中国科学院大学
University of Chinese Academy of Sciences

Loongson

高级计算机系统结构

沈海华

shenhh@ucas.ac.cn

第三讲 存储系统设计：高速缓存

- 存储层次结构
  - 寄存器
  - Cache&TLB
  - Memory
  - Disk&Flash
  - 网络存储
- 高速缓存结构
- 高速缓存优化策略

局部性原理：存储层次结构的基础

- 局部性原理：
  - 程序通常只访问地址空间中相对较小的一部分
- 局部性的两种类型：
  - 时间局部性，Temporal Locality (Locality in Time): 如果一个单元被访问，很可能不久它将再次被访问(例如：循环、重用)
  - 空间局部性，Spatial Locality (Locality in Space): 如果一个单元被访问，靠近它的那些单元很可能不久被访问(例如：线性代码，数组访问)

访问局部性

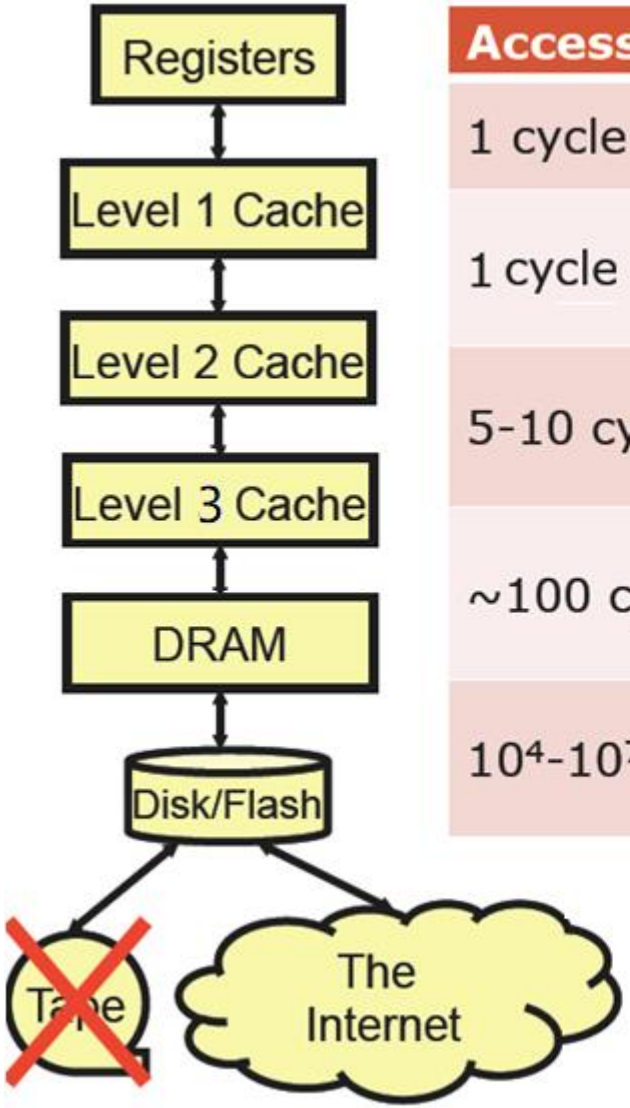
- **访问局部性**是程序本身的特性，在机器设计中被采用
- 过去20年，硬件依赖局部性来提高访问速度

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

例子中的局部性：

- 数据
  - 依次访问数组元素 (空间局部性)
- 指令
  - 依次访问指令 (空间局部性)
  - 通过循环重复指令 (时间局部性)

存储层次结构（Everything is a Cache for the next level）



Access Time	Capacity	Managed by
1 cycle	~500B	Software/compiler
1 cycle	~64KB	Hardware
5-10 cycles	1-10MB	Hardware
~100 cycles	~10GB	Software/OS
10^4-10^7 cycles	~1TB	Software/OS

存储器层次结构管理

	Reg	L1 Inst	L1 Data	L2	L3	DRAM	Disk
Size	1~4K	64K	64K	512K	32M~64M	1G	100G
Latency Cycles, Time	1, 0.6 ns	1, 1.9 ns	1, 1.9 ns	5~10, 6.9 ns	40, 15ns	120, 55 ns	10^7, 12 ms

理想目标：

让程序的寻址空间可以扩展到硬盘大小，访问速度感受如同寄存器

第三讲 经典技术回顾：高速缓存

- 存储层次结构
  - 寄存器
  - Cache&TLB
  - Memory
  - Disk&Flash
  - 网络存储

寄存器

- 逻辑寄存器

-由ISA定义的程序员可见的寄存器结构，例如：通用寄存器、浮点寄存器等

- 物理寄存器

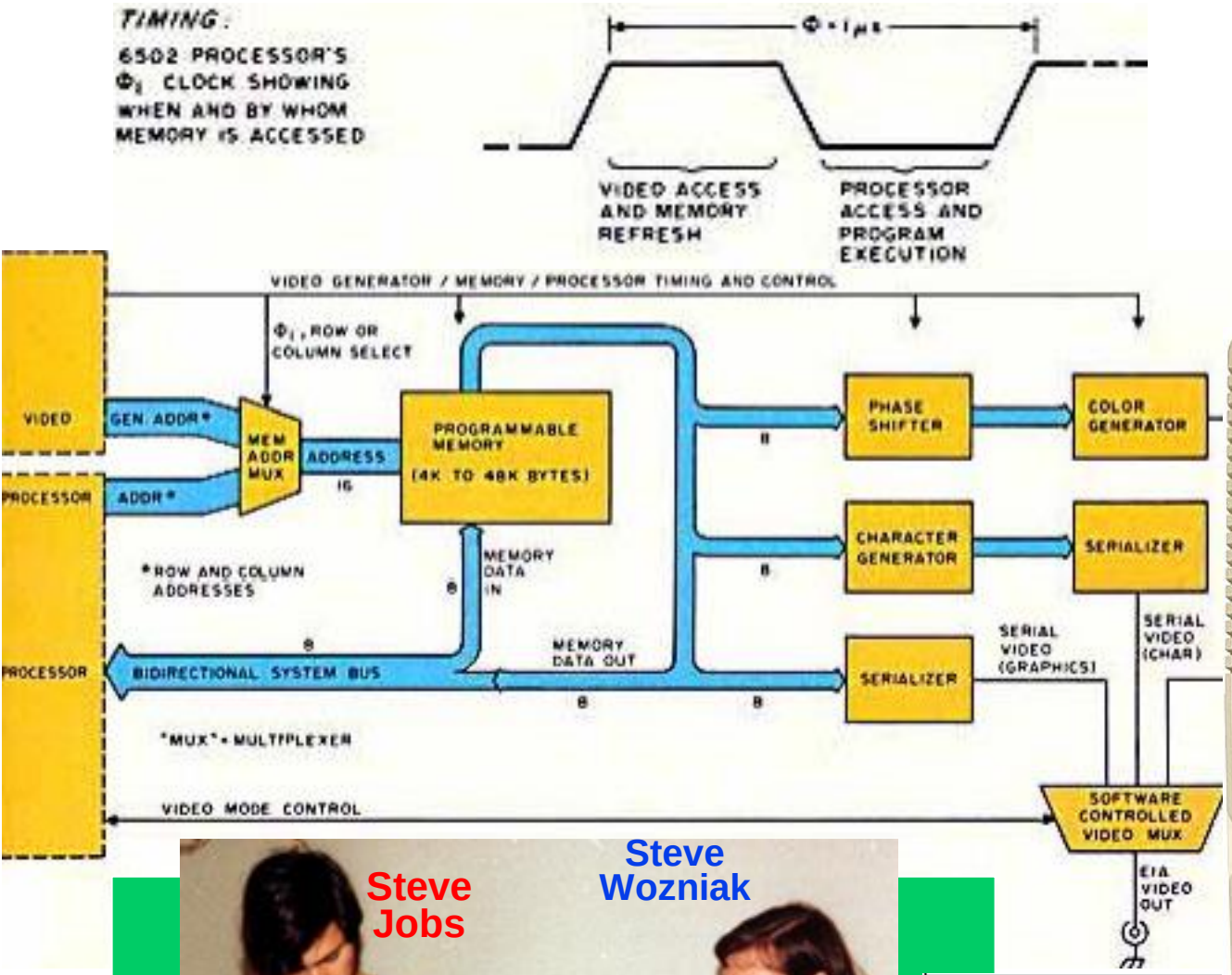
-物理寄存器堆通常由快速的静态随机读写存储器（SRAM）实现，这种RAM具有专门的读端口与写端口，可以支持多路并发访问。

-寄存器堆通常在CPU关键路径上，性能对CPU整体性能至关重要，实际设计中需要综合考虑寄存器堆大小、结构及端口数等，设计方法多采用全定制设计（full-custom design）。

高速缓存(Cache)

- 为什么需要设计Cache?
- Cache的经典组织结构
- Cache的操作策略
- Cache的性能
- Cache的功耗

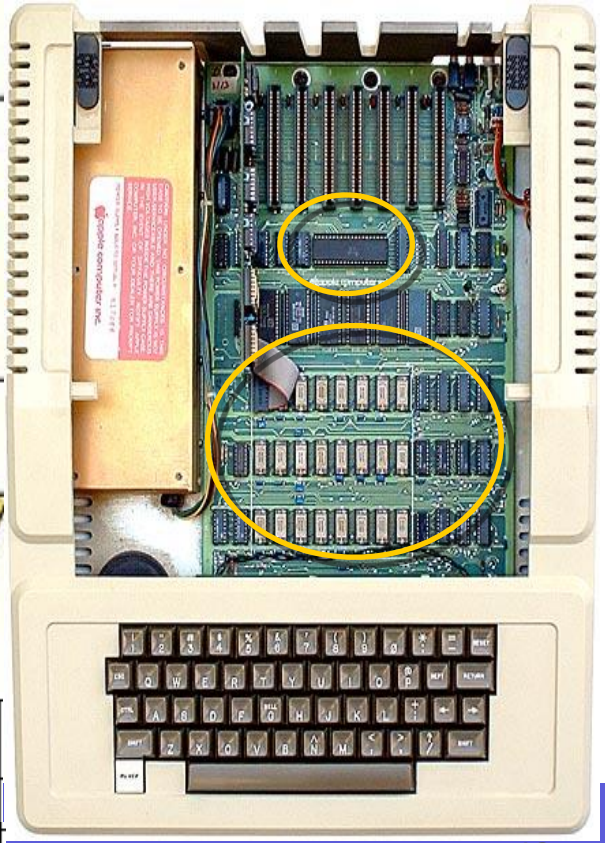
1977年：DRAM比microprocessors更快



Apple][(1977)

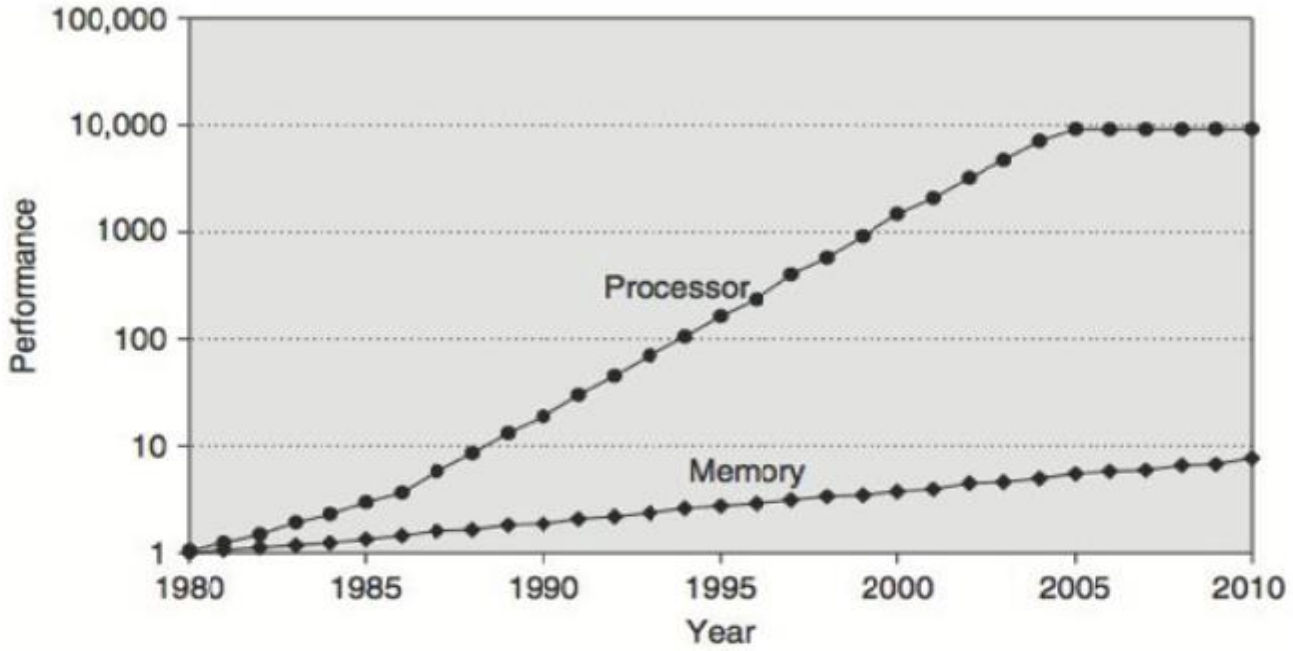
CPU: 1000 ns

DRAM: 400 ns



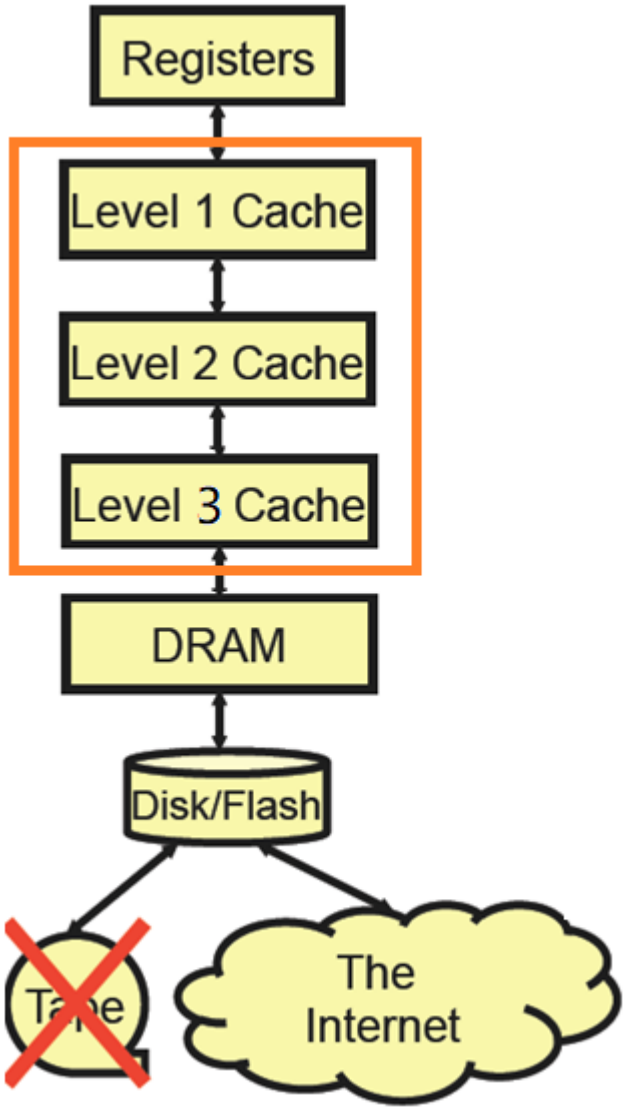
RAM Complement	Apple II System
4K	$ 1,298.00
48K	2,638.00

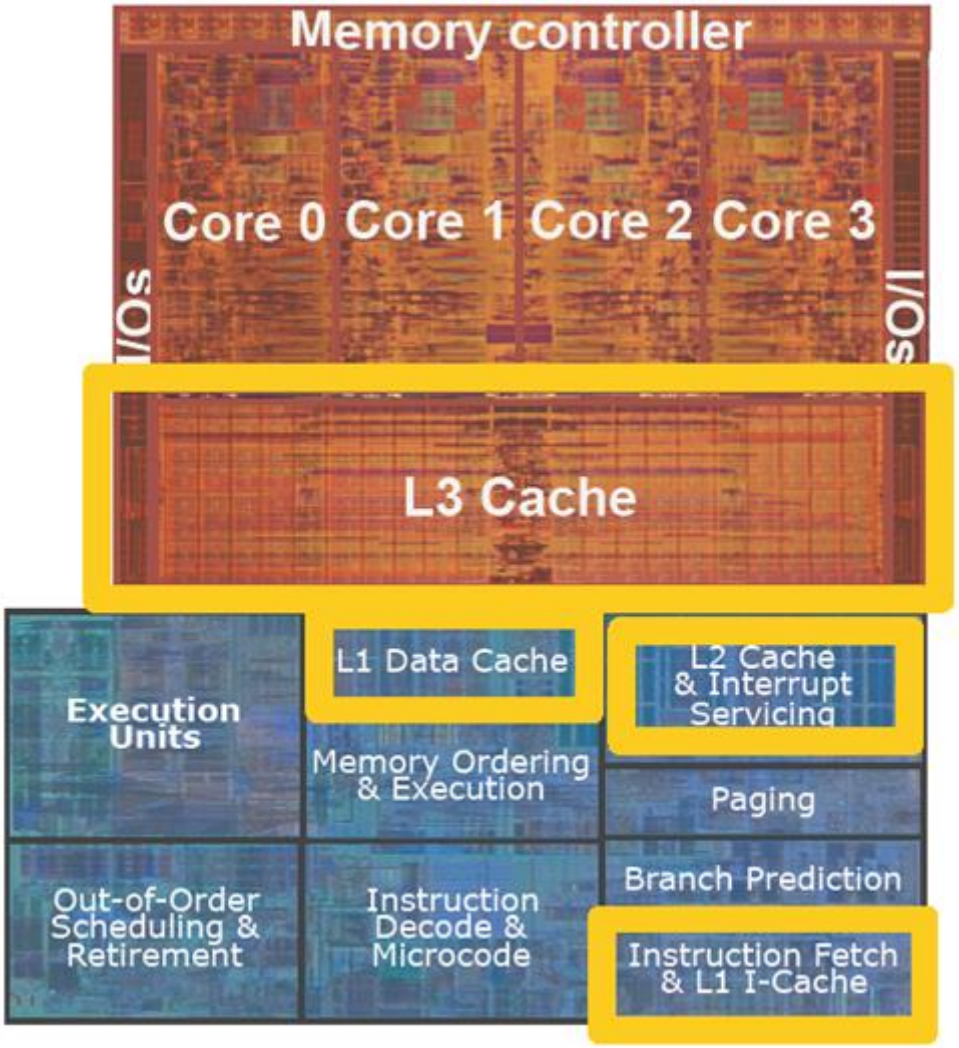
为什么要用Cache?



- Main memory (DRAM): Large & cheap but slow
  - A 1-cycle access in 1980 takes 100s of cycles in 2010
- Registers: fast but small and expensive
- We want: fast, large, and cheap memory

Caches


Registers
Level 1 Cache
Level 2 Cache
Level 3 Cache
DRAM
Disk/Flash
Tape
The Internet


Memory controller
Core 0 Core 1 Core 2 Core 3
I/Os
I/O
L3 Cache
Execution Units
L1 Data Cache
L2 Cache & Interrupt Servicing
Memory Ordering & Execution
Paging
Out-of-Order Scheduling & Retirement
Instruction Decode & Microcode
Branch Prediction
Instruction Fetch & L1 I-Cache

关于存储器层次结构的4个问题

- Q1: *数据块放置，Block placement：*

一个数据块可以放到上层存储器的哪个位置?

- Q2: *数据块识别，Block identification：*

如何发现一个数据块是否在上层存储器中?

- Q3: *数据块替换，Block replacement：*

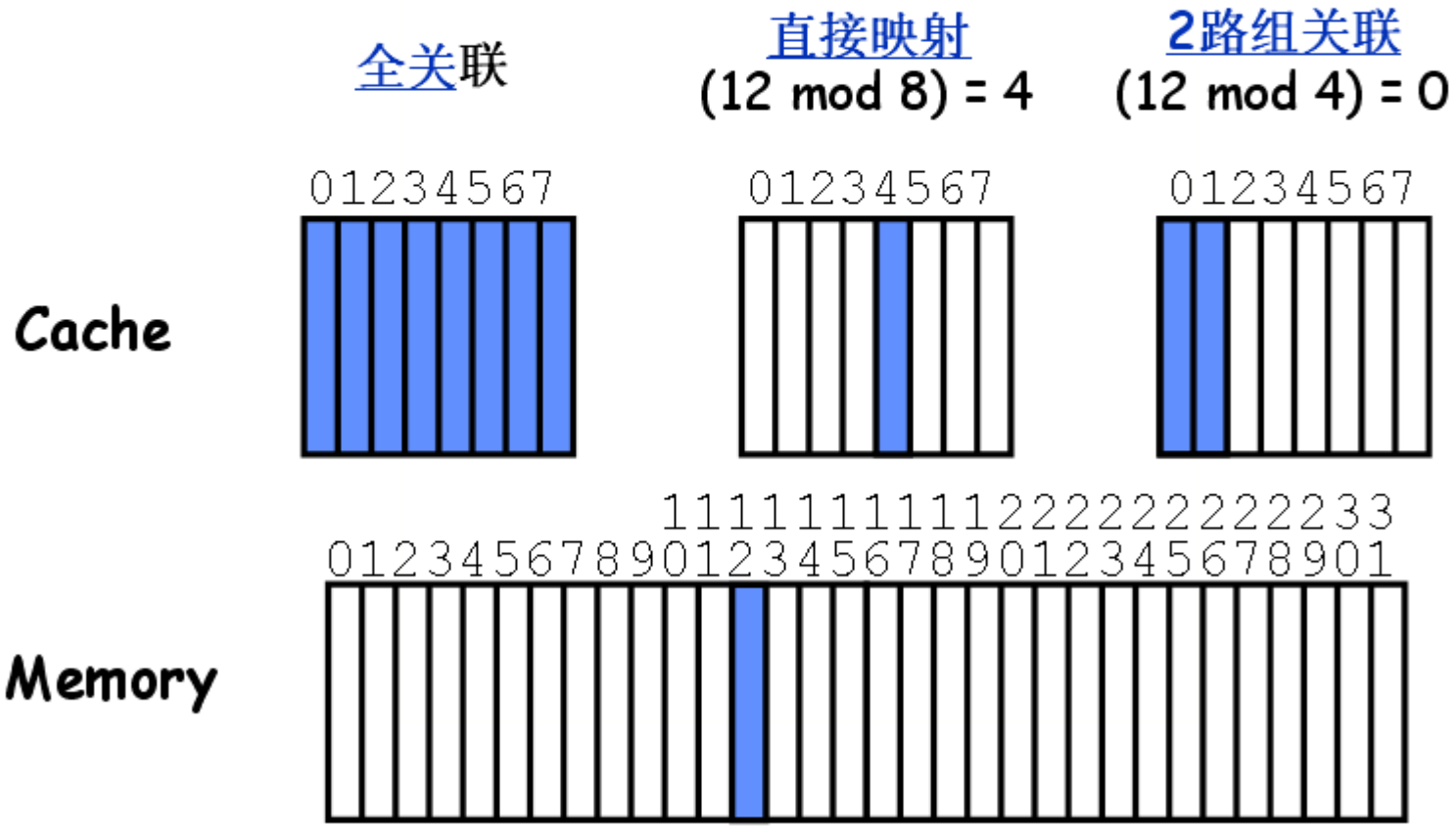
一旦缺失，应该替换哪个数据块?

- Q4: *写策略，Write strategy：*

写请求到来时怎么办?

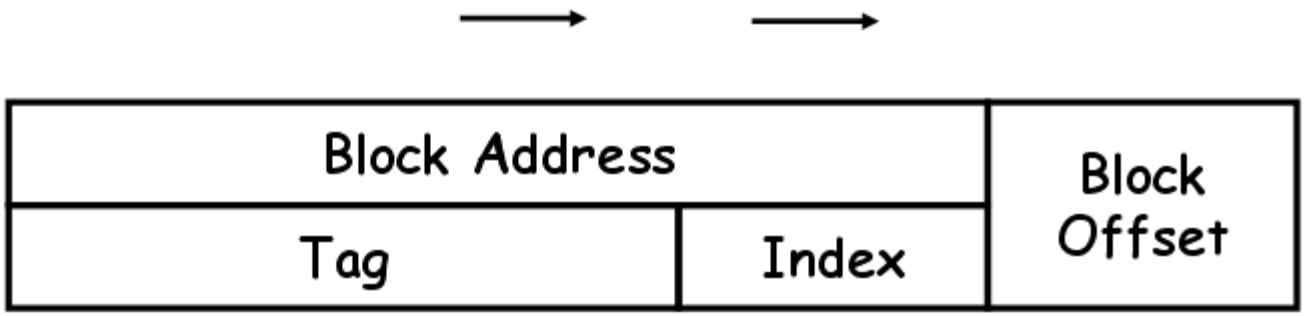
Q1: 数据块放置

- 数据块12放到大小为8块的cache中：
  - 全关联，直接映射，2路组关联
  - 组关联 = 块号 mod 组数



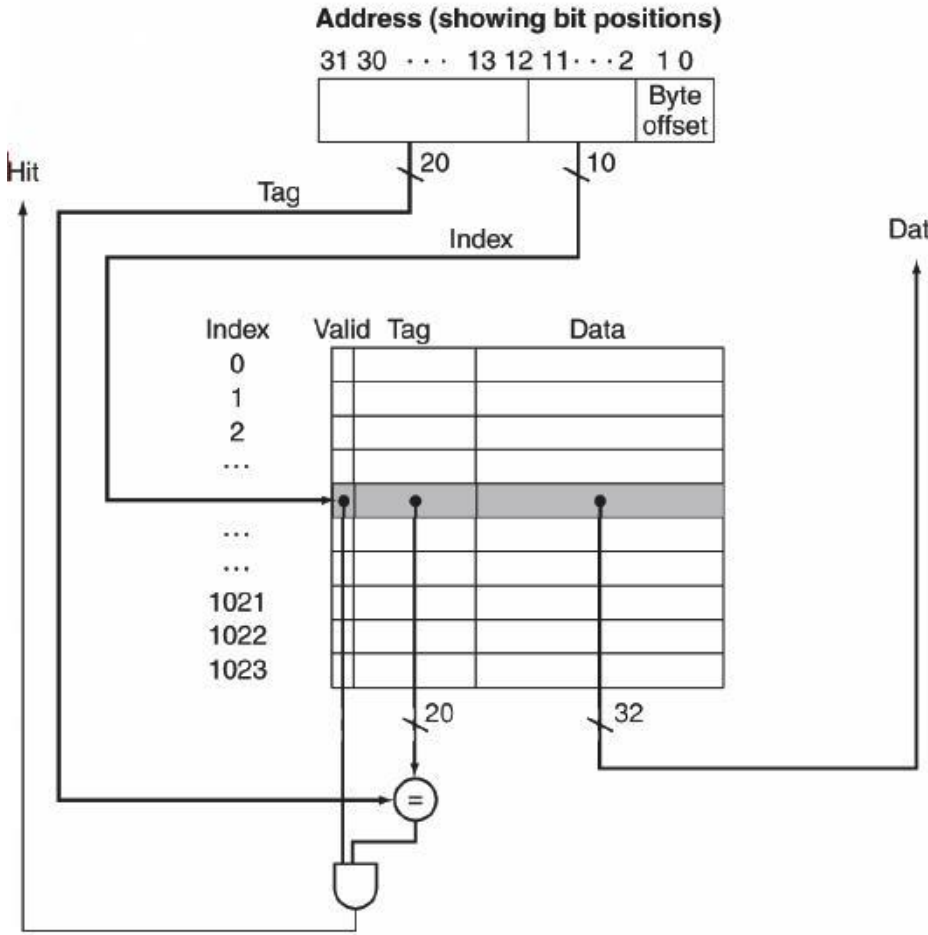
Q2: 数据块识别

- 在Cache中的每个数据块上添加标签Tag
  - 不需要检查 index 或者 block offset
- 增加关联度，缩小index位数，扩大了 tag位数



为什么index在低位，而tag在高位？

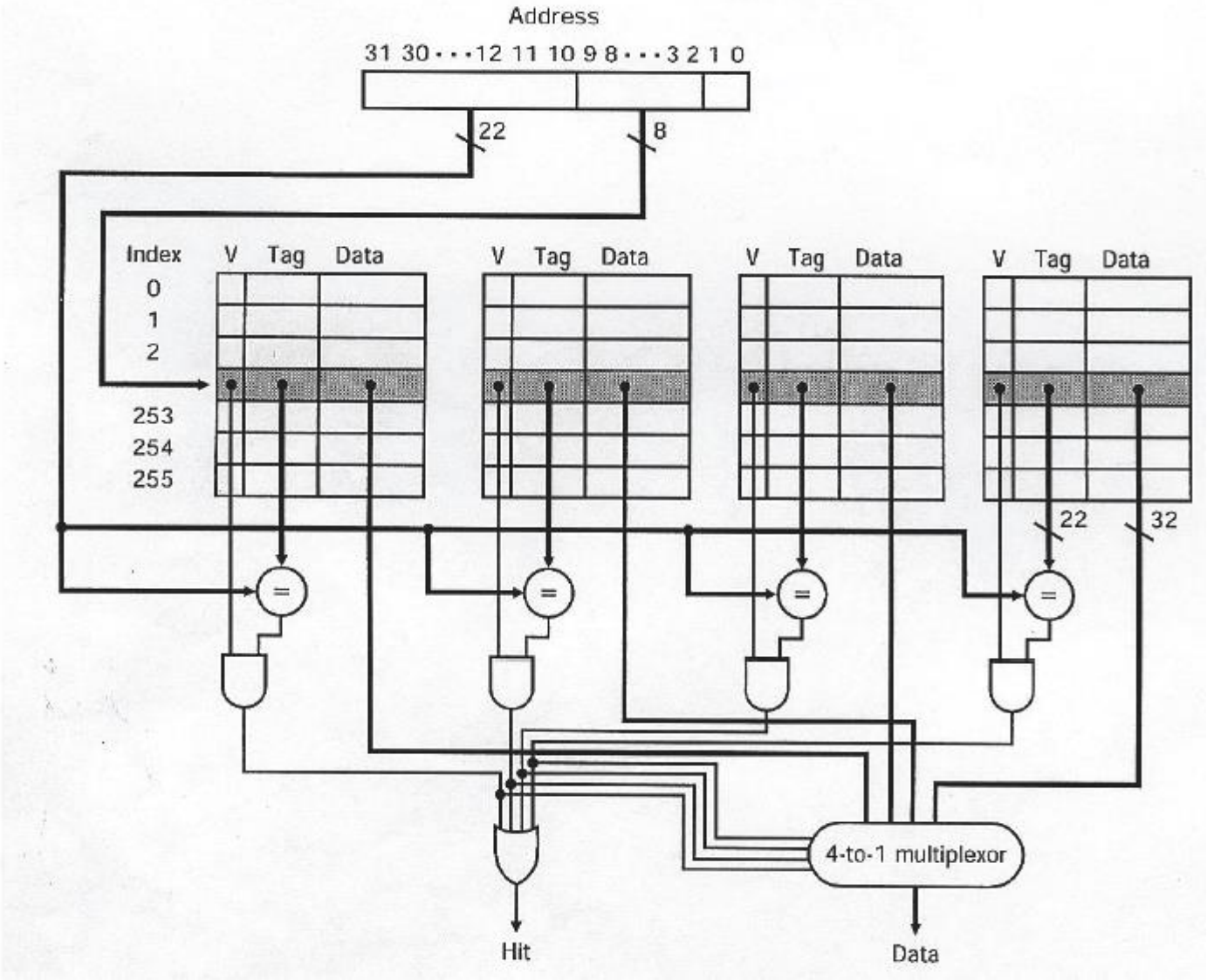
直接映射Cache



- 4KB direct mapped cache
  - 1024 blocks, 1 word/block
  - 32-bit address
- Access steps
  1. Use index to read V, tag from cache
  2. Compare tags from cache/address
  3. If match, return data & hit signal
  4. Otherwise, return miss

4路组相连Cache

“Set”



Q3: 数据块替换

- 对于直接映射是非常容易的，那么，对于组关联和全关联又如何呢？
- 最优化算法：
  - 将以后最长时间内不会使用的数据块替换掉
  - 约束：必须知道未来会发生什么事
- 常用算法：
  - 最近最少使用 (LRU)
    - » 替换最近最少访问的数据块
    - » 记录这些信息需要做些额外的事
  - 随机 (RAND)
    - » 替换组中随机的一块
    - » 容易实现
  - FIFO

Q4: 如何处理写操作?

	写通过	写回
策略	写到cache块中的数据，也写到底层存储器中	只写到cache中， 当从cache中替换一数据块时，才更新底层
调试	容易	难
读缺失是否产生写操作?	否	是
重复写到达底层?	是	否

Write Miss Policies

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

- Which data access patterns benefit from each policy?

Cache的性能评估

- Misses cause stalls
  - $CPI = CPI_{ideal} + CPI_{pipeline} + CPI_{caches}$
  - AMAT = hit time + miss rate * miss penalty
    - But reducing these 3 may be conflicting goals
- Classification of misses: the 3 Cs
  - Compulsory or cold misses
    - First access to an address within a program
    - Misses even with an infinite sized cache
  - Capacity misses
    - Cache not large enough to fit working set
    - Misses in fully associative cache of a given size
  - Conflict or interference misses
    - Misses due to lack of associativity
    - E.g. two addresses map to same block in direct mapped cache

Mark D. Hill（University of Wisconsin–Madison）

2019年获得Eckert-Mauchly奖

Eckert-Mauchly 奖是计算机体系结构领域最负盛名的奖项，以 1947 年诞生的世界上第一台电子计算机 ENIAC 的两位发明人John Presper Eckert 和 John William Mauchly 的名字命名。

贡献一：Cache Miss 的“3C 模型”

贡献二：缓存一致性模型

开发了一个使用顺序一致性(SC)的一致性模型，可以避免数据竞争(data race free)

贡献三：事务性内存

开发了LogTM事务性内存系统，这是最早的、且被广泛引用的事务性内存方法之一。

贡献四：并行计算机评估

开发了多种业内知名的评估工具，包括Dinero缓存模拟器、GEMS full system模拟器和gem5，最新开发的工具BadgerTrap，研究虚拟内存行为。

Cache Miss 的“3C 模型”

- 缺失的分类：3 Cs
  - 必然缺失、冷启动缺失：Compulsory (“Cold Start”) Misses
    - » 对不在cache中的数据行的第一次访问
  - 容量缺失：Capacity Misses
    - » 内存的活跃部分超出了cache大小
  - 冲突缺失：Conflict Misses
    - » 地址空间的活跃部分没有超出cache大小，但是许多行映射到相同的cache项
    - » 只涉及直接映射和组关联放置策略
- 可能有第四类：4th “C”:
  - 一致性缺失：Coherence Misses
    - » 多处理器cache一致性机制，使得一些数据块失效

Cache的功耗

- Power ≅ $C \bullet V^2 \bullet F + Vdd \bullet I_{leakage}$
  - Energy = Power • Execution Time

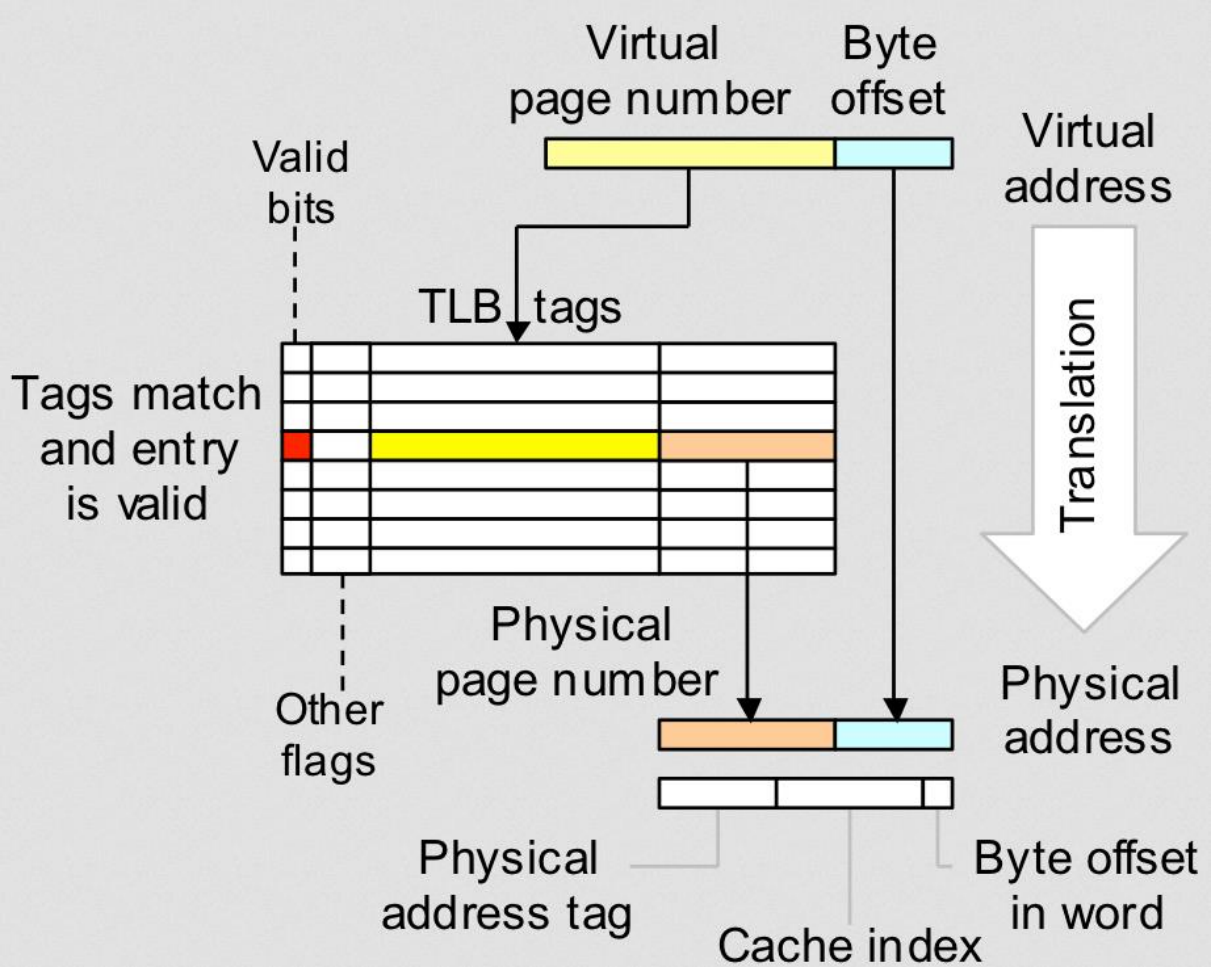
- Keep in mind
  - Dynamic power consumed for tags and data
  - Leakage is proportional to cache capacity
  - Power consumed on wires connected caches/processors
    - Long & wide wires can be expensive
  - Exampled @ 45nm CMOS technology
    - Read 64b from 8KB SRAM costs ~15pJ
    - Read 256b from 1MB SRAM costs ~500pJ
    - Communicating 256b over 10mm costs ~300pJ

TLB(Translation Lookaside Buffer)

- TLB是CPU内部的一个小存储管理单元，用于完成虚拟地址到物理地址快速映射和转换。

- 如果没有TLB，则每次取数据都需要两次访问内存，即查页表获得物理地址和取数据。

TLB基本结构


Virtual
page number
Byte
offset
Virtual
address
Valid
bits
TLB tags
Translation
Tags match
and entry
is valid
Physical
page number
Other
flags
Physical
address
Physical
address tag
Byte offset
in word
Cache index

高速缓存优化策略

- Multi-level caches and inclusion
- Victim caches
- Pseudo-associative caches
- Skew-associative caches
- Non-blocking caches
- Critical word first
- Prefetching
- Multi-ported caches

Cache优化技术

- 平均存储器访问时间的计算公式：

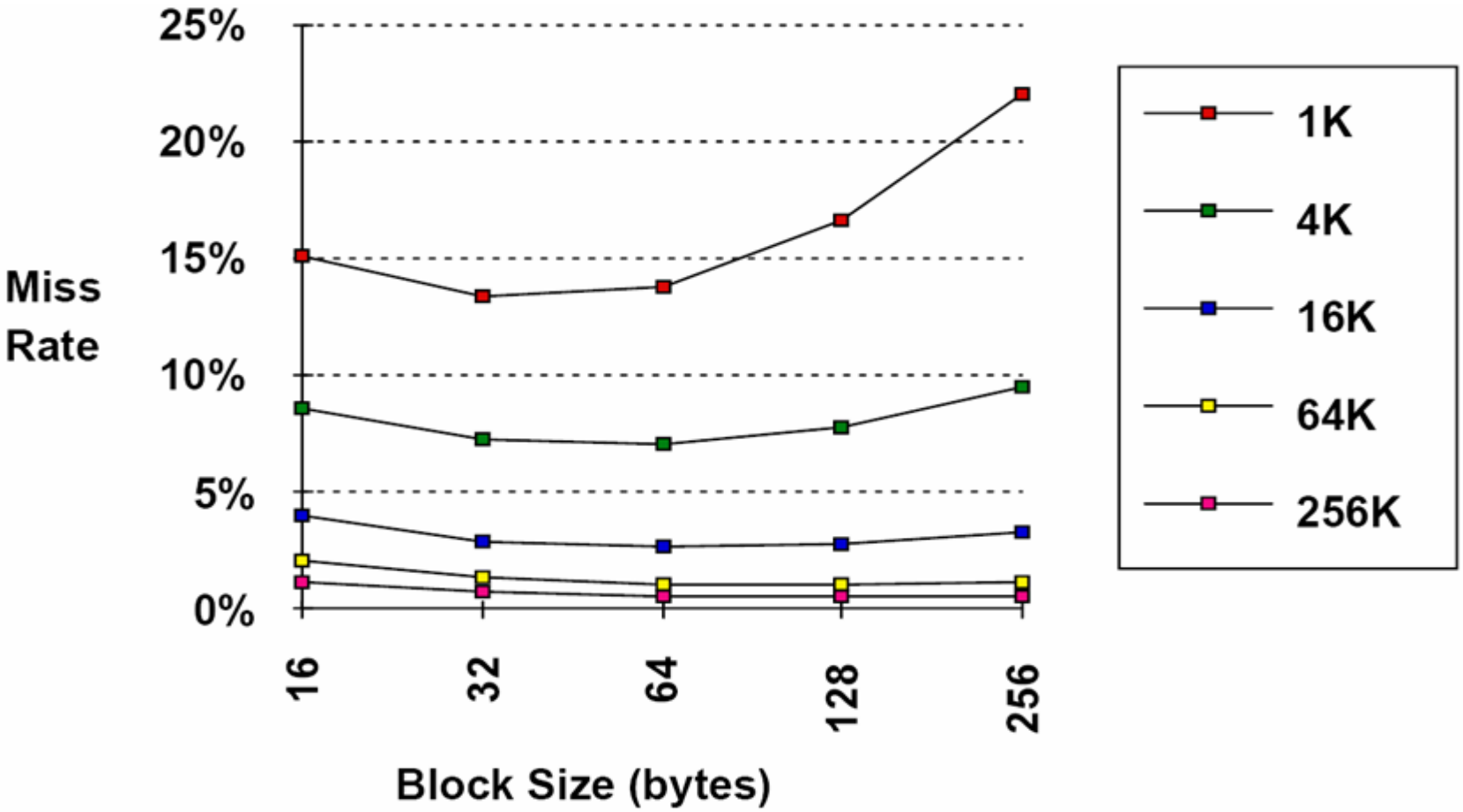
$$AMAT = 命中时间 + 缺失率 \times 缺失损失$$

- 为了提高cache性能，有三种方式：
  - 降低缺失率
  - 降低缺失损失
  - 降低cache命中时间
- 我们依次看这三种方式……

如何减少缺失?

- 3 Cs: Compulsory, Capacity, Conflict
- 在所有情况下，假设cache总大小不变
- 如果做如下改变，哪类缺失将明显受影响？
  - 改变块大小：
  - 改变关联度：
  - 改变编译器：
  - ……
- 我们依次来看一下…

减少容量缺失：增加块大小



- 减少容量缺失
- 更大的块，允许利用更多的空间局部性

降低冲突缺失：提高关联度

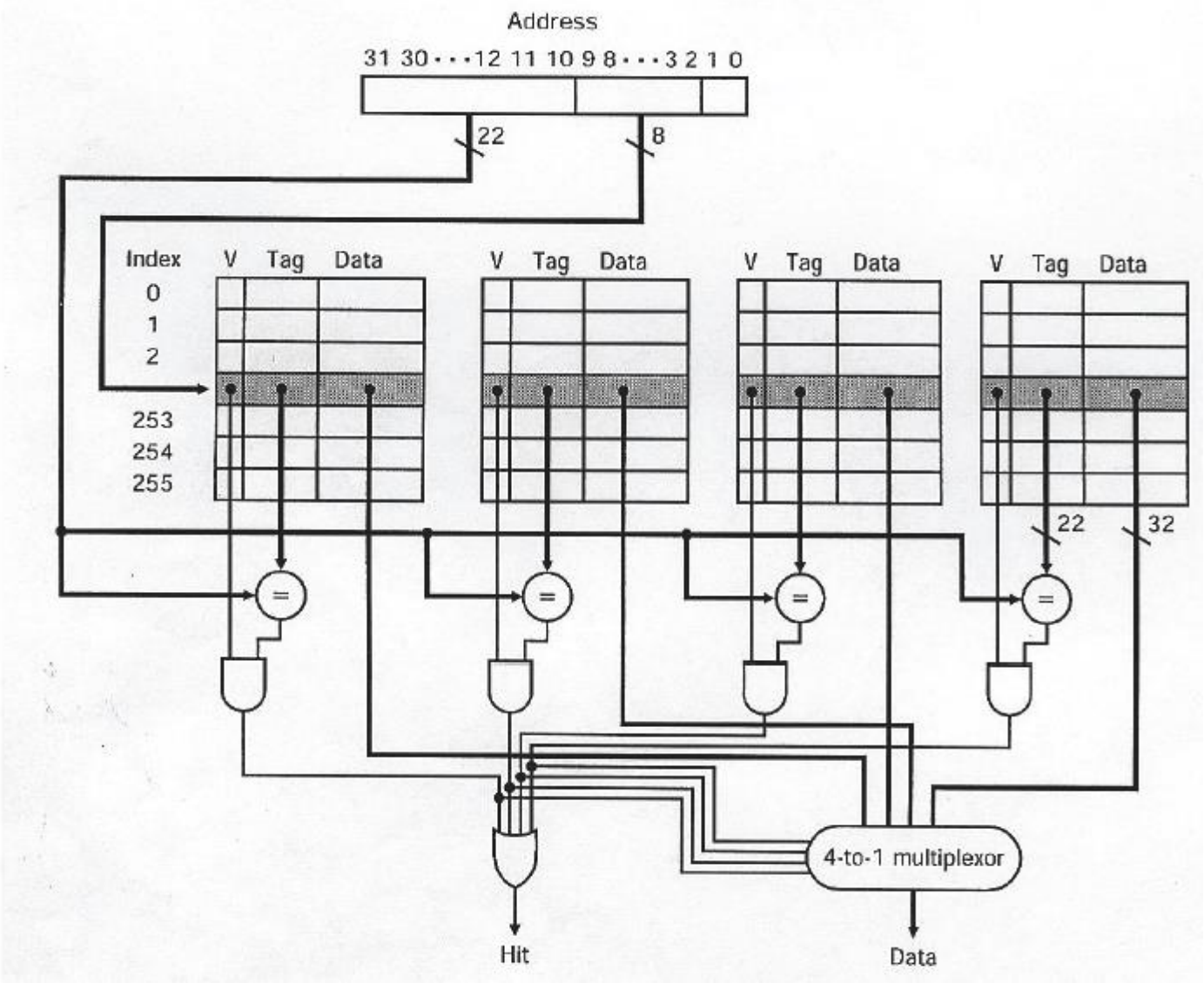
- 更高的关联度
  - 降低冲突缺失
- 注意：执行时间才是重要的
  - 时钟周期会不会增加？
- 例子：假设时钟周期 =
  - 1路时1.00，直接映射，
  - 2路时1.10，
  - 4路时1.12，
  - 8路时1.14，
- 提高关联度，缺失率降低了，但是cache命中时间轻微增加了

Cache大小 (KB)	关联度			
	1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	*1.48*	*1.47*	1.43
16	*1.29*	*1.32*	*1.32*	*1.32*
32	*1.2*	*1.24*	*1.25*	*1.27*
64	*1.14*	*1.2*	*1.21*	*1.23*
128	*1.1*	*1.17*	*1.18*	*1.2*

(红色代表提高关联度并没有提高平均访问时间)

Set-Associative Cache

"Set"



伪关联：缺失一次？再试！

- 如何兼具直接映射的快速命中时间和2路组关联的较低的冲突缺失？
- 将cache分成两半：一旦缺失，检查另一半cache看是否存在，如果存在则有一次伪命中pseudo-hit (慢命中)

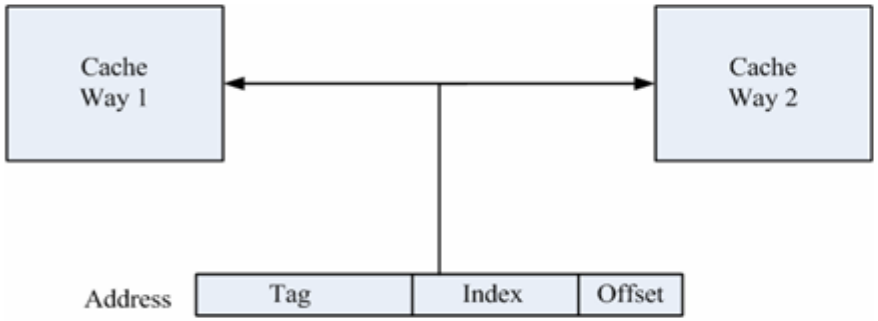
- 使用“路预测器”(way-predictor)来猜测先尝试哪一半
- 缺陷：因为命中需要1个或者2个时钟周期，CPU流水线难于设计
  - 用在MIPS R10000 L2 cache, 类似的用在UltraSPARC

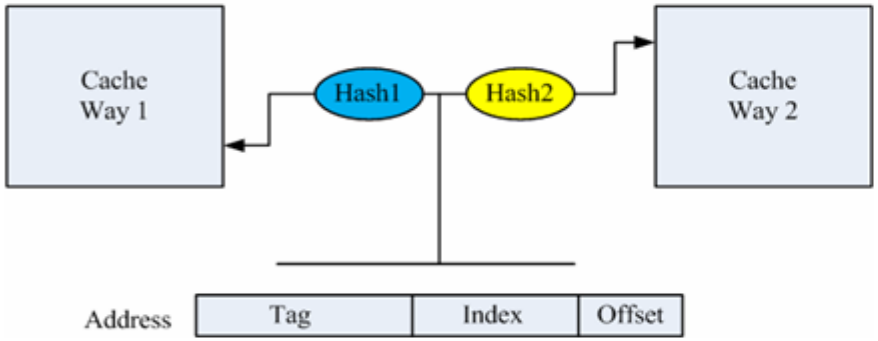
伪关联的扩展

- Idea: search the N ways sequentially
  - First search in way 1, if hit pass the data to processor
  - If miss, search way 2 in 2nd cycle, …
- Plus: hit time of direct mapped, miss rate of N-way SA
  - Each cycle only 1 way can provide data (fast multiplexing)
- Disadvantage: multiple hit times to handle
  - Depending on which way produces hit
  - Optimization: start from MRU way or predict (e.g., I-caches)
- Usage
  - With L1 caches to reduce miss rate without affecting hit time
  - With external caches (L3) to reduce board traces needed

斜关联：冲突缺失减少

- 思想：为每路cache使用不同的index，降低冲突缺失
  - N路cache：当 N+1 块的地址有相同的index时，发生冲突





- 通过hashing生成不同index
  - 例如，index数据位与一些tag位异或
  - 例如，对index位进行重排序
- 好处：index被随机化
  - 不太可能两块有相同index
  - 冲突缺失减少，cache利用得更好
  - 也许会降低关联度
- 成本：hash函数的延迟

多级Cache与多核处理器

- Multi-level caches
  - Private L1 instruction and data caches and (optionally) L2 caches
  - Shared last level cache (L3 cache in this diagram)
  - Connected through wide links

多级Cache

- Motivation
  - Optimize each cache for different constraints
  - Exploit cost/capacity trade-offs at different levels
- Private L1 caches (always)
  - Optimized for fast access time by one core
  - 8KB-64KB, direct-mapped to 4-way associative, 1-3 CPU cycles
- Private L2 caches (often)
  - Extend the capacity of L1, shield from latency of shared LLC cache
  - 256KB-512KB, 4 to 8-way associate
- L3 caches (aka, last level cache LLC)
  - Shared for best utilization (how?)
  - Optimized for low miss-rate: multi-MB, highly associative (why?)

Cache 性能公式

- L1 AMAT = HitTimeL1 + MissRateL1 * Miss PenaltyL1
  - MissPenaltyL1 is low, so optimize HitTimeL1
- MissPenaltyL1 = HitTimeL2 + MissRateL2 * MissPenaltyL2
  - MissPenaltyL2 is high, so optimize MissRateL2
- MissPenaltyL2 = DRAMaccessTime + (BlockSize/Bandwidth)
  - If DRAM time high or bandwidth high, use larger block size

- L2 miss rate
  - Global: L2 misses / total CPU references
  - Local: L2 misses / CPU references that miss in L1
  - The equation above assumes local miss rate
- Another useful metric: MPKI
  - Misses per kiloinstruction

多级Cache的包含关系

- Inclusion: data in L1 is always a subset of data in L2
- Advantages of maintaining multi-level inclusion
  - Easier coherence checks for multi-core and I/O
  - Check the lowest level only to determine if data in cache
    - Private L2 sees all coherence traffic for L1 as well
    - If L1 is write-through, L1 is involved only for invalidations
- Disadvantages
  - L2 replacements complicated if L2 and L1 block sizes differ
  - Wasted space if L2 not much larger than L1
    - The motivation for non-inclusion for some chips

多级Cache的真包含怎样维护

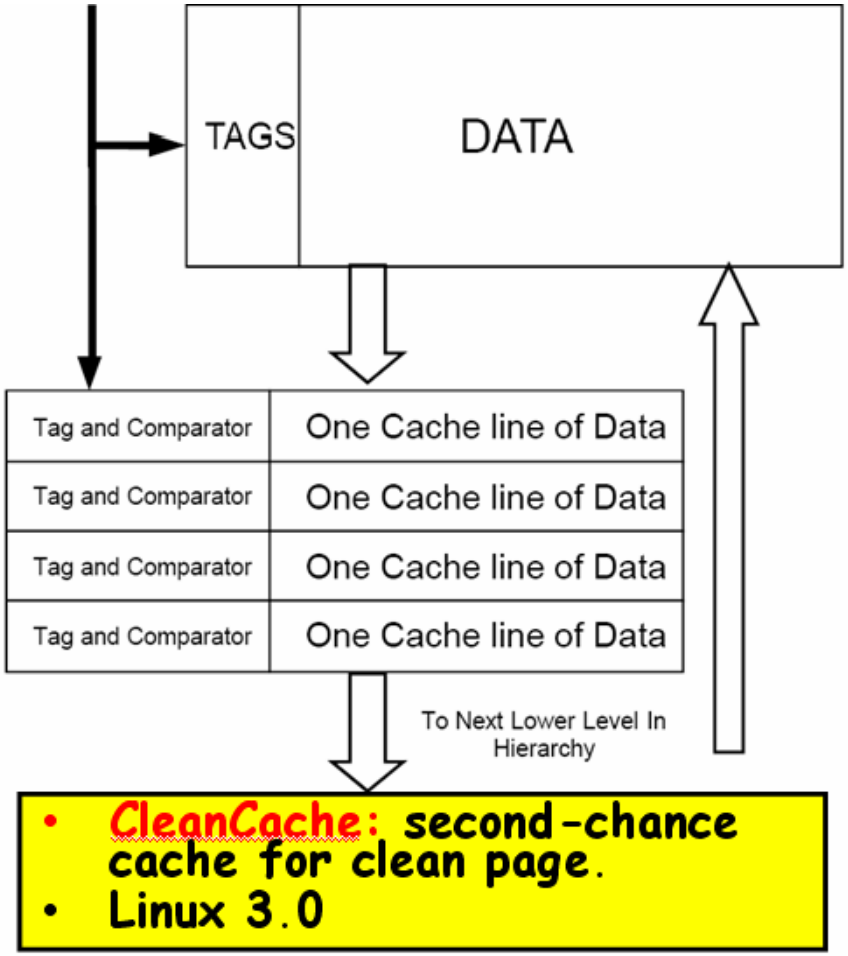
- On L1 misses
  - Bring block into L2 as well
- On L2 evictions or invalidations
  - First evict all block(s) from L1
  - Can simplify by maintaining extra state in L2 indicates which blocks are also in L1 and where (which cache way)
- L1 instruction cache inclusion?
  - For most systems, instruction inclusion is not needed (why?)
  - Bad for applications that stress the L2 capacity with small code
    - E.g. matrix multiply with huge matrices…

不用真包含模式的多级Cache设计

- Exclusive caches
  - Data in the L1 are NOT in the L2
  - Saves space
- Non inclusive caches
  - Data in the L1 may be in the L2

降低冲突缺失：Victim Cache

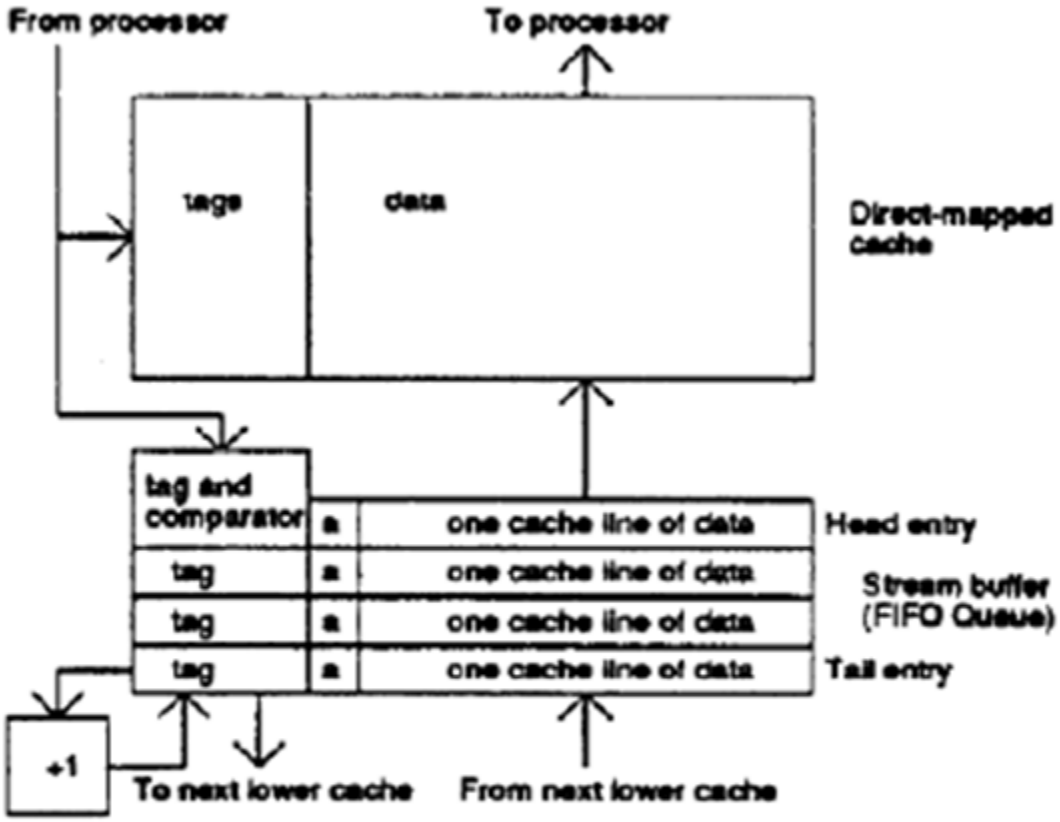
- 如何保留直接映射的快速命中间，而又避免冲突缺失？
- 增加缓冲区，放置从cache中丢弃的数据
- Jouppi [1990]: 对于一个4 KB的直接映射数据cache，4单元的 victim cache 消除了20% 到95% 的冲突缺失
- 在AMD Opteron, Barcelona, Phenom, IBM Power5, Power6中被采用



Jouppi, N. P. 1998. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In25 years of the international Symposia on Computer Architecture (Selected Papers) (Barcelona, Spain, June 27 - July 02, 1998). G. S. Sohi, Ed. ISCA '98. ACM, New York, NY, 388-397. DOI= http://doi.acm.org/10.1145/285930.285998

通过硬件预取来减少缺失

- 额外的块放到流缓冲区
- Cache缺失后，流缓冲区发起预取下一块
- 但不在cache中分配
  - 为避免cache污染
- 与cache一起，同时检查流缓冲区
- 依赖于有额外的内存带宽



From Jouppi [1990]:
http://doi.acm.org/10.1145/285930.285998

- SARC: Sequential Prefetching in Adaptive Replacement Cache.
- Usenix'2005

通过软件预取来减少缺失

- 数据预取
  - 加载数据到寄存器 (HP PA-RISC loads)
  - Cache预取：加载进 cache (MIPS IV, PowerPC, SPARC v. 9)
- 预取可分为两类：
  - 绑定预取：Binding prefetch，直接加载到寄存器
    - » 必须是正确的地址和寄存器！
  - 非绑定预取：Non-Binding prefetch， 加载到cache
    - » 可能是不正确的，不用去猜测!
- 发射预取指令花费时间
  - 预取发射的成本 < 降低缺失带来的收益?
  - 往往只有硬件预取才是有效的

通过编译器优化来减少缺失

- McFarling [1989]* 降低指令cache缺失达75%（8KB 直接映射 cache, 每块4 字节）
- 指令：
  - 根据调用图、分支结构和特征数据，选择指令的内存布局
  - 在内存中对过程进行重新排序，以减少冲突缺失
  - 实际上，这需要了解整个程序，是链接时的优化
- 数据：
  - 存储布局转换
  - 迭代空间转换

* “Program optimization for instruction caches”, ASPLOS89, http://doi.acm.org/10.1145/70082.68200

Cache优化技术

- 平均存储器访问时间的计算公式：

$$AMAT = 命中时间 + 缺失率 \times 缺失损失$$

- 为了提高cache性能，有三种方式：
  - 降低缺失率
  - 降低缺失损失
  - 降低cache命中时间
- 我们依次看这三种方式……

写策略：写通过 vs 写回

- 写通过：所有写操作更新cache以及底层的存储器
  - 可以随时丢弃cache数据 – 最新数据是在存储器中
  - Cache控制位：只有一个有效位valid
- 写回：所有写操作只更新cache
  - 不能简单的丢弃cache数据 - 可能必须写回数据到存储器中
  - Cache控制位：包括valid和dirty两位
- 二者的好处：
  - 写通过：
    - » 存储器（或者其他处理器）总是最新数据
    - » cache管理简单
  - 写回：
    - » 更低的带宽要求， 因为数据经常覆盖写多次
    - » 更好的容忍长延迟的存储器

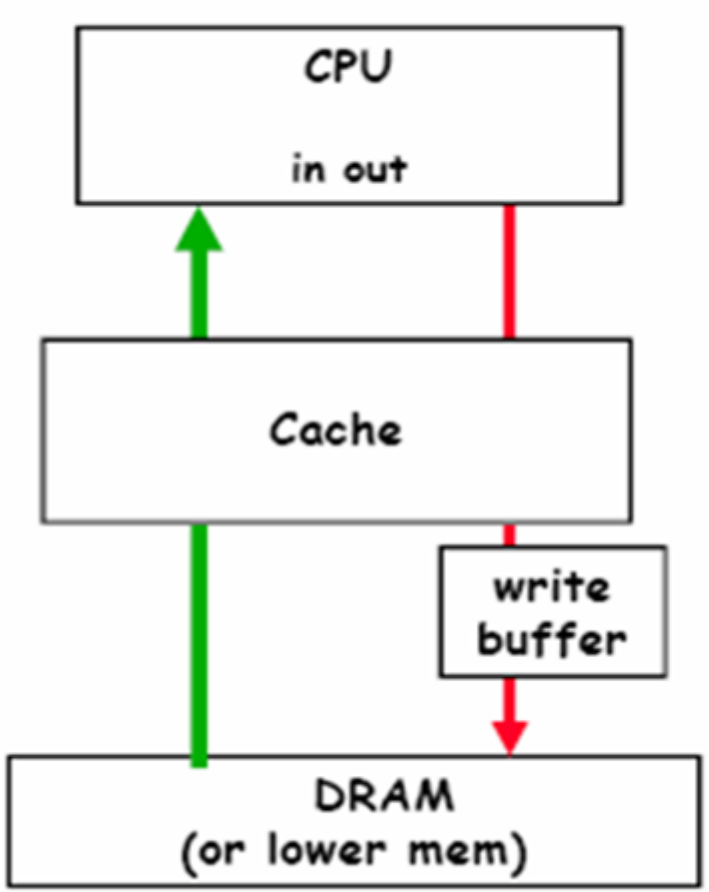
写策略2：写分配 vs 写非分配

写缺失时会发生什么？

- 写分配： 在cache中分配新的cache行
  - 通常，你必须首先执行“读缺失”来填充cache行中的其余部分！ [Why so ?]
  - 备选方案：每个字一个有效位

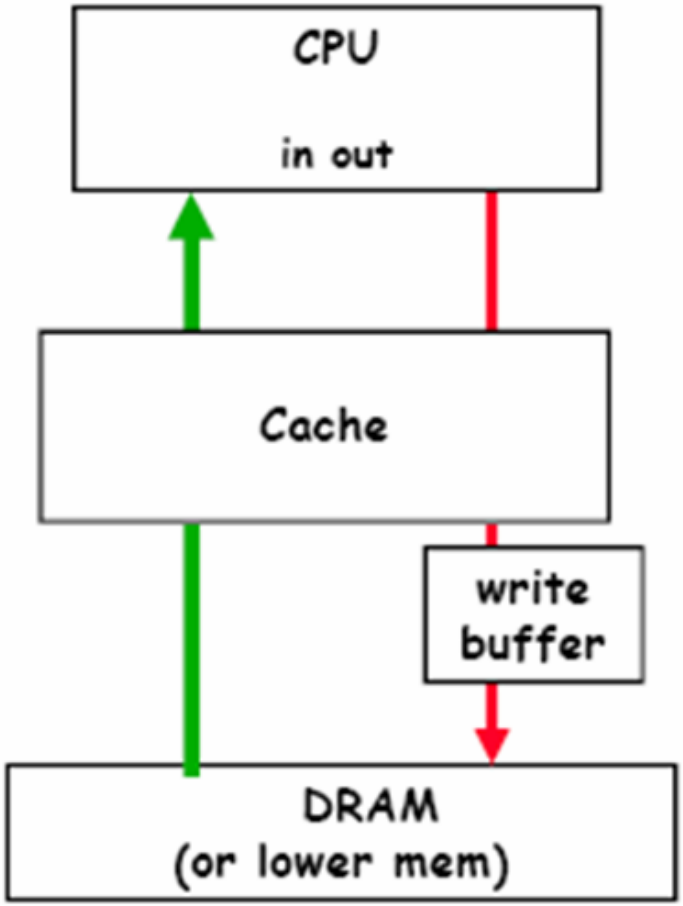
- 写非分配（或者“写绕过”）：
  - 直接简单的发送写数据给底层存储器，不分配新的cache行！

降低缺失损失1：设置写缓冲区



- 考虑带有写缓冲区的写通过
  - 缺失后的主存读，RAW冲突
    - » 可以简单等到写缓冲区为空，才允许读操作
    - » 风险是严重增加读缺失损失 (老的 MIPS 1000 高达50%)
    - » 解决方法：读之前检查写缓冲区内容；如果没有冲突，存储器访问继续
- 写回也需要缓冲区来保存替换出的块
  - 读缺失 ==→ 替换脏块
  - 正常：写脏块到存储器，然后执行读操作
  - 替代：复制脏块到写缓冲区，然后执行读操作，然后再写
  - CPU停顿少

写缓冲区问题



- 大小：2-8项典型的是足够的
  - 尽管一项可能存储整个cache行
  - 确保写缓冲区能够处理典型的写突发…
  - 分析你的常用程序，考虑低级存储的带宽
- 聚合写缓冲区
  - 合并邻接写到同一项
- 依赖性检查
  - 比较器检查load地址和缓冲区中pending stores
  - 如果匹配就存在依赖，load必须停滞
- 优化：load转发
  - 如果匹配且store有其数据，转发数据给load…
- 与victim cache集成?

降低缺失损失2：早启动和关键字优先

- 不要等到整块load完才重启CPU
  - 早启动：块中请求的字一到达，就将其发送给CPU，并让CPU继续执行
  - 关键字优先：也称作请求字优先。从存储器中首先请求缺失的字，一到达就发往 CPU； 在填充块中其他字的同时，让CPU继续执行。
- 一般来讲，对于大的数据块是有用的
- 对于顺序连续的多个字的访问很常见 - 不会从任何一种模式中受益 - are they worthwhile?

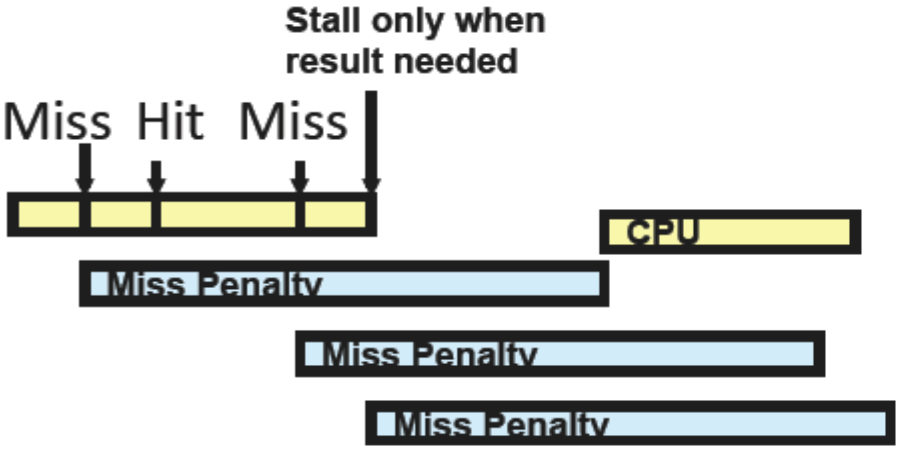
降低缺失损失3：
非阻塞Cache，降低缺失时CPU停顿

- 非阻塞cache或者非锁定 cache，允许数据 cache在一个缺失发生时，继续提供 cache命中
  - 允许乱序执行
- “hit under miss” 通过在缺失过程中工作，而不是忽略CPU请求，降低了有效的缺失损失
- “hit under multiple miss” or “miss under miss”可能通过交叠多个缺失，进一步降低有效的缺失损失
  - 显著增加了cache控制器的复杂性，因为可能有多个存储器访问同时发出去
  - Pentium Pro 允许 4 个同时发出的存储器缺失

Non-blocking Caches

Stall CPU on miss

Hit under miss



Multiple out-standing misses

Cache优化技术

- 平均存储器访问时间的计算公式：

$$AMAT = 命中时间 + 缺失率 \times 缺失损失$$

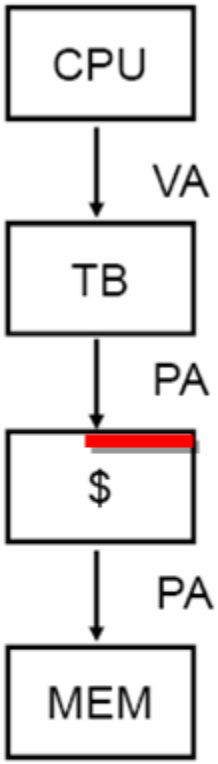
- 为了提高cache性能，有三种方式：
  - 降低缺失率
  - 降低缺失损失
  - 降低cache命中时间
- 我们依次看这三种方式……

降低cache命中时间

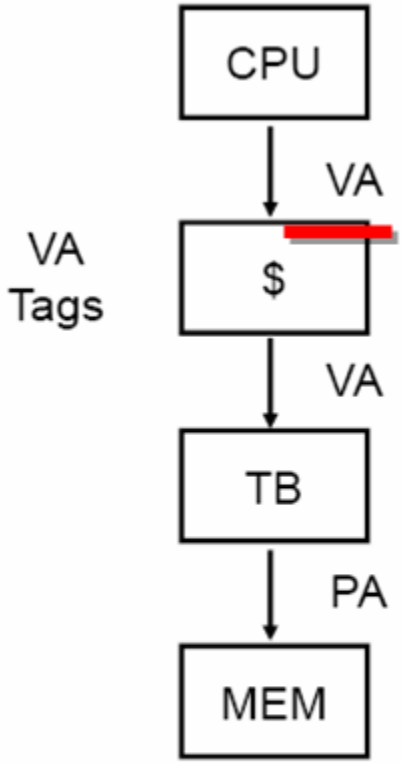
- 现代处理器如何有效地把多级cache都放在片上

-摩尔定律，片上晶体管数快速增长，与芯片面积相比，cache尺寸相对变小
-加速地址翻译
-Cache访问流水线化

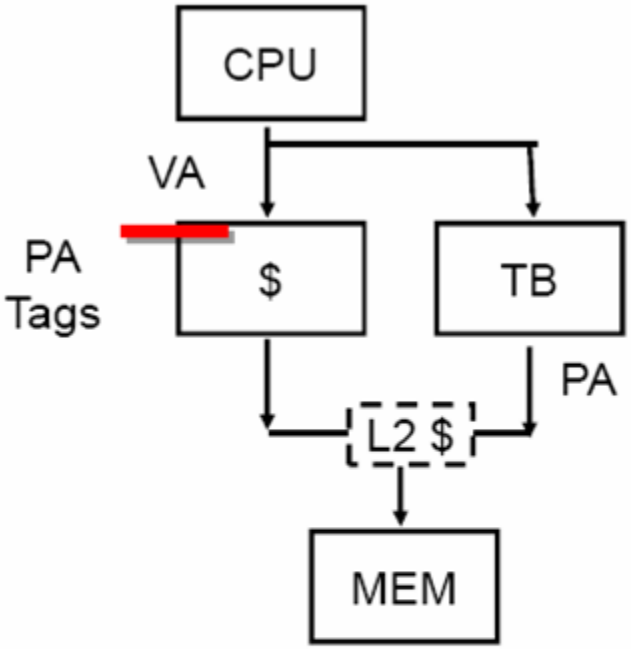
通过避免地址翻译达到快速命中



传统cache组织方法
cache访问前进行地址翻译



虚拟地址cache
缺失时进行地址翻译
存在同名问题



虚拟地址翻译与cache访问交叠
要求在地址翻译前后，cache index保持不变

通过避免地址翻译达到快速命中

- 发送虚拟地址给cache?
  - 称作虚拟寻址 Cache 或者 虚拟 Cache vs. 物理Cache
  - 每次进程切换 必须flush the cache; 否则得到虚假命中
    - » 成本是flush时间 + 由空cache引起的“compulsory”缺失
  - 处理别名问题(有时称作同义词);
    两个不同的虚拟地址映射到相同的物理地址;
- 别名问题的解决办法：页着色
  - 硬件确保cover index field & direct mapped必须是唯一的; called page coloring
- cache flush的解决办法
  - 添加进程标识符标签，标识进程以及进程内地址
  - 错误的进程，不能得到命中

Cache Optimization Summary

	Technique	MR	MP	HT	Complexity
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2