}^3$
  - 开关功率按立方减小，静态功率按平方减小

架构师们通常进行粗略的简化

74

74

Intel P4 660 电压频率缩放: FFT_{64K}

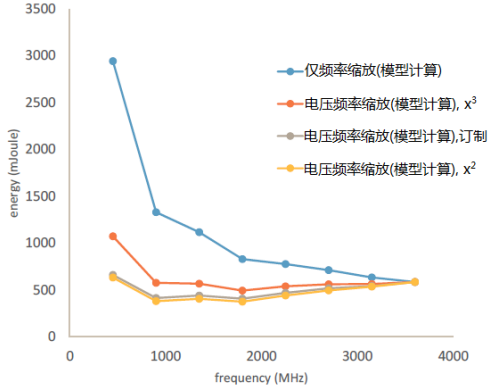


circa 2005, 90nm

75

75

Intel P4 660 电压频率缩放: FFT_{64K}

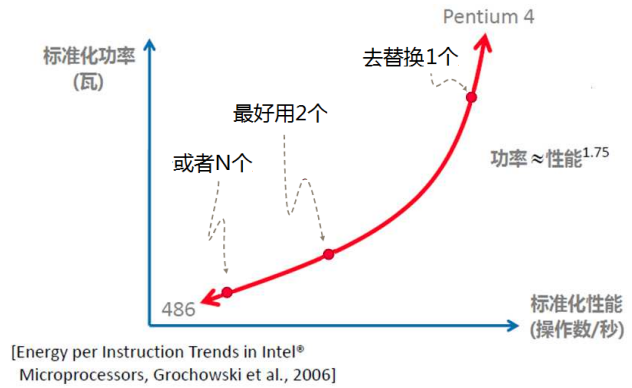


circa 2005, 90nm

76

76

并行化



[Energy per Instruction Trends in Intel® Microprocessors, Grochowski et al., 2006]

77

77

并行化

- 使用N个处理器时理想的并行化
  - $T = W / (c_{perf} N)$
  - $E = (c_{switch} + c_{static} / c_{perf}) \cdot W$

    注意：4x的静态功率，4x的执行时间提升
  - $P = N (c_{switch} c_{perf} + c_{static})$

    “我们曾经这样想”
- 或者，假设$s_{voltage} \approx s_{freq}$，我们可以基于$s_{freq} = 1/N$用加速比$N$来换取功率和能量的下降
  - $T = W / c_{perf}$
  - $E'' = (c_{switch} / N^2 + c_{static} / (c_{perf} N)) \cdot W$
  - $P'' = c_{switch} c_{perf} / N^2 + c_{static} / N$

    “我们现在这样想”

78

78

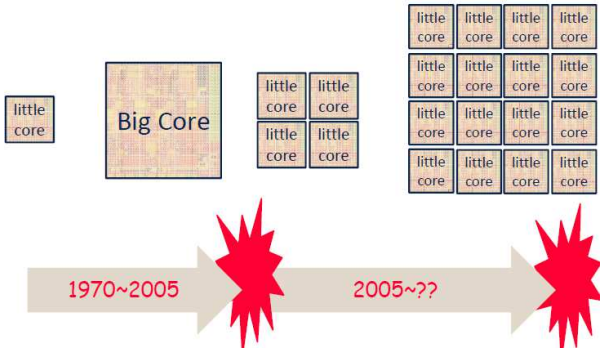
所以问题在哪儿？

- 我们知道如何在芯片上封装更多的核，从而在“聚合”或“吞吐”性能方面保持摩尔定律
- 如何使用它们？
  - 如果N个任务单元是N个独立的程序，生活是美好的⇒只需要运行它们就好了
  - 如果N个任务单元是同一程序的N个操作，该怎么办？⇒重写一个并行程序……就好了……
  - 如果N个任务单元是同一程序的N个串行相关的操作，该怎么办？⇒？？
- 能有效地使用多少核？

79

79

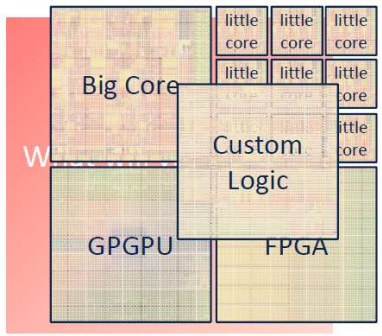
摩尔定律通过核数的增加继续扩展



80

80

记住：性能/瓦特和操作数/焦耳



81

81