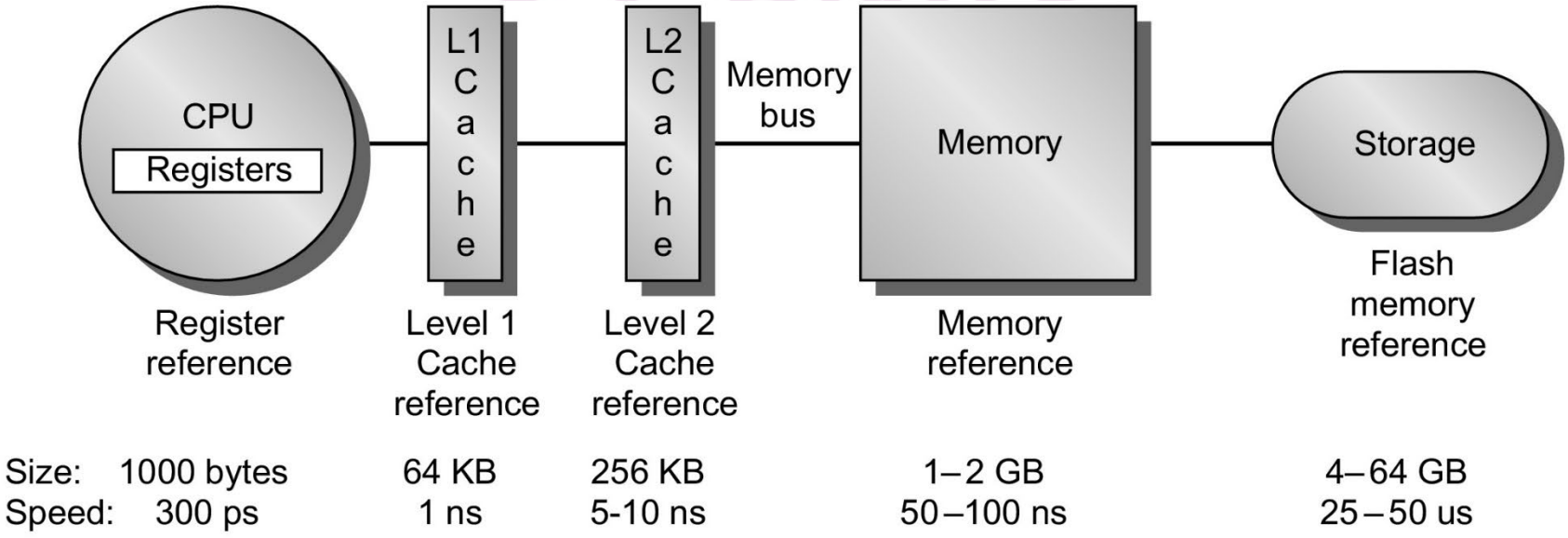
# 存储层次设计

# 引言

- 问题：每个程序员都期望拥有无限容量的快速存储器
  - 但，快速存储器的比特价格远高于慢速存储器的比特价格
- 解决方法：存储层次(memory hierarchy)
  - 性能和价格之间的权衡
    - 整个寻址空间 ⇔ 慢速大容量存储器
    - 容量越小、速度越快、比特价格越贵的存储器越靠近处理器
  - 局部性：时间局部性和空间局部性
    - 保证每次访问的数据都高概率地存储在最快访问速度的物理存储器中
- 从处理器的视角来看
  - 存储空间的容量：容量最大的物理存储器的存储容量
  - 存储空间的速度：速度最快的物理存储器的访问速度

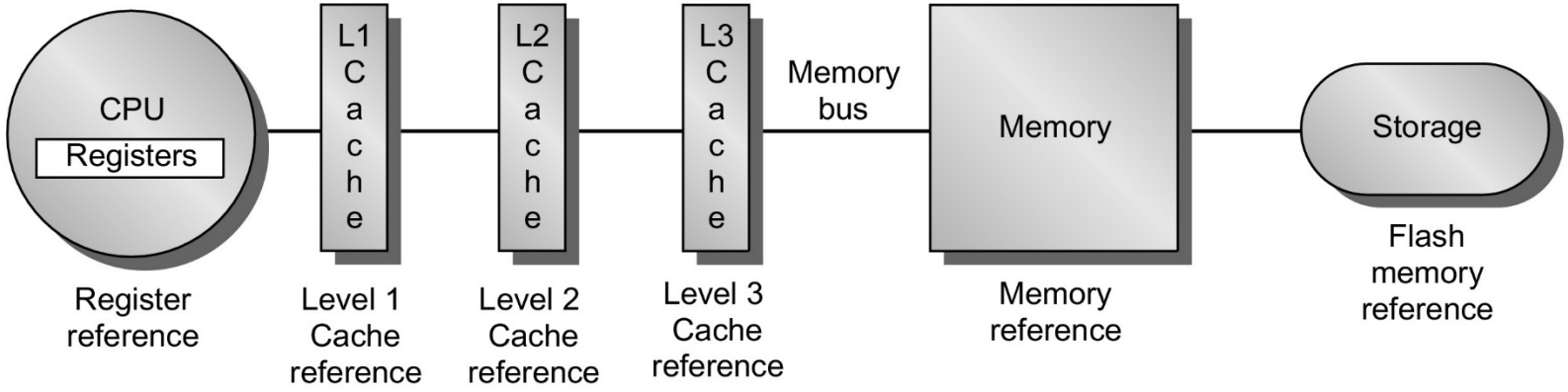
# 典型存储层次

- 个人移动设备



# 典型存储层次

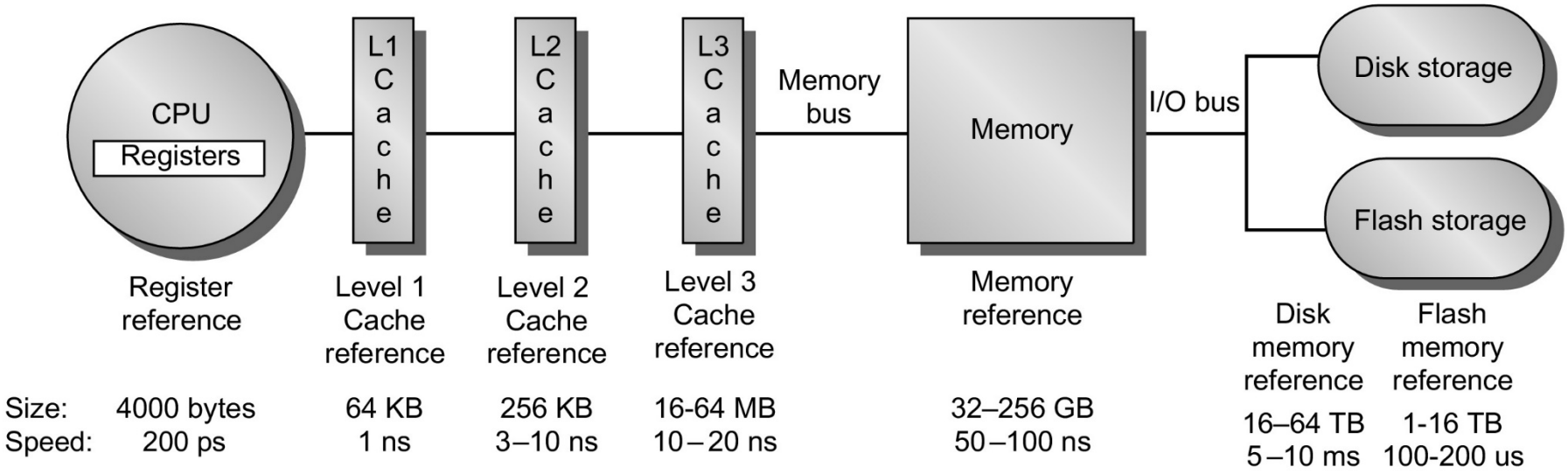
- 笔记本或台式机




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Laptop** | Size: | 1000 bytes | 64 KB | 256 KB | 4-8 MB | 4–16 GB | 256 GB-1 TB |
| | Speed: | 300 ps | 1 ns | 3–10 ns | 10–20 ns | 50–100 ns | 50-100 uS |
| **Desktop** | Size: | 2000 bytes | 64 KB | 256 KB | 8-32 MB | 8–64 GB | 256 GB-2 TB |
| | Speed: | 300 ps | 1 ns | 3–10 ns | 10–20 ns | 50–100 ns | 50-100 uS |

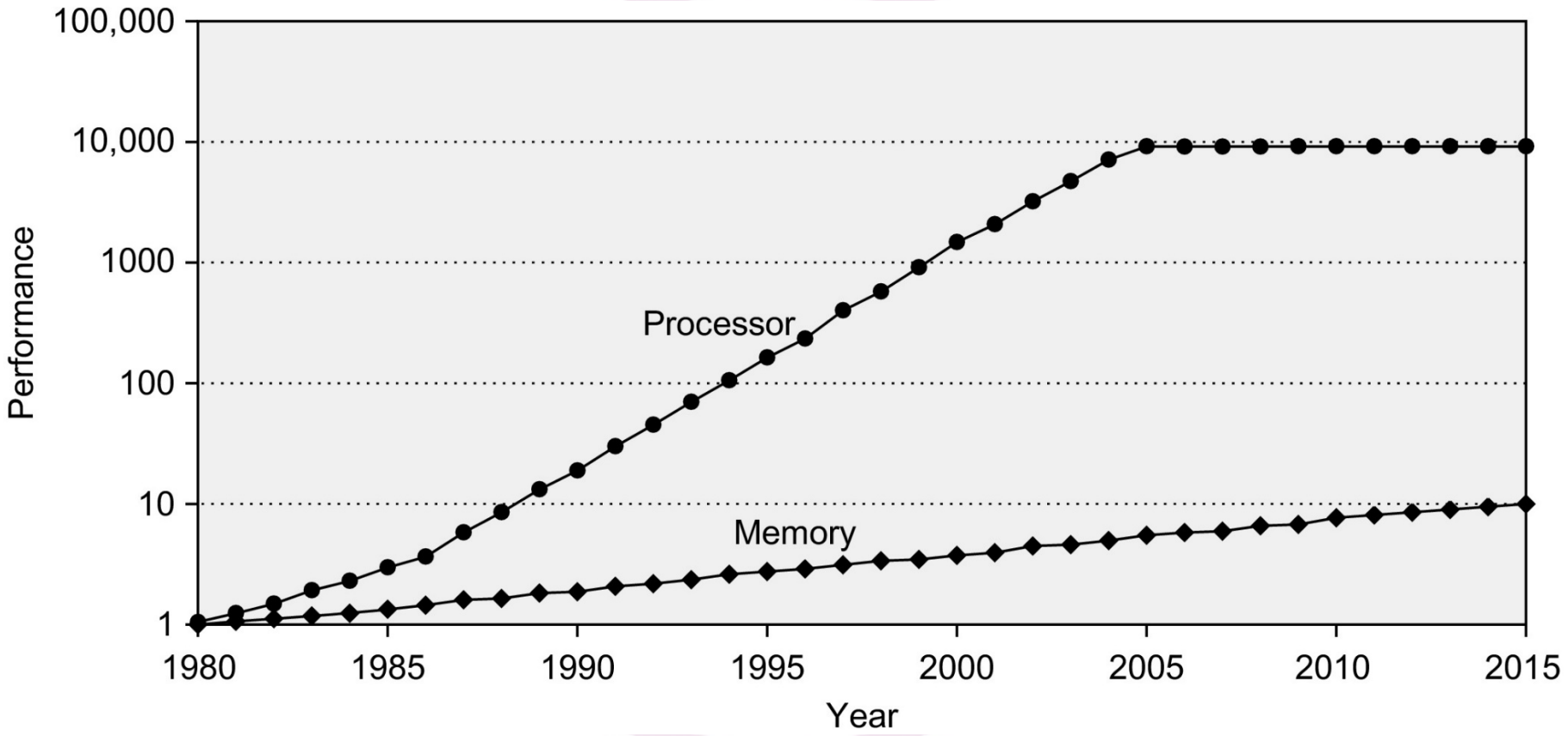
# 典型存储层次

- 服务器


CPU
Registers
L1 Cache
L2 Cache
L3 Cache
Memory bus
Memory
I/O bus
Disk storage
Flash storage
Register reference
Level 1 Cache reference
Level 2 Cache reference
Level 3 Cache reference
Memory reference
Disk memory reference
Flash memory reference
Size: 4000 bytes 64 KB 256 KB 16-64 MB 32–256 GB 16–64 TB 1-16 TB
Speed: 200 ps 1 ns 3–10 ns 10–20 ns 50–100 ns 5–10 ms 100-200 us

# 存储层次的重要性

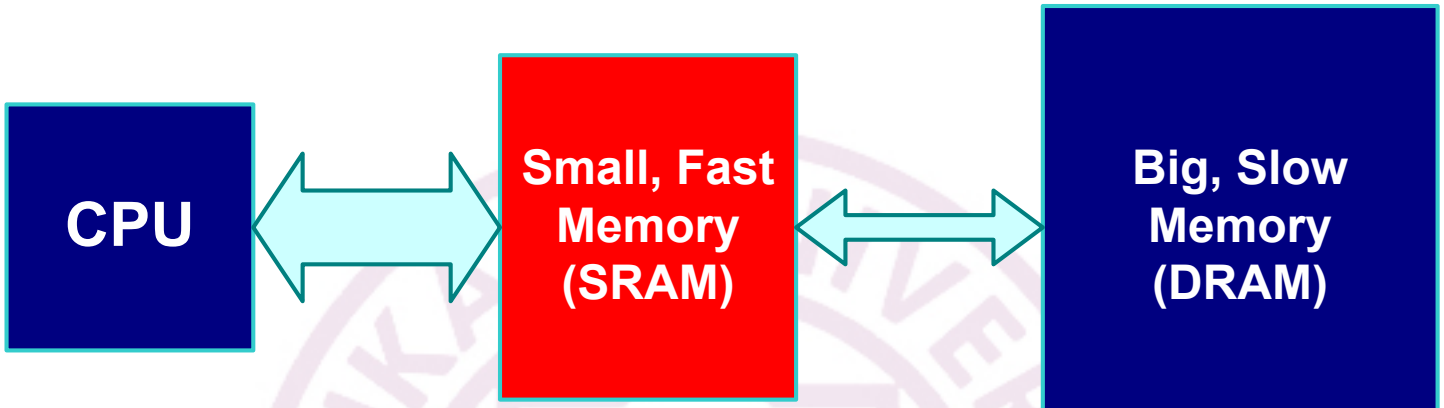
- 性能差距：处理器访存请求时间和DRAM访问延时时间



# 存储层次的重要性

- 对多核处理器来说，存储层次变得更加重要。
- 总峰值带宽随处理器核数目增加而增加
  - Intel Core i7 处理器每核每周期发出两次访存请求
  - 假设：四个处理器核和3.2GHz时钟频率
    - 每秒 256 亿 64 位数据访问 + 每秒 128 亿 128 位指令访问 = 409.6 GB/s
    - DRAM带宽只有期望值的 **8%**（34.1GB/s）
  - 解决图形
    - 多端口，流水Cache
    - 每核两级Cache
    - 片上共享第三级Cache
- 在高端微处理器芯片中，一般 Cache 容量大于 10MB。
  - 消耗大量的芯片面积和功率预算

# 高速缓冲存储器 Cache



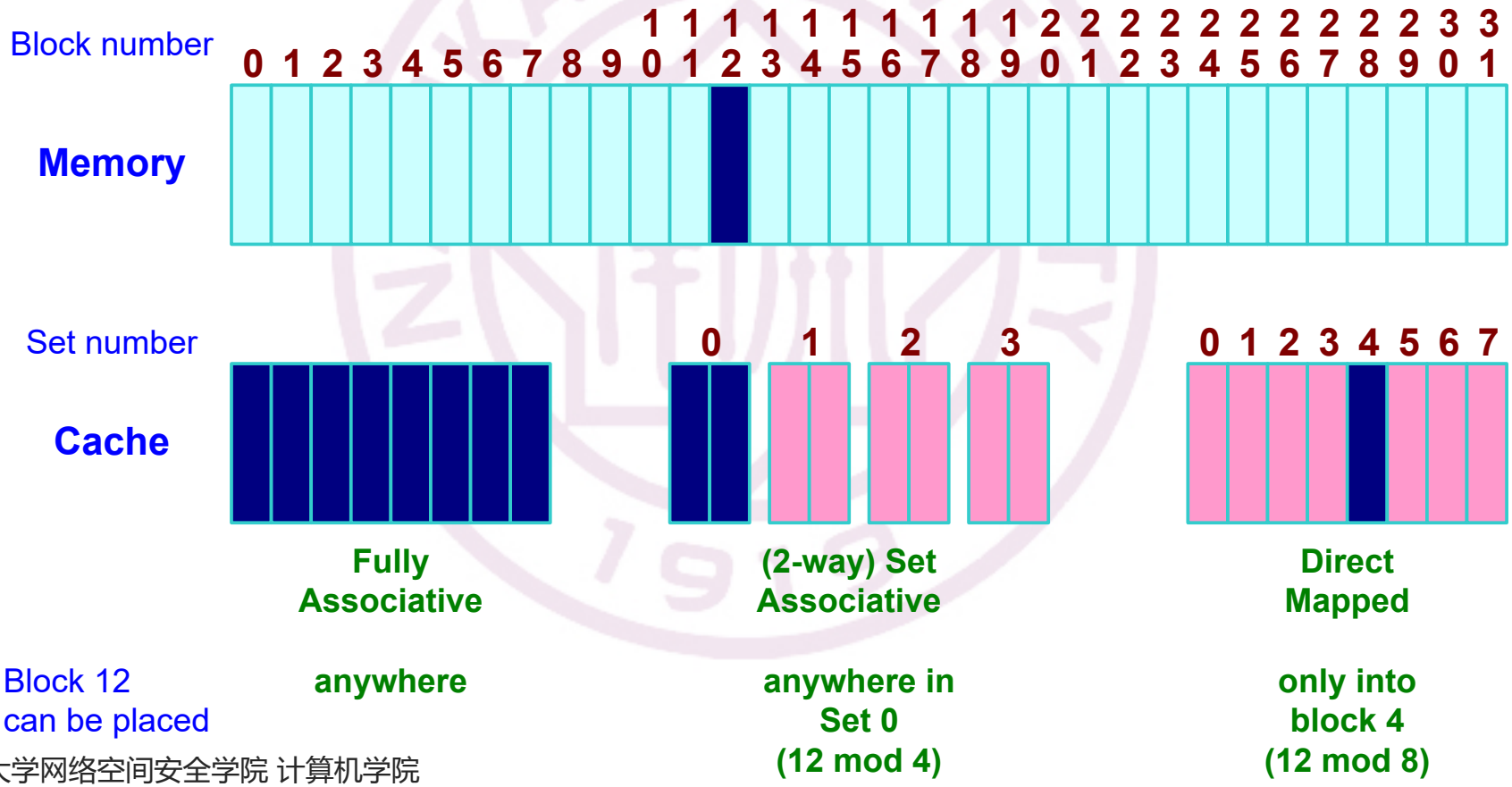

- 当数据在Cache中时，称为 Cache 命中；
- 当数据不在 Cache 中时，称为 Cache 未命中。
  - CPU 停止执行
  - 硬件将未命中数据块从主存移动到Cache中；
  - 如果数据不在主存中，则继续到下一个存储层次——虚存中寻找。
    - 主存与虚存的关系与Cache和主存的关系是一样的

# 四个问题

- 块放置(block placement)：数据块放在哪里?
- 块辨识(block identification)：如何发现想要的数据块?
- 块替换(block replacement)：在读入新数据块时替换哪个现有的数据块?
- 写策略(write strategy)：如何修改数据?

# 块放置

- 直接映射(direct mapped)：数据块与位置一一对应。
- 全关联(fully associative)：数据块可以放在任意位置
- 组关联(set associative)：数据块可以放在几个预定位置

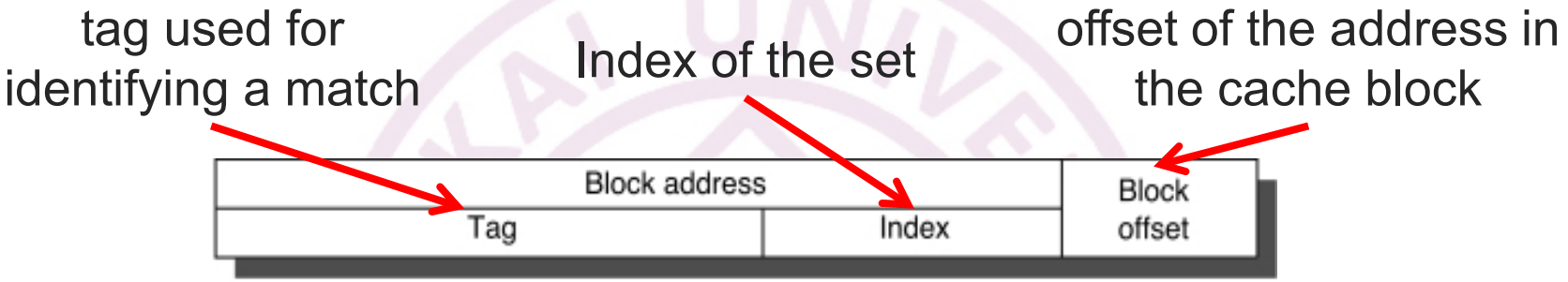


# 块放置

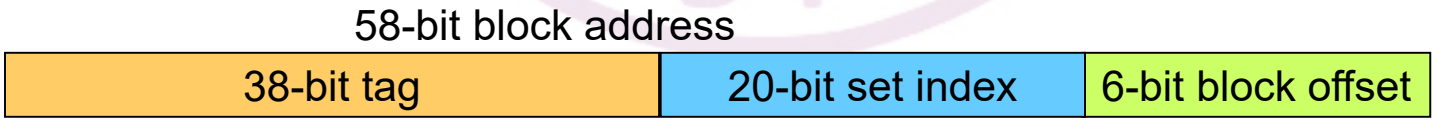
- 对于 $n$ 路组关联方式，$n$ 值越大，
  - 实现代价越高
    - 绝大多数cache采用 1 路（直接映射）、2 路或 4 路组关联
  - 振荡的可能性越低
    - 振荡：两个块竞争同一个块帧地址，并交替地循环访问。

# 块辨识

- 问题：已知存储地址，如何在 Cache 中找到对应的数据?
- 将存储地址分成三个部分




- 某个计算机系统
  - 地址空间 $2^{64}$ 字节（64 位地址线）
  - 块大小为 64 字节
  - Cache 容量 64MB ( $2^{26}$ 字节，$2^{20}$ 块)
  - 直接映射方式



# 块替换

- 当想要访问的数据块未命中时，cache控制器必须做出反应。
  - 对于直接映射方式，将使用从主存中读取的新数据块覆盖未命中数据块；
  - 对于 $n$ 路组关联方式，必须对 $n$ 个可能的被覆盖数据块位置进行选择。

- 三种选择策略
  - 随机：实现容易
  - 先进先出(First-In-First-Out, FIFO)：实现比较困难
  - 最近最少使用(Least Recently Used, LRU)：Cache容量越大，实现越困难。（思考题）

# 写策略

- 现象：读操作比写操作多
- 基本原理：提高经常情形——读操作的性能
  - Cache设计者花费绝大多数努力来加快读操作，而对写操作关注较少。
  - 根据 Amdahl 定律，如果写操作较慢，则整个系统的性能将较差。
  - 必须努力提高写操作性能。
- 写策略
  - 写透(write through)：新数据被同时写入Cache数据块和低一级存储器
  - 写回(write back)：
    - 如果写命中，则新数据只写入Cache数据块；
    - 如果写不命中，则首先将被替换数据块“写回”低一级存储器，然后新数据写入到相应的Cache数块。
- “dirty”位：表示被替换Cache数据块是否需要“实际写回”低一级存储器。

# 写策略比较

- 写透
  - 实现更加容易
  - 低一级 Cache 中总是保存数据的最新版本，简化了数据一致性问题。

- 写回
  - 因为写操作速度决定于 Cache，而不是低一级存储器的写速度，故写操作速度快。
  - 因为对同一 Cache 数据块的多次写操作被捆绑到一起写回低一级存储器，需要的存储器带宽较小，故写操作效率高。

# 写不命中处理方式

- 写分配(write allocate)
  - 在写不命中时分配 Cache 数据块
  - 数据块分配后是标准的写命中操作
  - 写不命中等价于读不命中
  - 与写回策略配合很好

- 非写分配(no-write allocate)
  - 写不命中不分配 Cache 数据块
  - 只更新低一层存储器
  - 只在读不命中时分配 Cache 数据块
  - 与写透策略配合很好

# 存储层次基础

- 未命中率(miss rate)：Cache 访问中未命中访问的比例
  - 局部未命中率(local miss rate)：在该 Cache 中未命中访问的数目与对该 Cache 访问的总数之比
  - 全局未命中率(global miss rate)：在该 Cache 中未命中访问的数目与处理器发出的访问总数之比

- 平均存储访问时间(average memory access time)

$$\overline{access\ time} = Hit\ time_{L1} + Miss\ rate_{L1} \times Miss\ penalty_{L1}$$

$$Miss\ penalty_{L1} = Hit\ time_{L2} + Miss\ rate_{L2} \times Miss\ penalty_{L2}$$

# 存储层次基础

- 每指令未命中(miss per instruction)

$$\frac{Misses}{Instruction} = \frac{Memory\ accesses}{Instruction\ count} \times \frac{Miss\ of\ accesses}{Memory\ accesses}$$

$$= Miss\ rate \times \frac{Memory\ accesses}{Instruction\ count}$$

  - 优点：独立于硬件实现；缺点：与体系结构有关
  - 由于我们是计算机体系结构课，所以……

- 未命中类型
  - 强制性：对数据块的第一次访问
  - 容量：前期被替换且稍后又被访问的数据块
  - 冲突：程序重复访问处于多个存储地址的不同数据块，而这些数据块恰恰被映射到 Cache 中相同位置
  - 一致性：多处理器中多个 Cache 内容不一致

# 六种基本Cache优化方法

- 加大数据块容量
  - 减少强制性不命中
  - 增加容纳能力和冲突不命中，增加不命中惩罚
- 加大Cache总容量
  - 降低未命中率
  - 增加命中时间，增加功耗
- 提高关联性
  - 减少冲突不命中
  - 增加命中时间，增加功耗
- 增加Cache级数
  - 减少总的存储器访问时间

# 六种基本Cache优化方法

- 读操作未命中优先于写操作
  - 减少未命中惩罚
- 避免Cache索引中的地址转换
  - 减少命中时间

# 存储器性能和类型

- 性能指标
  - 延时：Cache关心的问题
  - 带宽：多处理器和 I/O 关心的问题
  - 访问时间(access time)
    - 从发出读请求到访问存储字到达间隔的时间
  - 周期时间(cycle time)
    - 不相关存储请求之间的最小间隔时间

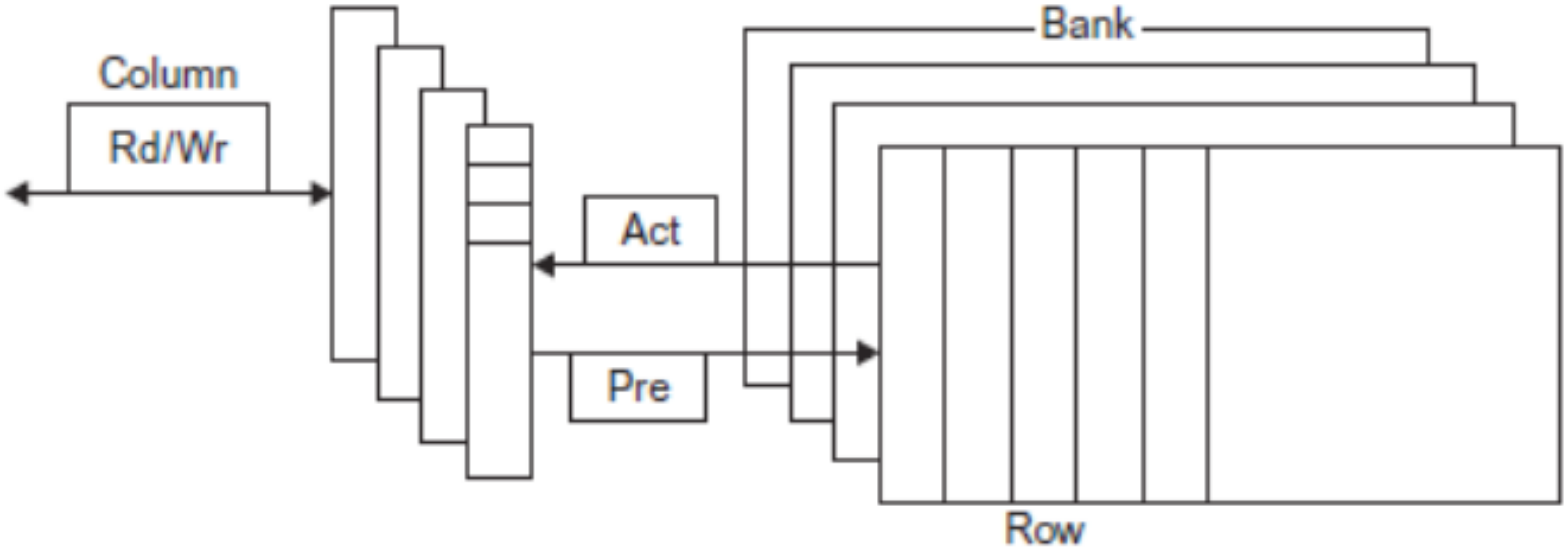
- 存储器类型
  - SRAM：低延时，用于集成在存储器芯片上的 1~3 级Cache
  - DRAM：将芯片组织为许多高带宽的存储体(memory bank)，用于主存

# 存储技术

- SRAM
  - 保存数据所需功率低
  - 访问时间接近周期时间
  - 每个比特位需要 6 个晶体管
- DRAM
  - 读操作之后必须重写(re-written)
  - 必须周期性刷新(refreshed)
    - 大约每 8ms 刷新一次（大约占 5% 的时间）
    - 每行可以同时刷新
  - 每个比特位需要 1 个晶体管
  - 地址线复用
    - 地址线高半部分：行访问选通(Row Access Strobe, RAS)
    - 地址线低半部分：列访问选通(Column Access Strobe, CAS)

# DRAM 内部组成

- 存储体数目上限为 16 (DDR4)
- 数据访问
  - 发出ACT命令
  - 打开一个存储体和一行且将该行加载到行缓冲器
  - 根据后续列地址，以任意DRAM宽度传输，或通过起始地址进行块传输
  - 预充电命令(PRE)关闭存储体和行，为新的访存做好准备

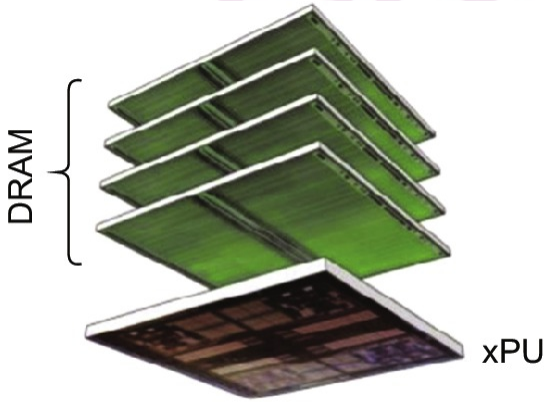


# 存储器性能的优化方法

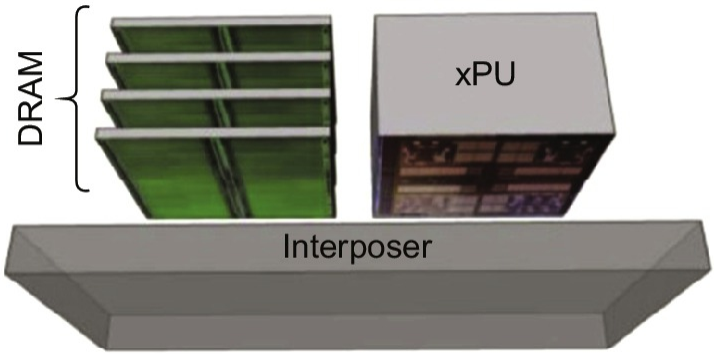
- 对相同行缓存多个列访问
- 同步DRAM(SDRAM)
  - 在DRAM接口中添加时钟，对指令进行流水操作
  - 关键存储字优先的突发模式
- 更宽的接口（4 比特，8 比特，16 比特总线）
- 双倍数据速率(Double Data Rate, DDR)
  - 在存储器时钟的上升沿和下降沿均传输数据
- 每个DRAM设备上存在多个存储体（重叠访问）

# 图形数据RAM

- Graph Data RAM 的带宽是 DDR3 的 2~5 倍
  - 更宽的接口（32 位 vs. 16 位）
  - 更高的时钟频率（可能原因是：DIMM模块使用焊接代替插座连接）
- 堆叠或嵌入DRAM
  - 与处理器封装在一起，降低访问延迟、增加带宽（2017年的创新）
  - 高带宽存储器(High Bandwidth Memory, HBM)




Vertical stacking (3D)




Interposer stacking (2.5D)

- 垂直堆叠(Vertical stacking)
- 介入堆叠(Interposer stacking)

# 术语

- 国际电工委员会制定

| 十进制单位 | | | 二进制单位 | | |
|---|---|---|---|---|---|
| 名称 | 缩写 | 次方 | 名称 | 缩写 | 次方 |
| kilobyte | KB | 10^3 | kibibyte | KiB | 2^10 |
| megabyte | MB | 10^6 | mebibyte | MiB | 2^20 |
| gigabyte | GB | 10^9 | gibibyte | GiB | 2^30 |
| terabyte | TB | 10^12 | tebibyte | TiB | 2^40 |
| petabyte | PB | 10^15 | pebibyte | PiB | 2^50 |
| exabyte | EB | 10^18 | exbibyte | EiB | 2^60 |
| zettabyte | ZB | 10^21 | zebibyte | ZiB | 2^70 |
| yottabyte | YB | 10^24 | yobibyte | YiB | 2^80 |

# Flash存储器

- 类型：电可擦可编程只读存储器(Electrically Erasable Programmable Read-Only Memory, EEPROM)
- 种类
  - NAND：密度更高，适合大容量非易失存储器
    - 编程速度快，擦除时间短
  - NOR：速度更快
    - 随机存取和对字节执行写操作
- 活跃研究方向：相变存储器(Phase-Change Memory, PCM)
  - 原理：使用加热元件使衬底状态在晶体和非晶体之间改变，由此具有不同的电阻特性
  - 相比 NAND：
    - 写性能提高大约 10 倍
    - 读性能提高大约 2 倍

# NAND Flash 存储器简介

- 顺序读取一整页（0.5~4KiB）
- 首字节 25μs，后续字节 40 MiB/s
- SDRAM：首字节 40ns，后续字节 4.8 GB/s
- 2KiB传输：75μs（SDRAM为 500ns），慢 150 倍
- 比磁盘快 300~500 倍
- 覆盖写之前必须先擦除（以块为单位）
- 非易失，可以零功率使用
- 写次数有限（每块至少 10 万次，尽量均匀访问）
- 价格 $2/GiB
  - SDRAN存储器：$20~40/GiB
  - 磁盘：$0.09/GiB

# 内存可靠性

- 存储器易受宇宙射线影响
- 软错误(soft errors)：动态错误
  - 通过纠错码(ECC)检测和修复
- 硬错误(hard errors)：永久性错误
  - 使用备用行替换缺陷行
- Chipkill：类似RAID磁盘的错误恢复技术

# Cache性能的高级优化方法

- 性能指标：平均存储器访问时间

$$Average\ memory\ access\ time = Hit\ time + Miss\ rate \times Miss\ penalty$$

  - 三要素：命中时间，未命中率和未命中惩罚

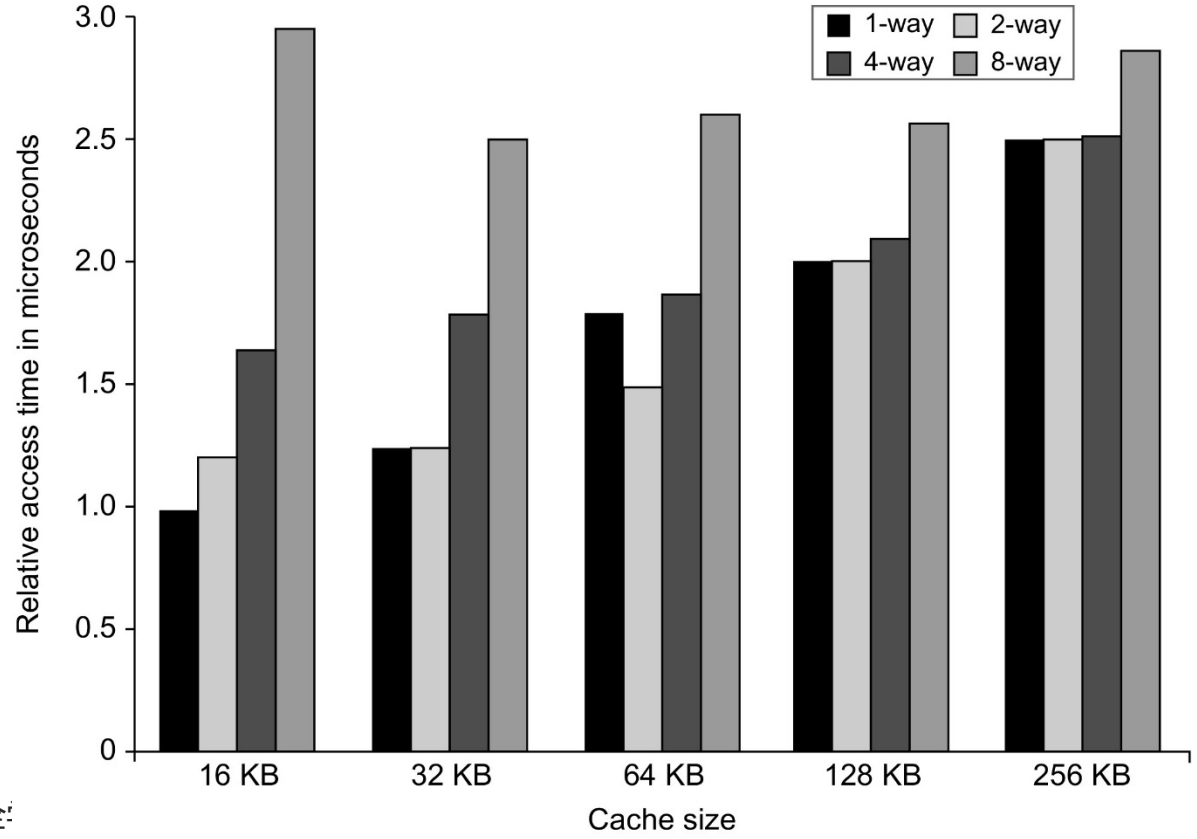
- 减少命中时间
  - 容量小且简单的第一级Cache
  - 路(way)预测
  - 降低能耗
- 增加 Cache 带宽
  - 流水Cache，多体Cache，非阻塞式Cache
    - 非阻塞式 Cache (non-blocking cache)：在发生Cache访问未命中且没有清除该未命中访问的情况下，处理器无需停顿而继续进行缓存访问。
  - 对功耗影响不同

# Cache性能的高级优化方法

- 降低未命中惩罚
  - 关键字优先
  - 合并写缓存区
  - 对功率影响很小
- 降低未命中率
  - 编译器优化
  - 改进能耗
- 通过并行化降低未命中惩罚或未命中率
  - 硬件预取或编译器预取
  - 增加功耗

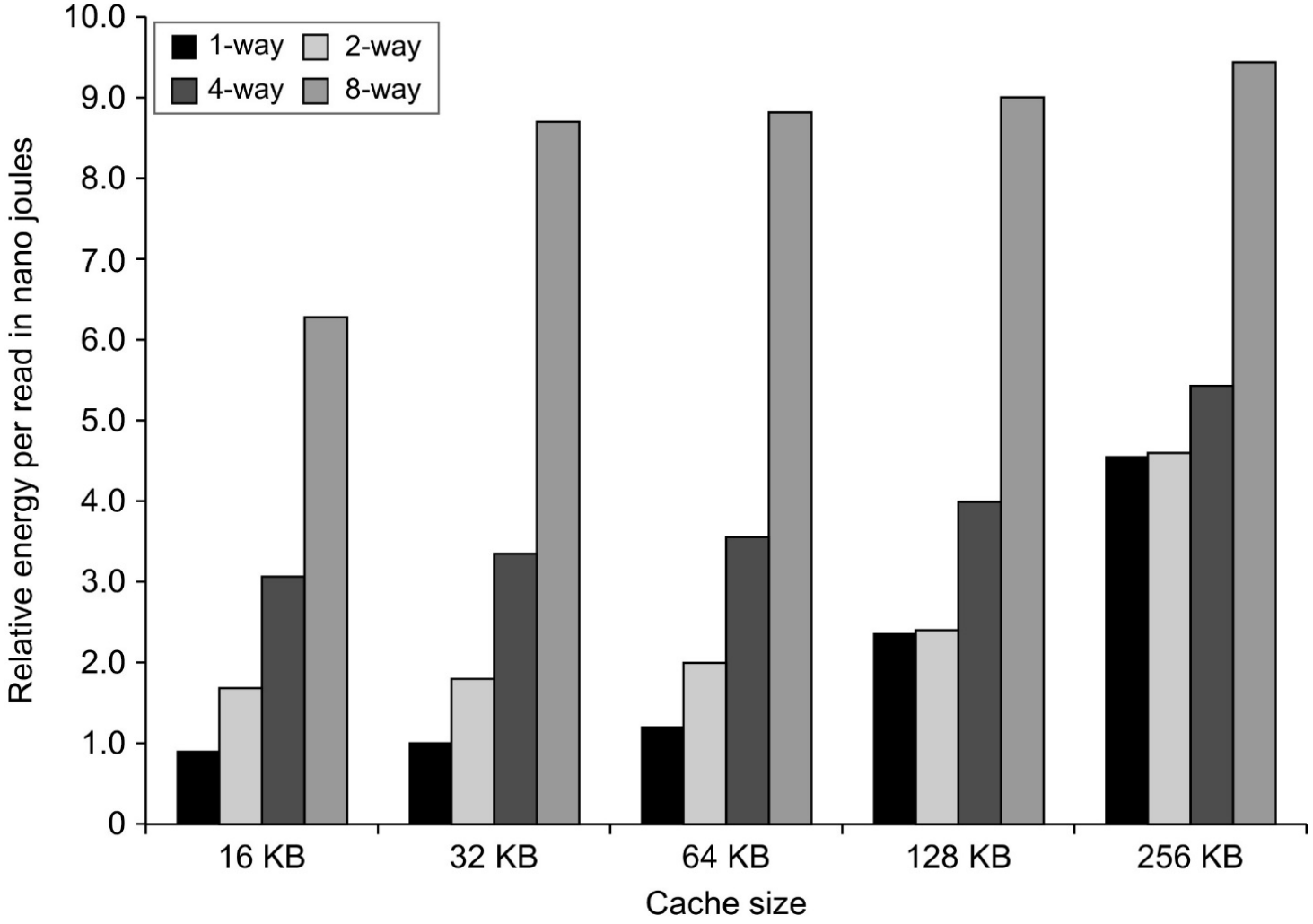
# 小容量和简单的第一级Cache

- 面对快速时钟周期和功率限制的双重压力
  - 限制第一级Cache容量
  - 使用较低关联度
- 随着Cache容量和关联度增加，相对访问时间也增加



# 小容量和简单的第一级Cache

- 随着Cache容量和关联度增加，读操作能耗增加



# 路(way)预测

- 降低冲突未命中，保持直接映射Cache的命中速度
- 通过预测，提前设置多路转换器来选择所需的Cache块
  - 每个Cache块添加一个块预测位
  - 预测错误将带来更长的命中时间（每次失效额外增加一个时钟周期延迟）
  - 预测准确率
    - 两路：＞90%
    - 四路：＞80%
    - 指令Cache的预测准确率比数据Cache高
  - 上世纪 90 年代中期首先用于MIPS R10000处理器
  - 用于ARM Cortex-A8处理器
- 扩展：块(block)预测
  - “路选择”
  - 增加预测错误惩罚

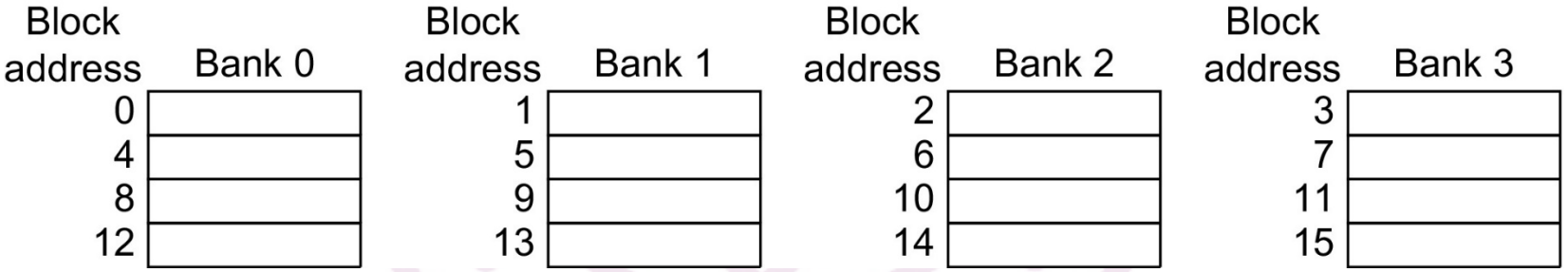
# 流水Cache

- 流水Cache访问以增加带宽
  - 主要用于L1 Cache
  - 例
    - Pentium：1 周期
    - Pentium Pro – Pentium III：2 周期
    - Pentium 4 – Core i7：4 周期
- 增加分支指令的错误预测惩罚
- 更容易增加关联性

# 多体Cache

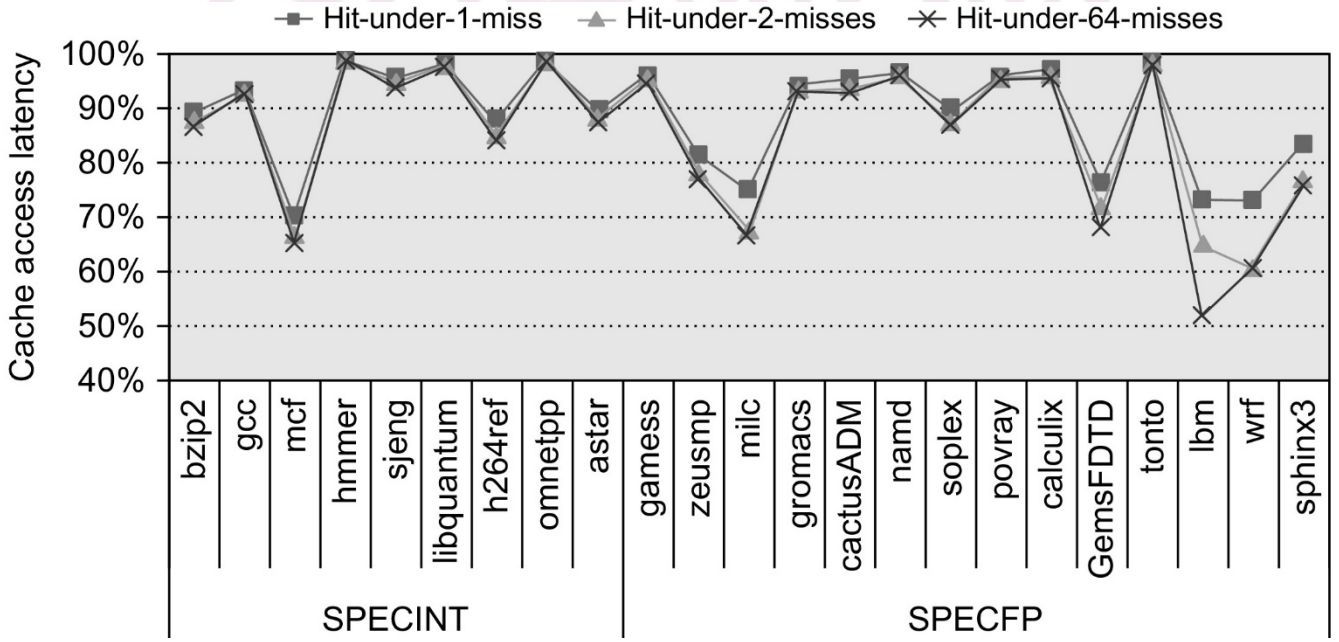
- 将Cache组织成若干个独立体(bank)
  - 满足每个周期多个数据Cache访问（超标量处理器）
  - ARM Cortex-A8 L2 Cache支持 1~4 个体
  - Intel i7 L1 Cache支持 4 个体、L2 Cache 8 个体

- 按照块地址交叉访问体



# 非阻塞式Cache

- 数据 Cache 访问不命中不会暂停乱序执行处理器的执行
  - 等待数据Cache返回未命中数据时，继续从指令Cache中取指令。
    - 在未命中下命中(hit under miss)：降低未命中惩罚
    - 在多未命中下命中(hit under multiple miss) 或者 在未命中下未命中(miss under miss)：进一步降低未命中惩罚
- 性能评估

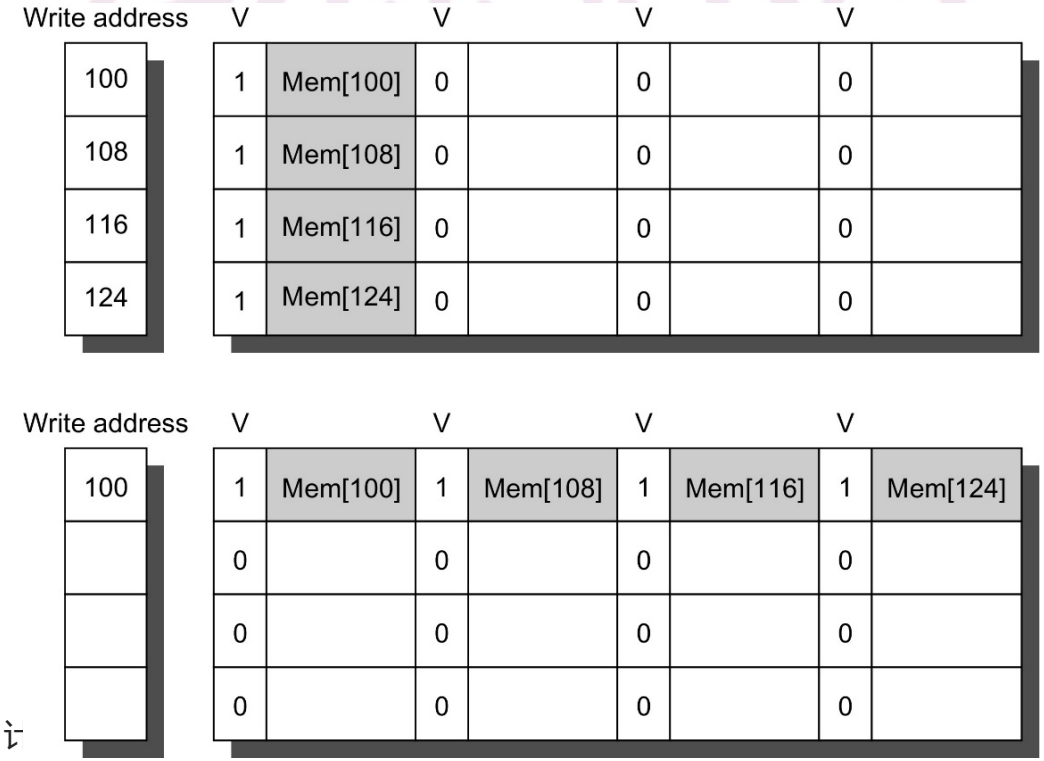


# 关键字优先，提前重启

- 关键字优先
  - 先从下一级存储器中请求未命中数据字，一旦获得马上发给处理器
  - 在填充数据块内其它数据字的同时，处理器连续执行
- 提前重启
  - 按正常顺序请求数据字
  - 一旦获得包括请求数据字的数据块，马上将请求数据字发给处理器
- 有效性取决于
  - Cache块容量
  - 对Cache块中还未读入部分进行另一次访问的可能性

# 合并写缓存

- 合并写缓存
  - 写缓存器中包含其它被修改的 Cache 数据块
  - 新写入 Cache 数据块地址与写缓存器中已有的 Cache 数据块地址相同，则二者合并
- 无合并写与合并写示意图



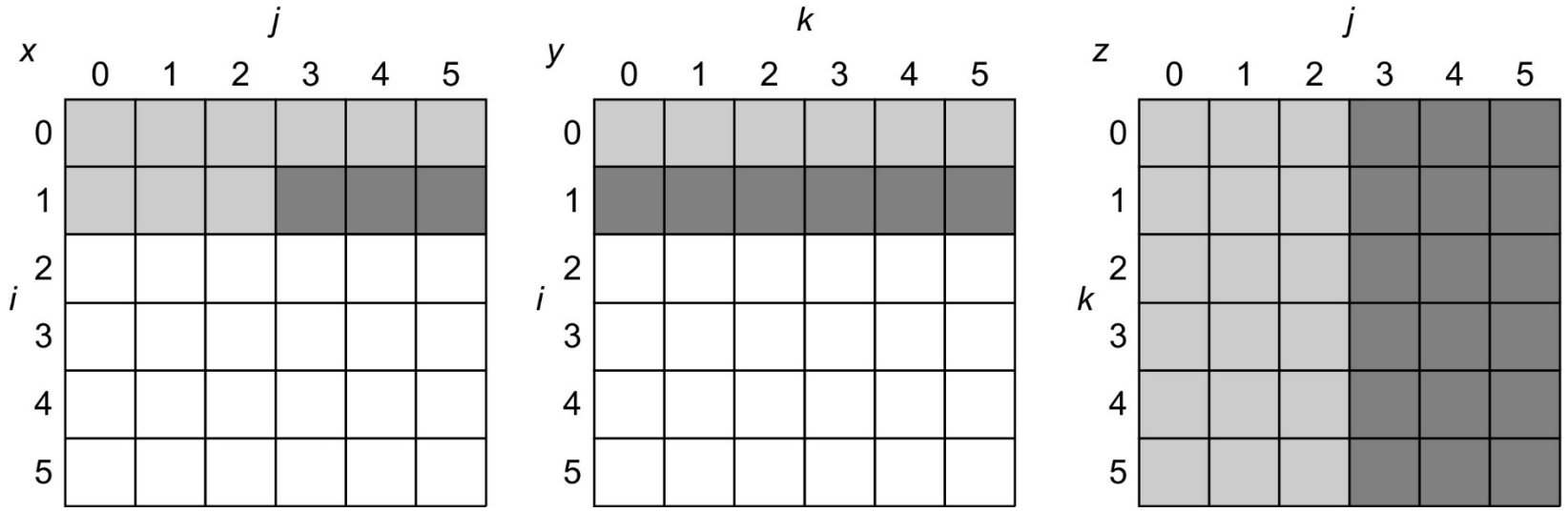
# 编译器优化

- 循环交换
  - 交换嵌套循环次序，以便可以按顺序访问存储器。
- 分块
  - 将矩阵细分为子矩阵来代替访问整行或整列
  - 虽然需要更多次内存访问，但提高了访存的局部性

# 编译器优化方法：分块

- 例：矩阵乘法

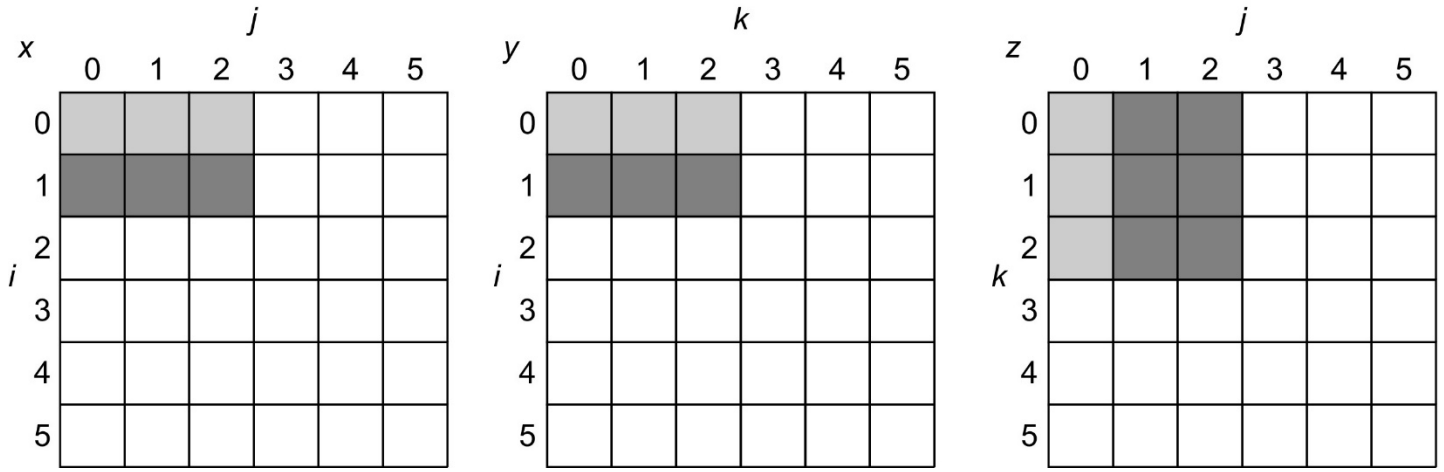
```
for(i=0;i<N;i=i+1)
  for(j=0;j<N;j=j+1){
    r=0;
    for(k=0;k<N;k=k+1)
      r=r+y[i][k]*z[k][j];
    x[i][j]=r;
  };
```



# 编译器优化方法：分块

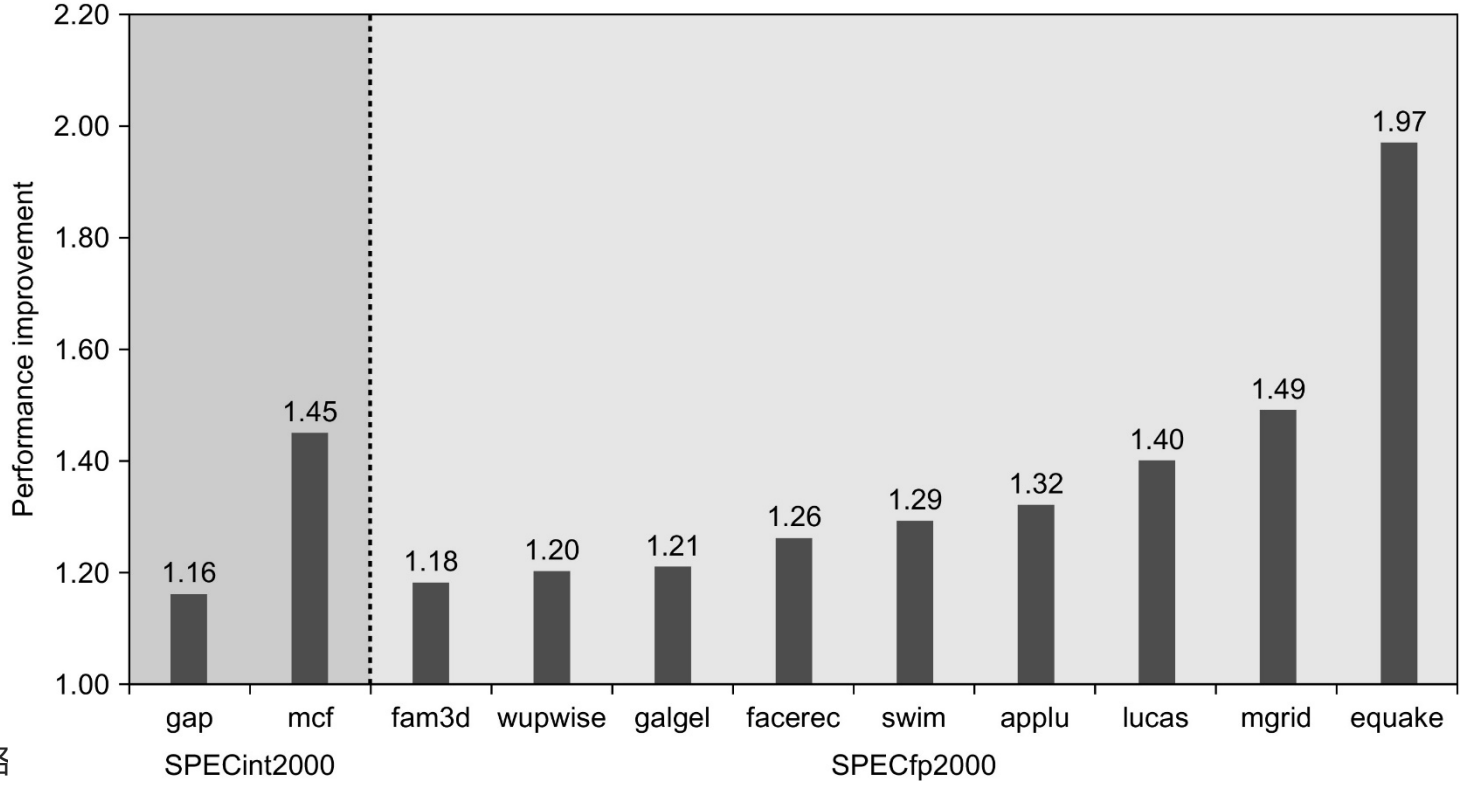
- 例：矩阵乘法

```
for(jj=0;jj<N;jj=jj+B)
  for(kk=0;kk<N;kk=kk+B)
    for(i=0;i<N;i=i+1)
      for(j=jj;j<min(jj+B,N);j=j+1){
        r=0;
        for(k=kk;k<min(kk+B,N);k=k+1)
          r=r+y[i][k]*z[k][j];
        x[i][j]=x[i][j]+r;
      };
```



# 硬件预取指令和数据

- 预先将处理器需要的指令和数据读入Cache
  - 未命中时，读取
    - 未命中块：放入Cache
    - 下一个相邻块：放入流缓存区，称为“预取”
- 基准程序在Intel Pentium 4上通过硬件预取获得的加速比



# 编译器控制预取

- 在处理器需要数据之前，通过插入预取指令请求数据
  - 寄存器预取：将数值加载到寄存器
  - Cache预取：将数据只加载到Cache而不是寄存器

- 与“循环展开”和“软件流水”相结合

# 使用高带宽存储器(HBM)扩展存储层次

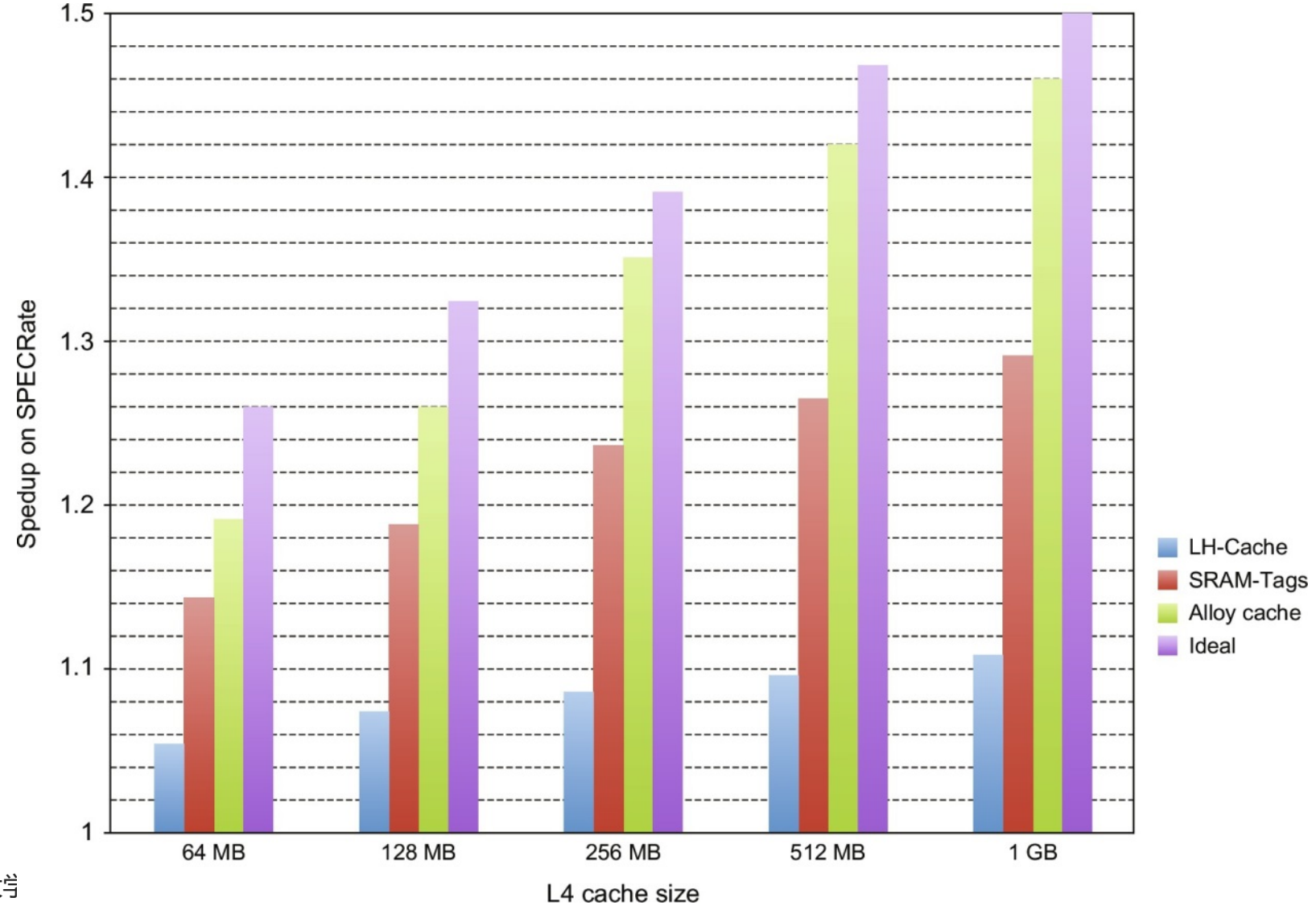
- 使用片内封装DRAM来构建大规模L4 Cache
  - 容量：128MiB~1GiB，远大于片上L3 Cache容量
  - 标签(tag)存储问题
    - 较小容量Cache块需要大量静态存储器存储标签
    - 较大容量Cache块可能效率低（存储不需要的内容和造成更多未命中）
- LH-Cache：由Loh和Hill在2011年提出
  - 每个SDRAM行是一个块索引
  - 每行包括一组标记和 29 个数据段
  - 29 组关联
  - 命中需要列访问选通（CAS）
- Alloy Cache：采用掺入（合金）技术
  - 将标记和数据封在一起
  - 使用直接映射

# 使用高带宽存储器(HBM)扩展存储层次

- 在非命中情形下，两个方案都需要两次完整的DRAM访问
  - 两种解决方法
    - 使用图(map)来保存Cache块出现（非位置）的踪迹
    - 使用存储器访问预测方法预测可能的未命中

# 使用高带宽存储器(HBM)扩展存储层次

- 内存密集型基准程序的加速比



# 诚信 创新 实践

# 虚拟内存和虚拟机

- 聚焦：共享相同处理器的进程之间的保护和隐私问题
  - 详细内容关注《操作系统》课程相关部分
- 通过虚拟内存保护
  - 页式虚存，包括缓存页表条目的快表(TLB)，构成进程保护的主要机制
  - 将进程保存在它自己的存储空间中
- 体系结构的作用
  - 提供用户模式和管理模式
  - 保护某些CPU状态
  - 提供在用户模式和管理模式之间切换的机制
  - 提供限制存储访问的机制
  - 提供转换查找缓存器(Translation Lookaside Buffer, TLB)完成虚拟地址到物理地址之间的转换

# 谬误和陷阱

- 谬误：使用一个程序预测另一个程序的泛化Cache性能
  - 不明智的。如商用数据库程序在大容量第二级Cache中也存在明显的不命中率，而不是像SPECCPU程序给出的那样。

- 陷阱：模拟足够多的指令以获得对存储层次的准确性能度量
  - 使用较小实验数据预测大容量Cache的性能
  - 在程序整个运行中局部性行为不是常数
  - 程序的局部性行为随着输入变化而变化

- 陷阱：基于Cache的系统中不提供高存储带宽
  - Cache有助于平均缓存延时，但不能给必须访问主存的程序提供高存储带宽。
  - 架构师必须在Cache后面提供高带宽存储器解决该问题