第八章存储器系统 (Memory Systems)

基于课程 PPT 整理

2025 年 12 月 30 日

核心考点清单

1. **存储层级结构与局部性原理**
   掌握时域局部性 (Temporal Locality) 和空域局部性 (Spatial Locality) 的概念及应用。

2. **存储性能分析计算**
   熟练掌握命中率、未命中率以及平均存储器访问时间 (AMAT) 的公式及计算方法。

3. **Cache 的三种映射方式**
   理解直接映射 (Direct Mapped)、组相联 (Set Associative) 和全相联 (Fully Associative) 的区别与结构。

4. **Cache 地址划分与容量计算**
   能够根据 Cache 参数计算标记 (Tag)、组索引 (Set Index) 和块偏移 (Block Offset) 的位数。

5. **Cache 未命中类型**
   掌握 3C 模型：强制未命中 (Compulsory)、容量未命中 (Capacity) 和冲突未命中 (Conflict)。

6. **Cache 的替换策略与写策略**
   理解 LRU/Random 替换算法，以及通写 (Write-through) 与回写 (Write-back) 策略。

7. **虚拟存储器与地址转换**
   掌握虚拟地址到物理地址的映射流程 (VPN $\rightarrow$ PPN)。

8. **页表与 TLB**
   理解页表结构、TLB (转换后备缓冲区) 的作用及其对性能的影响。

9. **ARM 存储架构特性**
   了解哈佛结构特点及 ARM 的两级地址转换机制。

1 存储系统性能分析

1.1 基本概念与术语

存储系统的性能主要取决于处理器能否快速在存储层级中找到所需数据。以下是核心评估指标：

命中 (Hit): 被请求的数据在当前存储层级（例如 Cache）中被找到。**未命中 (Miss):** 被请求的数据不在当前存储层级中，需要从下一级存储器获取。**命中率 (Hit Rate):**

$$\text{Hit Rate} = \frac{\text{Hits}}{\text{Memory Accesses}} = 1 - \text{Miss Rate} \tag{1}$$

未命中率 (Miss Rate):

$$\text{Miss Rate} = \frac{\text{Misses}}{\text{Memory Accesses}} = 1 - \text{Hit Rate} \tag{2}$$

1.2 平均存储器访问时间 (AMAT)

衡量存储系统性能的关键指标是平均存储器访问时间 (Average Memory Access Time)。

1.2.1 通用计算公式

对于多级存储系统，AMAT 的计算公式为:

$$AMAT = t_{cache} + MR_{cache} \times (t_{MM} + MR_{MM} \times t_{VM}) \tag{3}$$

其中:

t_{cache}: Cache 访问时间 MR_{cache}: Cache 未命中率 t_{MM}: 主存 (Main Memory) 访问时间 MR_{MM}: 主存未命中率 t_{VM}: 虚拟存储器 (Virtual Memory) / 硬盘访问时间

或者简化为两级层级（Cache + 主存）的公式:

$$[cite_start]t_{av} = h \times t_{cache} + (1 - h) \times t_{main} \quad \text{[cite: 394]} \tag{4}$$

其中 h 代表命中率。

1.3 性能分析计算示例

1.3.1 示例 1：计算命中率与未命中率

场景描述: 假设一个程序包含 2,000 条数据访问指令 (loads and stores)

- 1,250 条指令在 Cache 中找到数据。
- 剩余 750 条指令由主存或硬盘提供。

计算过程:

- **命中率 (Hit Rate):**

$$1250/2000 = 0.625$$

- **未命中率 (Miss Rate):**

$$750/2000 = 0.375 \quad (\text{或} 1 - 0.625)$$

1.3.2 示例 2：计算平均访问时间 (AMAT)

场景描述：基于示例 1 的程序，假设处理器有两级存储层级（Cache 与主存），且时间参数如下：

- Cache 访问时间 (t_{cache}) = 1 cycle
- 主存访问时间 (t_{MM}) = 100 cycles

计算过程：利用公式 $AMAT = t_{cache} + MR_{cache} \times t_{MM}$ 进行计算：

$$\begin{aligned} AMAT &= 1 + 0.375 \times 100 \\ &= 1 + 37.5 \\ &= 38.5 \text{ cycles} \end{aligned}$$

这表明虽然 Cache 命中率仅为 62.5%，但它显著降低了平均访问延迟（相较于直接访问主存的 100 cycles）。

2 直接映射高速缓存 (Direct Mapped Cache)

直接映射是最基本、最简单的 Cache 组织形式，理解它有助于掌握更复杂的组相联结构。其核心特征是主存中的每一个数据块只能映射到 Cache 中的一个特定位置。

2.1 映射规则与逻辑结构

定义与计算 在直接映射中，Cache 的每一组 (Set) 只包含一个块 (Block)，因此组数等于总块数。主存地址映射到 Cache 组的规则由取模运算决定：Cache 组号等于主存块地址除以 Cache 组数的余数。这意味着，如果两个内存块的地址在该模运算下结果相同，它们就会竞争同一个 Cache 位置，无法同时存在于 Cache 中。

地址划分 为了实现这种映射，CPU 发出的 32 位内存地址被硬件逻辑划分为三个部分。低位的**字节偏移 (Byte Offset)** 用于在数据块内部选择特定的字节；中间的**组索引 (Set Index)** 用于直接定位 Cache 中的具体行；高位的**标记 (Tag)** 则用于存储在 Cache 行中，作为身份验证的依据

2.2 硬件实现机制

查找流程 硬件首先利用地址中的组索引选取 Cache 中对应的行。每一行不仅存储了数据 (Data)，还存储了标记 (Tag) 和一个有效位 (Valid Bit)。系统读出该行的标记，并与地址中的标记字段进行比较。同时，系统检查有效位是否为 1。只有当“有效位为 1”且“标记匹配”这两个条件同时满足时（通过与门逻辑判断），才判定为**命中 (Hit)** 。

硬件开销 以 PPT 中的一个微型 Cache 为例（容量 8 字，块大小 1 字），其结构需要存储 1 位有效位、27 位标记位和 32 位数据位。这种结构相对简单，不需要复杂的并行比较电路或多路选择器来决定从哪一路输出数据，因此硬件成本低且访问速度快。

2.3 冲突未命中示例分析

冲突现象 直接映射最大的弱点在于**冲突未命中 (Conflict Miss)**。假设内存地址 '0x04' 和 '0x24' 都映射到同一个组（例如 Set 1）。当程序交替访问这两个地址时（例如在一个循环中同时操作数组 A 和数组 B），访问 '0x04' 会将 '0x24' 的数据踢出，反之亦然。

具体案例 在 PPT 的汇编示例中，指令反复读取地址偏移量为 '0x4' 和 '0x24' 的数据。由于这两个地址的索引位相同（均为 '001'），但标记位不同，导致每一次访问都会发现 Tag 不匹配，从而强制重新从主存加载数据并覆盖旧数据。这种现象会导致命中率为 0%，严重降低系统性能。

2.4 块大小的影响

利用空间局部性 为了缓解仅访问单个字带来的效率问题，可以增加块大小 (Block Size)。例如，将块大小从 1 个字增加到 4 个字。当 CPU 访问某个地址时，该地址相邻的 3 个字也会被一同载入 Cache。

性能权衡 利用空间局部性可以显著减少**强制未命中**。在 PPT 的示例中，通过增大块大小，未命中率从 20% 降低到了 6.67%，因为一次未命中加载了后续指令所需的多个数据。然而，增加块大小也可能导致 Cache 中包含的组数减少，进而增加冲突未命中的风险。

3 直接映射高速缓存详细示例解析

本节基于 PPT 第 39 页至 42 页提供的连续示例，演示一个 4 行（Block）的微型直接映射 Cache 如何处理一系列内存访问请求。

3.1 示例环境设定

- **地址空间：** 3 位地址（例如：000, 101）。
- **Cache 结构：** 共有 4 个块（Block 00 - Block 11）。
- **映射规则：**
  - **索引 (Index):** 地址的低 2 位。决定存放的行号。
  - **标记 (Tag):** 地址的第 3 位（最高位）。用于区分数据的身份。

3.2 访问序列数据

待处理的内存访问序列如下：

地址	数据
001	1111
010	0000
011	0110
100	1000
101	0001
111	0100

表 1: 内存访问请求序列

3.3 阶段一：冷启动与填充

操作步骤 1. **访问 001:** - 索引 ‘01‘，标记 ‘0‘。- Cache Block 01 为空 → **未命中 (Miss)**。- 写入：Tag=‘0‘, Data=‘1111‘。2. **访问 010:** - 索引 ‘10‘，标记 ‘0‘。- Cache Block 10 为空 → **未命中 (Miss)**。- 写入：Tag=‘0‘, Data=‘0000‘。

Cache 状态 此时 Cache 中 Block 01 和 10 被占用，Block 00 和 11 仍为空。

3.4 阶段二：继续填充

操作步骤 3. **访问 011:** - 索引 ‘11‘，标记 ‘0‘。- Cache Block 11 为空 → **未命中 (Miss)**。- 写入：Tag=‘0‘, Data=‘0110‘。4. **访问 100:** - 索引 ‘00‘，标记 ‘1‘。- Cache Block 00 为空 → **未命中 (Miss)**。- 写入：Tag=‘1‘, Data=‘1000‘。

Cache 状态 此时 Cache 的 4 个块已被全部填满。

- Block 00: Tag 1, Data 1000
- Block 01: Tag 0, Data 1111
- Block 10: Tag 0, Data 0000
- Block 11: Tag 0, Data 0110

3.5 阶段三：冲突与替换

这是直接映射最关键的演示部分，展示了当索引相同但标记不同时发生的情况。

操作步骤 5. **访问 101:** - **索引计算：** 地址 ‘101‘ 的低两位是 ‘01‘，映射到 **Block 01**。- **标记比较：** 地址 ‘101‘ 的 Tag 是 ‘1‘。- **冲突检查：** 查看 Block 01，当前存储的 Tag 是 ‘0‘ (来自地址 ‘001‘)。- **处理结果：** Tag 不匹配 ($1 \neq 0$) $\rightarrow$ **未命中 (Miss)**。- **替换动作：** 驱逐旧数据 ‘1111‘，写入新数据 ‘0001‘，并更新 Tag 为 ‘1‘。

6. **访问 111:** - **索引计算：** 地址 ‘111‘ 的低两位是 ‘11‘，映射到 **Block 11**。- **标记比较：** 地址 ‘111‘ 的 Tag 是 ‘1‘。- **冲突检查：** 查看 Block 11，当前存储的 Tag 是 ‘0‘ (来自地址 ‘011‘)。- **处理结果：** Tag 不匹配 $\rightarrow$ **未命中 (Miss)**。- **替换动作：** 驱逐旧数据 ‘0110‘，写入新数据 ‘0100‘，更新 Tag 为 ‘1‘。

Cache 最终状态 经过冲突替换后，Cache 的状态更新为：

- Block 00: Tag 1, Data 1000 (保持不变)
- **Block 01: Tag 1, Data 0001 (已替换)**
- Block 10: Tag 0, Data 0000 (保持不变)
- **Block 11: Tag 1, Data 0100 (已替换)**

3.6 示例总结

该系列示例清晰地证明了直接映射 Cache 的局限性：即使 Cache 总容量未满（或已被填满），只要新访问地址的**索引位 (Index)** 与旧数据相同，就会强制发生替换，而不管其他块（如 Block 10）是否空闲或未被使用。这正是**冲突未命中**的根源。

4 组相联高速缓存 (Set Associative Cache)

组相联映射是介于直接映射（硬件简单但冲突多）和全相联映射（无冲突但硬件昂贵）之间的折中方案。它是现代处理器中最常采用的 Cache 组织形式。

4.1 基本概念与结构

定义 在 N 路组相联 (N-Way Set Associative) Cache 中，Cache 被划分为 S 个组 (Set)。每个内存地址通过索引映射到一个唯一的组，但在该组内部，数据可以存放在 N 个块 (Way) 中的任意一个位置。

参数关系 如果 Cache 总容量为 C，块大小为 b，相联度为 N，则：

总块数 $B = C/b$ 组数 $S = B/N$

这种结构既保留了直接映射中通过索引快速定位组的优点，又利用了组内多个位置来降低冲突概率。

4.2 硬件实现机制

地址划分 与直接映射类似，物理地址被划分为 **标记 (Tag)**、**组索引 (Set Index)** 和 **块偏移 (Block Offset)**。

组索引用于选择具体的组。**标记**用于在组内进行比对。

并行查找 当 CPU 访问一个地址时，硬件首先根据组索引找到对应的组。该组内包含 N 个路 (Way)，每个路都有自己的 Tag 存储器和有效位。硬件会利用 N 个比较器，**并行地**将地址中的 Tag 与这 N 个路的 Tag 进行比较。

如果某个比较器匹配成功且有效位为 1，则产生命中信号 (Hit)，并通过多路选择器 (Mux) 输出对应的数据。如果所有比较器都不匹配，则产生未命中信号。

4.3 硬件实现分析 (Hardware Implementation Analysis)

基于 2 路组相联高速缓存的硬件原理图，我们可以将其工作流程拆解为以下关键步骤：

地址分解 (Address Decomposition): CPU 发出的 32 位物理地址被划分为三部分：**标记 (Tag, 28 bits):** 用于身份比对。**组索引 (Set, 2 bits):** 用于定位 Cache 中的具体行。图中 2 位索引意味着该 Cache 共有 $2^2 = 4$ 个组。**块偏移 (Byte Offset, 2 bits):** 用于字对齐的字节寻址。

并行访问 (Parallel Access): 这是组相联与直接映射最大的不同之处。组索引 (Set Index) 同时被送入 **Way 0** 和 **Way 1** 的存储阵列。这意味着硬件会在同一时刻读取两个路中对应组的 Tag 和数据。

并行比较与有效性检查 (Comparison & Logic): 硬件包含两组比较器 (Comparators)：

- 比较器 1 将地址中的 Tag 与 Way 1 读出的 Tag 进行比较。
- 比较器 0 将地址中的 Tag 与 Way 0 读出的 Tag 进行比较。

比较结果分别与各自路中的**有效位 (Valid Bit, V)** 进行逻辑 “与” 运算 (AND Gate)。只有当 “Tag 匹配” 且 “有效位为 1” 时，该路的命中信号 (Hit_1 或 Hit_0) 才会被置为有效。

命中信号生成 (Hit Generation): 最终的 Cache 命中信号 (Hit) 由一个逻辑 “或” 门 (OR Gate) 生成。只要任意一路发生命中 ($Hit = Hit_1 \vee Hit_0$)，CPU 即可判定数据获取成功。

数据选择 (Data Selection): 由于同时读出了两路的数据，系统需要一个 **多路选择器 (Multiplexer/Mux)** 来决定输出哪一路的数据。选择器的控制信号源自比

较逻辑的输出（例如 Hit_1）。如果 Way 1 命中，Mux 选择 Way 1 的数据；否则（假设命中），选择 Way 0 的数据。

总结：组相联架构通过增加比较器和多路选择器等硬件开销，实现了多路数据的并行检索，从而在保持访问速度接近直接映射的同时，显著降低了冲突未命中率。

4.4 性能分析示例

为了量化组相联的优势，我们对比其与直接映射在处理**冲突地址**时的表现。

测试场景 假设执行一段循环代码，反复交替访问两个内存地址：`0x4` 和 `0x24`。

在直接映射 Cache (1 路) 中，这两个地址映射到同一个组，互相驱逐，导致未命中率高达 100%（颠簸现象）。在 2 路组相联 Cache 中，虽然它们映射到同一个组，但该组有两个位置 (Way 0 和 Way 1)。

执行过程 (2 路组相联)

第一次访问 0x4：组内未找到，加载到 Way 0。(强制未命中) **第一次访问 0x24**：组内未找到，加载到 Way 1。(强制未命中) **后续访问**：再次访问 0x4 时，在 Way 0 命中；再次访问 0x24 时，在 Way 1 命中。

结果对比 对于上述序列，2 路组相联将未命中率从直接映射的 100% 降低到了 20% (仅剩初始的冷未命中)。这证明了提高相联度能显著减少**冲突未命中 (Conflict Misses)**。

4.5 相联度的权衡

虽然提高相联度可以降低未命中率，但也带来了代价：

收益递减：根据“未命中率趋势图”，从 1 路到 2 路、4 路的提升效果显著，但超过 8 路后，收益逐渐变小，因为冲突未命中已基本消除。**硬件开销：**需要更多的比较器和更宽的多路选择器，增加了电路面积和功耗。**访问延迟：**复杂的比较和选择逻辑可能会略微增加 Cache 的命中时间 (Hit Time)。

命中与未命中

命中是指请求的数据存在于当前存储层级（如Cache）中 ，而未命中为请求的数据不在当前存储层级中，需向下一层级请求 。

命中率定义为命中次数与总访问次数之比 。

强制性未命中：

指当处理器第一次访问某个存储单元时，由于该数据从未进入过 Cache，必然会导致未命中，也称为冷未命中 。

- 这种未命中是不可避免的，因为 Cache 初始状态通常为空 。
- 程序稳定运行后，在总未命中中所占比例相对较小 。

优化手段：通过增加块大小可以利用空域局部性，在一次访问中带入更多相邻数据，从而减少后续的强制性未命中 。

容量未命中 ：

由于 Cache 的总容量过小，无法容纳程序运行所需的全部活动数据（即“工作集”过大），导致之前调入的数据被挤出，随后再次访问时产生的未命中 。且即使采用全相联映射，只要 Cache 容量小于工作集，这类未命中依然会发生 。

优化手段：增加 Cache 容量是减少此类未命中直接且有效的方法 。

冲突未命中：

发生在组相联或直接映射 Cache 中。由于多个内存地址映射到了同一个 Cache 组（Set），导致即使 Cache 还有空余空间，特定组内的块也被频繁替换，从而产生未命中 。

- 在全相联映射中不会发生冲突未命中，因为全相联只有一个组。
- 两个数组（如 $a[i]$ 和 $b[i]$）映射到同一个 Cache 组，会导致交替访问时不断冲突 。

优化手段：

- 提高相联度：从直接映射转为 N 路组相联可以显著减少此类未命中 。
- 增加块大小：增大块大小可减少强制未命中，但会降低可容纳的块数，可能增加冲突/容量未命中。

平均存储器访问时间(串行访问)：

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{\text{MM}} + MR_{\text{MM}} \cdot t_{\text{VM}})$$

- $t_{\text{cache}}, t_{\text{MM}}, t_{\text{VM}}$：分别为高速缓存（Cache）、主存（Main Memory）和虚拟存储器（Virtual Memory，通常指硬盘）的访问时间。
- $MR_{\text{cache}}, MR_{\text{MM}}$：分别为 Cache 和主存的**缺失率**（Miss Rate）。

读命中

所希望的情况： 高速缓存直接传送数据到处理器。

读未命中

暂停CPU， 从主存中获取数据块， 传送到高速缓存， 重新启动

写命中

- 可以替换高速缓存和主存中的数据（ 通写）
- 只将数据写入高速缓存（ 稍后写入主存）

写未命中

- 将整个数据块读入高速缓存， 然后写某个字。
- 不分配 Cache 块，直接写主存（通常与 Write-Through 搭配）

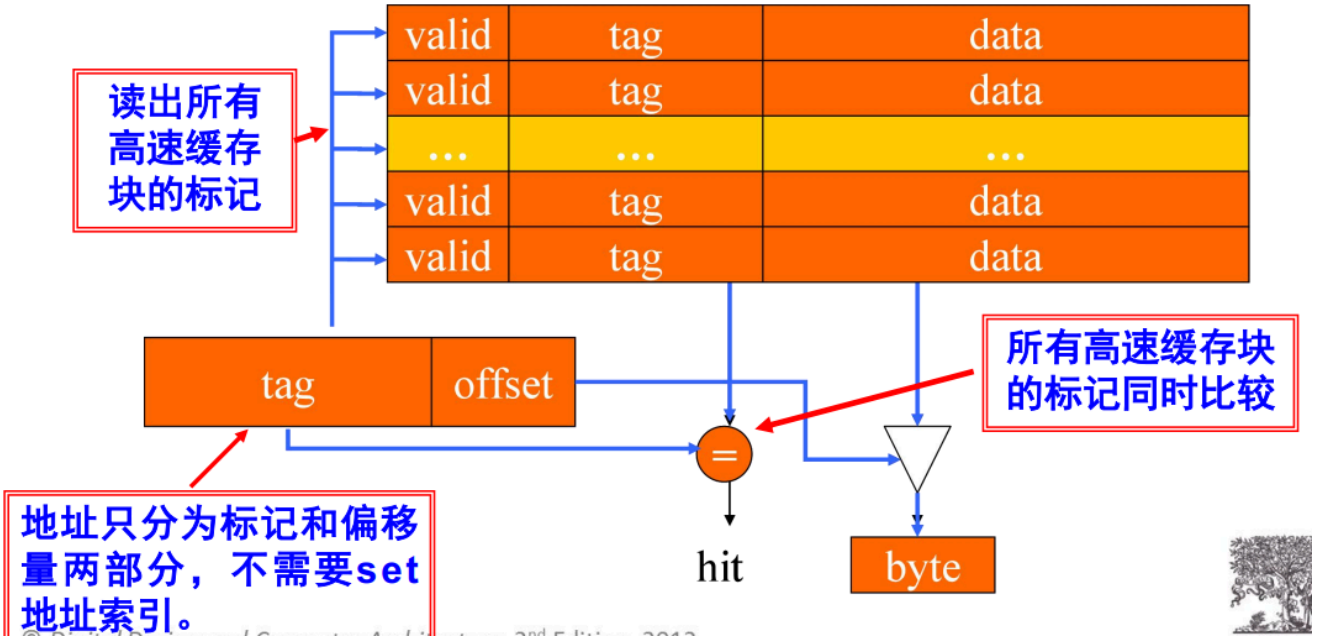
相联高速缓存性能

映射策略对比

特征	直接映射	N路组相联	全相联
每组块数 (N)	1	$1 < N < B$	B (总块数)
组数 (S)	B	B/N	1
优点	硬件简单、速度快	性能与成本的折中	命中率最高，无冲突
缺点	冲突未命中率高	硬件比直接映射复杂	硬件极其昂贵、功耗大

全相联高速缓存

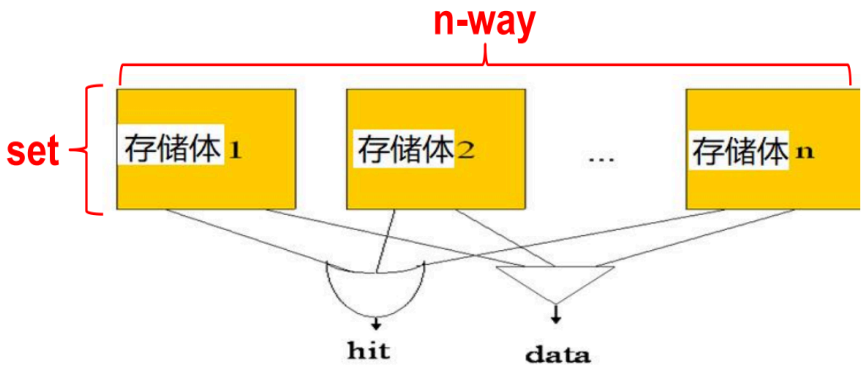
任意个内存数据块可以映射到高速缓存中的任意个地址 。



- 全相联不限制主存数据在高速缓存中的存放位置，使映射具有最大的灵活性， 命中率最高。
- 但是， 由于全相联通常需要采用内容可寻址存储器（ CAM、 根据给定的TAG寻址出cache里相符的block地址），耗费很高； 要在每次数据访问时读出并比较所有标记单元，速度慢， 能耗也非常大。
- 因此， 全相联高速缓存一般只用于设计小容量缓存器。

组相联高速缓存

- 组相联（ Set Associative） 是直接映射和全相联之间的折中。 组（Set） 由共享同一索引的所有块组成。
- 由每个组的高速缓存块个数来标识， N路组相联。
- 高速缓存访问请求按照索引广播给组内所有缓存块，判断是否命中。



N路组相联性能的提升核心在于通过增加组内块数（N），将直接映射中的**冲突未命中**转化为**命中**，从而大幅降低 MR，最终减小 $AMAT$ 。

性能趋势分析

- **增加相联度 (N)**：可以显著减少**冲突未命中**，但对强制和容量未命中无影响。
- **增加块大小 (b)**：利用**空域局部性**减少强制性未命中，但块过大会导致冲突未命中增加。

替换策略：
在组相联和全相联中，常用**最近最少使用 (LRU)** 策略来提高命中率，将组内或全相联高速缓存中最近最少使用的数据块。

- 优点:命中率高。
- 缺点:硬件耗费大。

策略	逻辑依据	优点	缺点
LRU	替换掉最近最少使用的块	**命中率高**，充分利用时域局部性	硬件复杂，随着相联度 N 的增加，开销剧增
Random	随机选择一个块替换	硬件简单，耗费低	命中率相对较低
FIFO	替换掉最早进入 Cache 的块	实现复杂度介于两者之间	命中率表现一般

Cache 存储量计算

地址位划分

对于32位系统，物理地址通常被划分为三部分：

- **偏移量位**：包括块内偏移和字节偏移，由块大小 (b) 决定。
  - 若块大小为 4 个字（16 字节），则偏移量为 $\log_2(16) = 4$ 位。
- **组索引位**：取决于 Cache 组织的组数 ($S = B/N$， 位数 = $\log_2(S)$)，决定该地址映射到哪个组。
- **标记位**：剩余的高位部分，用于确认该块是否为目标地址。
  - 标记位数 = $32 - 组索引位数 - 偏移量位数$

存储量公式

Cache 的总存储量不仅包含数据位，还包括有效位和标记位：

$$Total\_Bits = S \times N \times (Valid\_bit + Tag\_bits + Data\_bits)$$

- $S \times N$：是 Cache 中这类块的总数。
- **Valid_bit (有效位)**：
  - **大小**：通常为 **1 bit** 。
  - **作用**：指示该 Cache 块中的数据是否有效（1表示有效，0表示初始状态或数据已作废） 。
- **Tag_bits (标记位)**：
  - **定义**：存储内存地址的高位部分，用于唯一标识该块存储的是内存中哪个地址的数据。

    - 相联度 N 越高，组数 S 越少，索引位就越少，从而导致标记位越长。
- **Data_bits (数据位)**：
    - **大小**：取决于**块大小（Block Size, b）**。
    - **示例**：如果块大小为 4 个字，每个字 32 位，那么数据位就是 $4 \times 32 = 128$ 位 。

计算示例（P56）

假设一个Cache有4096个块， 块大小为4个字， 地址为32位，分别计算在直接映射、 2路组相联、 4路组相联、 全相联映射中， cache的总组数以及总的标记位数？

- 直接映射: 总的标记位数=32-12-4（offset） =16
- 2路组相联： 总的标记位数=32-11-4（offset） =17
- 4路组相联： 总的标记位数=32-10-4（offset） =18
- 全相联： 总的标记数=32-4（offset） =28

4路组相联时的cache总存储量：

$$
\begin{aligned}
S_{total} &= 4 \times 1024 \times (\underbrace{1 \text{ bit}}_{\text{Valid}} + \underbrace{18 \text{ bit}}_{\text{Tag}} + \underbrace{4 \times 32 \text{ bit}}_{\text{Data}}) \\
&= 4 \times 147 \text{ Kbit} \\
&= 588 \text{ Kbit}
\end{aligned}
$$

Cache 写操作

由于 Cache 与主存可能存在不一致，需采用特定写策略：

- **写命中**：
    - **通写 (Write-Through)**：每次写操作都将同时改变高速缓存和主存单元。 这种模式保证了高速缓存和主存的一致性， 但会产生额外的主存通信。
    - **回写 (Write-Back)**：只是在将某一单元从高速缓存中移出时才进行写存储器操作（利用dirty bit来判断该单元是否被写过， 系统不能突然断电） 。 这种模式可以减少那些该单元移出高速缓存之前对它进行的多次写操作。
- **写未命中**：
    - **写分配**：将整个块从主存读入 Cache，然后再写入 。
    - **非写分配**：不调块，直接写主存 。

虚拟内存及示例

虚拟存储器

虚拟存储提供容量更大的存储空间，使得逻辑上的存储器容量不受限于物理内存的大小 ，可以将主存（DRAM）视为硬盘（HDD/SSD）的高速缓存 。

物理内存（DRAM）作为“物理存储”，而硬盘空间则承载完整的“虚拟存储” 。

虚拟地址

程序运行时直接引用虚拟地址，其完整的虚拟地址空间持久化存储于硬盘，而主存（DRAM）仅作为该空间的一个活跃子集缓存。CPU 负责将虚拟地址实时转换为物理地址（DRAM 地址），一旦所需数据未在 DRAM 中命中，系统则启动调页机制从硬盘获取。

存储器保护

操作系统为每个程序维护独立的虚拟到物理地址映射表，使得不同程序即便使用相同的虚拟地址也能访问各自独立的物理空间。这种机制确保了程序运行的透明性，无需感知其他程序的地址分配，同时从硬件层面隔离了进程，防止恶意程序或病毒破坏其他程序的存储空间。

高速缓存 (Cache)	虚拟存储器 (Virtual Memory)
块 (Block)	页 (Page)
块大小 (Block Size)	页大小 (Page Size)
块偏移 (Block Offset)	页偏移 (Page Offset)
未命中 (Miss)	缺页 (Page Fault)
标记 (Tag)	虚拟页号 (VPN)

- 页大小Page size: 从硬盘到DRAM一次传输的数据大小
- 地址转换Address translation: 将虚拟地址转换成物理地址
- 页表Page table: 通过查找页表来进行虚拟地址到物理地址的转换

物理内存作为虚拟内存的高速缓存来使用，系统利用有限的物理内存，动态地存放虚拟地址空间中最活跃、最常用的数据。

地址转换与页表

地址转换

地址转换是指将程序产生的虚拟地址（Virtual Address, VA）转换为内存硬件识别的物理地址（Physical Address, PA）的过程 。

地址结构的组成
物理地址和虚拟地址都被划分为两个部分：

- 页号：
  - 虚拟页号 (VPN)：用于在存储系统中定位数据所在的“逻辑页”，对应于 Cache 中的 Tag 。
  - 物理页号 (PPN)：数据在 DRAM 中的实际物理块编号 。
- 页偏移：
  - 确定数据在页内的具体字节位置 。

- 在转换过程中，**页偏移保持不变**，直接从虚拟地址拷贝到物理地址 。

转换逻辑
在地址转换过程中，CPU 首先从虚拟地址中提取出虚拟页号，并通过页表这一映射机制将其转换为对应的物理页框号（PPN）；随后，系统将获取到的 PPN 与原虚拟地址中的页偏移量进行位拼接，从而组合成最终用于访问 DRAM 的物理地址（PA）。

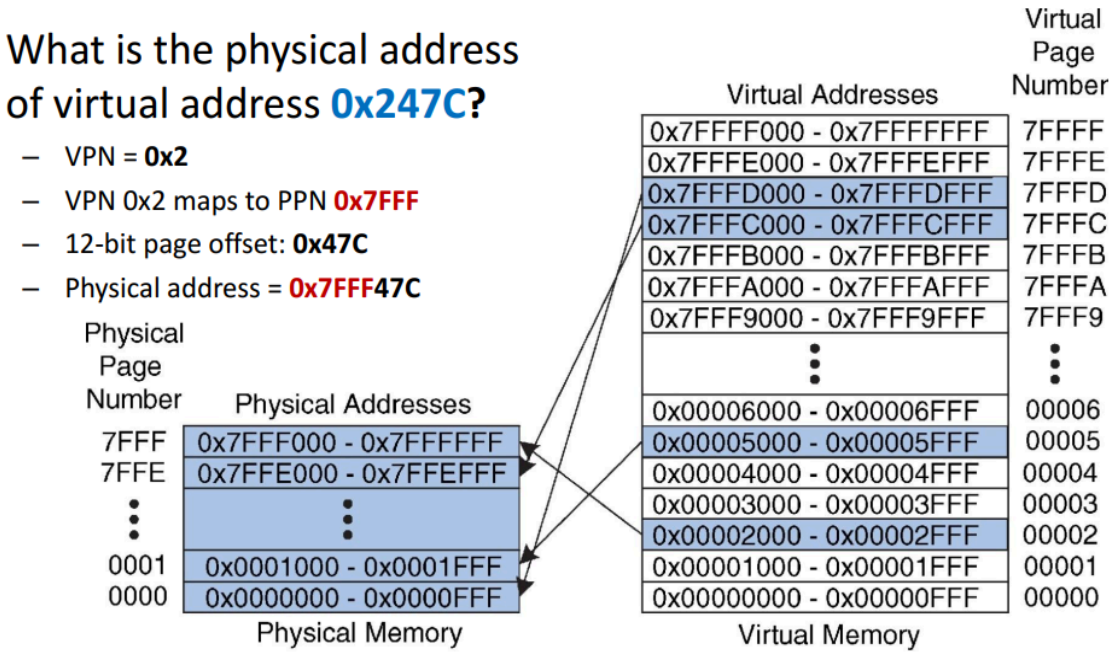
页表

页表作为实现地址转换的核心数据结构，其条目（PTE）数量与虚拟地址空间中的虚拟页总数一一对应，确保每个虚拟页都有唯一的记录空间。在转换时，系统直接将虚拟页号作为页表的偏移索引，这种随机访问机制使得 CPU 能够以 $O(1)$ 的时间复杂度快速定位目标条目并获取物理页框号。

页表条目 (PTE) 的组成:
每个条目通常包含以下核心位：

- 有效位 (Valid bit)：
  - 若为 1：表示该页当前已加载到物理内存中 。
  - 若为 0：表示该页不在物理内存中（在硬盘上），访问将触发“缺页异常（Page Fault）”。
- 物理页号 (PPN)：记录该虚拟页对应的实际内存地址编号 。
- 控制位：包括脏位（标记是否被改写，用于回写策略）和使用位（用于近似 LRU 或 Clock 等替换算法） 。

计算示例（P81）



问题:

将程序给出的虚拟地址 `0x247C` 转换为主存（DRAM）中实际的物理地址 。

- **页偏移位数 (n)**：$n = \log_2(\text{Page Size}) = \log_2(4\ \text{KB}) = 12\ \text{bits}$
- **地址结构**：$\text{Address} = \text{Page Number} \times 2^n + \text{Page Offset}$

第一步： 提取页偏移 ($Offset$)

$$Offset = VA \pmod{2^n} = 0x247C \pmod{0x1000} = \mathbf{0x47C}$$

注：在十六进制中，这等同于取虚拟地址的最后 $n/4 = 3$ 位

第二步： 提取虚拟页号 (VPN)

$$VPN = \lfloor VA/2^n \rfloor = \lfloor 0x247C/0x1000 \rfloor = \mathbf{0x2}$$

注：这等同于将虚拟地址右移 12 位

第三步： 查表获取物理页号 (PPN)

根据页表映射关系：

$$PPN = \text{PageTable}[VPN] = \text{PageTable}[0x2] = \mathbf{0x7FFF}$$

第四步： 生成物理地址 (PA)

$$PA = PPN \times 2^n + Offset$$

$$PA = 0x7FFF \times 0x1000 + 0x47C = \mathbf{0x7FFF47C}$$

得到最终答案

4. TLB (转换后备缓冲区)

- **定义**：它是页表的专用 Cache，存储最近使用的页表项 。
- **性能**：TLB 通常是全相联映射，访问时间小于 1 个时钟周期，命中率通常 > 99% 。

ARM 物理 Cache 工作流程

ARM 通常采用**物理 Cache**（即在访问 Cache 前先完成地址转换）

具体工作流程 (P99)：

1. **发出虚拟地址 (VA)**：处理机发出访问请求。
2. **查询 TLB**：
   - **命中**：产生实地址（物理地址 PA）。
   - **未命中**：查询主存中的页表 。若页表也未命中，则从后备存储器（硬盘）加载页 。
3. **查询 Cache**：
   - 使用产生的物理地址查询 Cache 。
   - **命中**：直接将数据传送回处理机 。
   - **未命中**：从主存读取数据块，更新 Cache，再传回处理机 。

> **注意：** ARM 支持二级地址转换模式，将地址映射细分为段（1MB）、大页（64KB）或小页（4KB）映射 。