[存储原理 2](#_Toc505863323)

[程序进入计算机： 3](#_Toc505863324)

[并行主存系统 8](#_Toc505863325)

[虚拟存储 16](#_Toc505863326)

[Cache 23](#_Toc505863327)

[工作过程 26](#_Toc505863328)

[LRU算法的硬件实现 30](#_Toc505863329)

[举例：DEC的Alpha AXP21064中的内部数据Cache 34](#_Toc505863330)

[Cache优化 39](#_Toc505863331)

[Disk 42](#_Toc505863332)

[RAID 43](#_Toc505863333)

# 存储原理

存储器要求：大容量、价格低、速度快，但往往速度快的存储器容量小价格昂贵，因此要采用：多种存储器技术，构成多级存储层次结构

借助于辅助硬件，Cache与主存构成一个整体，这个层次的工作由硬件实现，对于应用程序员和系统程序员都是透明的

|  |  |
| --- | --- |
| 主存-Cache | 主存-外存 |
| 弥补主存速度不足 | 弥补主存容量不足 |
| 全部由专用硬件实现 | 主要由软件实现 |
| 速度比值：几比一 | 速度比值：几万比一 |
| 典型块大小：几十~几百B | 典型块大小：几千或更大B |
| 不命中时CPU不切换 | 不命中时CPU切换到其他进程 |

|  |  |
| --- | --- |
| 局部性  原理 | 程序访问的指令和数据在存储器中不是均匀分布的，而是：  时间局部性：程序中存在大量循环操作，某指令(数据)被执行，不久后可能再次被执行(访问)  空间局部性：某存储单元被访问，不久后可能访问其附近的存储单元，即某时间段内所访问的指令或数据趋于聚集在一个连续区域中 |
|  | 内存分为系统区和用户区，系统区提供给OS，位于内存的低址；用户区存放用户程序(一个用户区存放一道用户程序)  存储管理的目标：方便用户和 提高内存利用率 |
| CPU  访问  内存 | 以字或字节为单位 |

映像规则：[当把一个块调入高一层(靠近CPU)存储器时，可以放在哪些位置上?](5-1-16.exe)

查找算法：[当所要访问的块在高一层存储器中时，如何找到该块?](5-1-17.exe)

替换算法：主存中的块比Cache中的多，所以会出现当从主存中调一个块到Cache中时，该内存块所映射的Cache块全部被占用，这时就需要从中替换，当发生不命中时，应替换哪一块？

写策略：[当进行写访问时，应进行哪些操作?](5-1-18.exe)

|  |  |
| --- | --- |
| 多级存储层次 | 由一组采用不同技术实现的存储器构成的存储系统  存储层次中，个存储器之间一般满足包容关系，即任何一层存储器中的内容都是下一层(离CPU更远)存储器中内容的子集。  目标：从CPU看，该存储系统的速度接近于离CPU最近的存储器的速度，而容量接近于离CPU最远的存储器的容量 |
| 命中时间 | 访问Cache命中所用的时间 |
| 失效率 | CPU访存时，在一级存储器中找不到所需信息的概率 |
| 失效开销 | CPU向二级存储器发出访问请求吧这个数据调入一级存储器所需要的时间 |
|  |  |
|  |  |
|  |  |

# 程序进入计算机：

1.编译：编译程序对用户源程序进行编译，形成成若干个目标模块

2.链接：各目标模块，以及所需库函数链接在一起，形成一个完整的装入模块

链接程序将各个模块链接成一个完整的可执行目标程序时，

依次按各个模块的相对地址构成统一的从0号单元开始编址的逻辑地址空间

编译过后的程序需要经过链接才能装载，链接后形成的目标程序中的地址也就是逻辑地址

以C语言为例：C语言的程序经过预处理（CPP），编译（ccl），汇编（as），链接（ld），产生了可执行文件，

其中链接的前一步，产生了可重定位的二进制目标文件。

C语言采用源文件独立编译的方法，如程序main.c，file1.c，file2.c，file1.h，file2.h

在链接的前一步产生了main.o、file1.o、file2.o ，这些称为目标文件，即目标模块，它们采用的逻辑地址都是从0开始，但只是相对于该模块的逻辑地址。连接器将这三个文件，libc、其他库文件链接成一个可执行文件。

链接阶段主要是完成了重定位，形成整个程序的完整逻辑空间

静态连接：链接完成后，不能再拆开

装入时动态链接：边装入，边链接

运行时动态链接：在程序执行中，需要某个目标模块时，才对它进行链接 ，便于修改和更新，便于实现目标模块的共享

3.装入：

将装入模块下装到内存

物理地址空间是内存中物理单元的集合，它是地址转换的最终地址，

进程在运行时执行指令和访问数据最后都要通过物理地址从主存中存取。

当装入程序将可执行代码装入内存时，必须通过地址转换将逻辑地址转换成物理地址，这个过程称为地址重定位

绝对装入：编译时，确定程序将存储到内存的位置，产生表示该地址的代码

装入时，根据该代码，将程序装入到对应的位置

适用：单道程序环境，此时只有一个目标模块

优点：不需要地址变换

静态重定位是在程序运行之前由装入程序完成，必须分配其要求的全部连续存储空间。

可重定位装入：装入时进行重定位：修改指令和数据的地址，把逻辑地址转化为物理地址

对于多道程序环境，编译生成多个目标模块，每个目标模块的起始地址都是零，

一次性完成，分配好要求的全部内存空间，如果不够就不能装入

作业一旦装入后也不能再改变在内存中的位置，也不能申请新的内存空间

运行时动态装入：程序装入内存后，并不立即把逻辑地址转换为物理地址，而是在执行时，再计算物理地址

在进程运行时，执行到某条访存指令时，再把逻辑地址转换为主存中的物理地址，

具体实施时，是通过硬件地址转换机制实现的

装入后还有可能在之后换出，所以同一个模块在内存的物理地址是可能改变的

需要重定位寄存器（基址寄存器），存储参考地址的值，相对地址与参考地址相加得到绝对地址

特点：

可以将程序分配到不连续的存储区中

在程序运行之前可以只装入它的部分代码即可投入运行，然后再程序运行期间，根据需要动态申请分配内存

便于程序段的共享

可以实现虚拟内存

重定位：逻辑地址→物理地址

保护

1. 存取控制保护

2. 地址越界保护

逻辑地址：链接阶段形成

物理地址：

内存中的实际地址

静态重定位：装入过程

动态重定位：执行过程

逻辑地址： 是内部和编程使用的，并不唯一

用于逻辑段管理内存

程序员是使用逻辑地址访问设备的

对应的存储空间称为虚拟空间或程序空间

物理地址

是外部链接使用的，且是唯一的，它与“地址总线相对应”

内存中的实际地址就是物理地址

逻辑地址是根据实际访问的进行设置

物理地址是根据存储器的实际情况进行设置

如用含有1024个存储单元的存储器，512个数据

那么物理地址空间是10位

逻辑地址空间是9位

对应的存储空间是内存

编译后一个目标程序所限定的地址范围称为逻辑地址空间，

换句话说，地址空间仅仅是指程序用来访问信息所用的一些列地址单元的集合。

这些单元的编号称为逻辑地址，通常，编译地址都是相对于起始地址0，因为也称逻辑地址为相对地址

要区分编译后形成的逻辑地址与 链接后形成的最终逻辑地址

内存保护：

支持：由操作系统和硬件机构合作实现 ，

功能：内存分配前保护操作系统不受用户进程影响，多进程在主存中彼此互不干扰地运行

方法：

1.界地址保护：每个进程拥有独立的私有存储空间

在CPU中设置一对上、下限寄存器，存放用户作业在主存中的下限和上限地址，每当CPU要访问一个地址时，分别和两个寄存器的值相比，判断有无越界

2：通过重定位寄存器和界地址寄存器（限长寄存器）来实现这种保护

重定位寄存器存储最小的物理地址值，界地址寄存器含有逻辑地址的最大值

装入时，比较逻辑地址与界地址，若未发生越界，则加上重定位寄存器的值后，映射成物理地址，交付给内存单元

为使地址转换不影响到指令的执行速度，必须要有硬件地址变换结构的支持，即需要在系统中增设一个重定位寄存器，用来存放程序和数据在内存中的起始地址

在执行程序或访问数据的时候，真正访问的内存地址是相对地址与重定位寄存器中的地址相加得到的，这时将起始地址存入重定位寄存器，之后的地址访问即可通过硬件变换实现

因为系统处理器在同一时刻只能执行一条指令或访问数据，所以为每道程序或数据设置一个寄存器是没有必要的，同时也不现实，因为寄存器成本很高

只需要在切换程序执行时重置寄存器的内容即可

当某个进程通过调度进入执行状态，派遣程序会初始化重定位寄存器和界地址寄存器，每一个逻辑地址都需要和这两个寄存器进行核对，以保证操作系统和其他用户程序及数据不被该进程的运行所影响

内存保护是内存管理的一部分，是操作系统的任务，但出于安全性和效率考虑，必须由硬件实现

多进程的执行通过内存保护实现互不干扰，如页式管理中有页地址越界保护，段式管理中有段地址越界保护

页表和段表同样存储在内存中，系统提供给用户的物理地址空间为总的空间大小减去页表或段表的长度

分区管理(固定)：

分区：单道程序系统不分区，多道程序系统将内存(用户区)划分为多个分区

分区大小相等，适用于各个进程占存差不多的情况

分区大小不等，事先对进程的大小做好调查

管理分区：分区表，每个表项对应一个分区，包含分区的起始地址、大小、状态(是否已分配)，等信息，按一定顺序排列，每当要使用存储空间时，遍历分区使用表，找到合适的分区来进行分配

碎片：一个较小的进程占领一个较大的分区时，产生内部碎片，浪费存储空间

硬件支持：界地址寄存器、越界检查机构

装入方式：静态重定位

分配：单道程序系统，用户区单一连续分配给单个进程或单道作业

多道程序系统，一个分区分配给一个进程，分区内连续存储

分区管理(动态)：

分区：用户区初始是一整个大空闲区，不进行划分，有分配请求时根据请求空间的大小进行分配，因此没有内部碎片

分区的大小和数量是可变的，分配时，分区数量增加，回收时，分区数量可能减少也可能不变

碎片：随着时间的推移，用户区不断被划分，出现越来越多的小空间，称为外部碎片

硬件支持：上下界寄存器，越界检查机构、基地址寄存器、长度寄存器、动态地址转换机构

优点：满足多道程序设计要求，简单，代价小

回收：作业运行完毕，收回它的分区，回收时，采用拼接技术对空闲区进行合并

将上邻空闲区、下邻空闲区和回收分区合并为一个空闲区，因此空闲区数减少了一个

仅有上邻空闲区、下邻空闲区时，空闲区不减少

# 并行主存系统

主存的主要性能指标：延迟和带宽

以往：Cache主要关心延迟，I/O主要关心带宽。

现在：Cache关心两者

并行主存系统是在一个访存周期内能并行访问多个存储字的存储器。

能有效地提高存储器的带宽。

一个单体单字宽的存储器

字长与CPU的字长相同。

每一次只能访问一个存储字。假设该存储器的访问周期是T，字长为W位，则其带宽为：B=W/T



在相同的器件条件（即TM相同）下，可以采用两种并行存储器结构来提高主存的带宽：

单体多字存储器

多体交叉存储器

单体多字：



如单体m字，每个存储周期读出m个CPU字，最大带宽提高到原来的m倍

B=m×W/T

优点：实现简单

缺点：访存效率低

1.一次读取m个指令字，如果指令字中有分支指令，而且分支成功，那么该分支指令后的指令是无用的

2.一次取出m个数据，不一定都是有用的

另一方面，当前执行指令所需要的多个操作数也不一定正好都存放在同一个长存储字中，由于数据存放的随机性比程序指令存放的随机性大，所以发生这种情况的概率较大

3.必须凑齐m个数才能写入存储器，如果只写个别字，就必须先把响应的长存储字读出来，放到数据寄存器，然后在地址码的控制下修改其中一个字，最后再将该长存储字写回存储器，十分麻烦

4.当读出的数据组和要写入的数据自处于同一个长存储字内时，读和写的操作就无法在同一个存储周期内完成

多体交叉

由多个单字存储体构成，每个体都有自己的地址寄存器以及地址译码和读/写驱动等电路。

问题：对多体存储器如何进行编址？

存储器是按顺序线性编址的。如何在二维矩阵和线性地址之间建立对应关系？



编址

|  |  |
| --- | --- |
| 高位交叉 | 高位交叉编址  对存储单元矩阵按列优先的方式进行编址  特点：同一个体中的高位都是相同的（体号）    处于第i行第j列的单元，即体号为j、体内地址为i的单元，其线性地址为： A＝j×n＋i  j＝０，1，2，…，m－1  i＝０，1，2，…，n－1  一个单元的线性地址为A，则其体号j和体内地址i为：  j=  i＝A mod n  把A表示为二进制数，则其高log2m位就是体号，而剩下的部分就是体内地址。 |
| 低位交叉 | 有效地解决访问冲突问题  对存储单元矩阵按行优先进行编址  特点：同一个体中的低log2m位都是相同的(体号)    处于第i行第j列的单元，即体号为j、体内地址为i的单元，其线性地址为：A＝i×m＋j  i＝０，1，2，…，n－1  j＝０，1，2，…，m－1  一个单元的线性地址为A，则其体号j和体内地址i为：  i=  j＝A mod n  把A表示为二进制数，则其高log2m位就是体号，而剩下的部分就是体内地址。 |

例：采用低位交叉编址的存储器

由8个存储体构成、总容量为64。格子中的编号为线性地址。



为了提高主存的带宽，需要多个或所有存储体能并行工作。

在每一个存储周期内，分时启动m个存储体。

如果每个存储体的访问周期是TM，则各存储体的启动间隔为： t=TM/m。



增加m的值就能够提高主存储器的带宽。但是，由于存在访问冲突，实际加速比小于m。

通过一个模型分析并行主存系统的实际带宽

1.一个由m个独立分体组成的主存系统

2.CPU发出的一串地址为A1，A2，…，Aq的访存申请队列

3.存储控制器扫描这个队列，并截取从头起的A1，A2，…，Ak序列作为申请序列。

(1)申请序列是满足以下条件的最长序列：k个地址所访问的存储器单元都处在不同的分体中。

(2)A1～Ak不一定是顺序地址，只要它们之间不出现分体冲突。

(3)k越接近于m，系统的效率就越高。

4.设P(k)表示申请序列长度为k的概率，用B表示k的平均值，则B=

其中：k=1，2，…，m

每个主存周期所能访问到的字数的平均值，正比于主存实际带宽。

5.P(k)与具体程序的运行状况密切相关。如果访存申请队列都是指令的话，那么影响最大的是转移概率λ。

6.转移概率λ：给定指令的下条指令地址为非顺序地址的概率。

当k＝1时，所表示的情况是：第一条就是转移指令且转移成功。

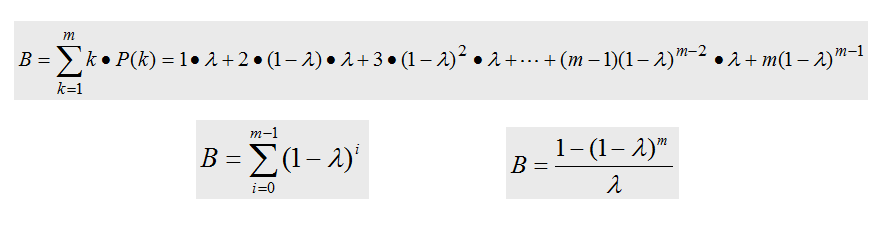
P(1)＝λ＝·λ

当k＝2时，所表示的情况是：第一条指令没有转移（其概率为1－λ），第二条是转移指令且转移成功。所以有：P(2)＝·λ

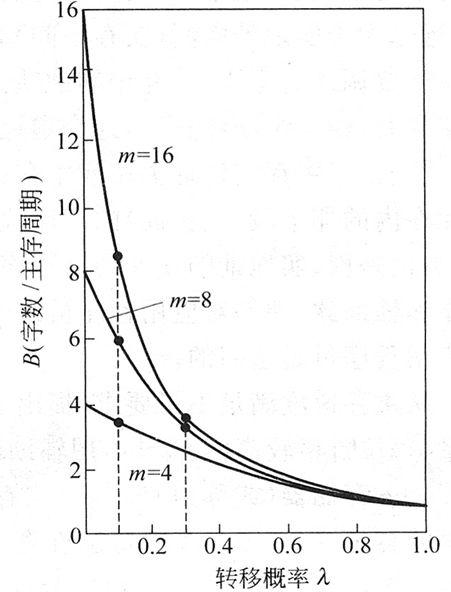
同理，P(3)＝·λ

依此类推，P(k)＝·λ，1≤k＜m

如果前m－1条指令均不转移，则不管第m条指令是否转移，k都等于m，因此有： P(m)=，



7.m等于4、8、16时，B与λ的关系曲线



对于数据来说，由于其顺序性差，m值的增大给B带来的好处就更差一些。

若机器主要是运行标量运算的程序，一般取m≤8。

如果是向量处理机，其m值可以取大些。

8.单纯靠增大m来提高并行主存系统的带宽是有限的，而且性能价格比还会随m的增大而下降。

原因：

程序的转移概率不会很低

数据分布的离散性较大

避免存储体冲突

体冲突：两个请求要访问同一个体。

减少体冲突次数的一种方法：采用许多体

例如，NEC SX/3最多可使用128个体

这种方法存在问题：假如我们有128个存储体，按字交叉方式工作，并执行以下程序：

int x [ 256 ][ 512 ]；

for （ j = 0； j < 512； j = j+1 ）

for （ i = 0； i < 256； i = i+1 ）

x [ i ][ j ] = 2 \* x [ i ][ j ]；

因为512是128的整数倍，同一列中的所有元素都在同一个体内，无论CPU或存储系统多么高级，该程序都会在数据Cache不命中时暂停。

解决体冲突的方法

1.软件方法(编译器)

循环交换优化

扩展数组的大小，使之不是2的幂。

2.硬件方法

使体数为素数

体内地址＝地址A mod (存储体中的字数)

可以直接截取

举例

顺序交叉和取模交叉的地址映像举例



# 虚拟存储

基本原理

虚拟存储器是“主存- 辅存”层次进一步发展的结果。

虚拟存储器可以分为两类：页式和段式

页式虚拟存储器把空间划分为大小相同的块。(页面)

段式虚拟存储器则把空间划分为可变长的块(段)

页面是对空间的机械划分，而段则往往是按程序的逻辑意义进行划分。

Cache和虚拟存储器的参数取值范围

|  |  |  |
| --- | --- | --- |
| 参数 | 第一级Cache | 虚拟存储器 |
| 块（页）大小 | 16-128字节 | 4096-65,536字节 |
| 命中时间 | 1-3个时钟周期 | 100-200个时钟周期 |
| 不命中开销 | 8-200个时钟周期 | 1,000,000-10,000,000个时钟周期 |
| （访问时间） | （6-160个时钟周期） | （800,000-8,000,000个时钟周期） |
| （传输时间） | （2-40个时钟周期） | （200,000-2,000,000个时钟周期） |
| 不命中率 | 0.1-10% | 0.00001-0.001% |
| 地址映像 | 25-45位物理地址到14-20位Cache地址 | 32-64位虚拟地址到25-45位物理地址 |

快速地址转换技术

1.地址变换缓冲器TLB

TLB是一个专用的高速缓冲器，用于存放近期经常使用的页表项；

TLB中的内容是页表部分内容的一个副本；

TLB也利用了局部性原理。

2.TLB中的项由两部分构成：标识和数据

标识中存放的是虚地址的一部分。

数据部分中存放的则是物理页帧号、有效位、存储保护信息、使用位、修改位等。

3.AMD Opteron的数据TLB的组织结构

包含40个项

采用全相联映像

AMD Opteron的地址转换过程

4.一般TLB比Cache的标识存储器更小、更快。

保证TLB的读出操作不会使Cache的命中时间延长。



页式虚拟存储器实例：64位Opteron的存储管理

1.Opteron的页面大小：4KB，2MB和4MB。

2.AMD64系统结构

虚拟地址： 64位 物理地址：52位

进行虚→实地址转换时，是把64位的虚拟地址映射到52位的物理地址。

要求：64位虚拟地址中的高16位是由低48位进行符号位扩展而来的

规范格式

3.采用多级分层页表结构来映射地址空间，以便使页表大小合适。

分级的级数取决于虚拟地址空间的大小

Opteron的48位虚拟地址的4级转换

每个分级页表的偏移量分别来自4个9位的字段

4.Opteron的每一级页表都采用64位的项

其中：

前12位留给将来使用

随后的52位是物理页号

4.最后的12位包括保护和使用信息。

不同级的页表中有所不同，但大都包含以下基本字段：

存在位：说明该页面在存储器中。

读/写位：说明该页面是只读还是可读写。

用户/管理位：说明用户是否能访问此页或只能由上面的3个特权级所访问。

修改位：说明该页面已被修改过。

访问位：说明自上次该位被清0后到现在，该页面是否被读或写过。

页面大小：说明最后一级页面是4KB还是4MB；如果是4MB，则Opteron仅使用三级页表而非四级。

非执行位：在有些页面中用来阻止代码的执行。

页级Cache使能：说明该页面能否进入Cache。

页级写直达：说明该页是允许对数据Cache进行写回还是写直达。

5.Opteron通常在TLB不命中时要遍历所有四级页表，故有3个位置可以进行保护限制的检查。

仅遵从底层的PTE，而在其他级上只需确认有效位是有效的即可。

6.在保护方面，如何避免用户进行非法的地址转换？

页表本身已经被保护，用户程序无法对它们进行写操作。

操作系统通过控制页表项来控制哪些物理地址可以被访问，哪些不能访问。

多个进程共享存储器是通过使各自的地址空间中的一个页表项指向同一个物理页面来实现的。

7.Opteron使用4个TLB以减少地址转换时间

两个用于访问指令，两个用于访问数据。

8.和多级Cache类似，Opteron通过采用两个更大的第二级TLB来减少TLB不命中。

一个用于访问指令

另一个用于访问数据

9.Opteron中第一级和第二级指令、数据TLB的参数

|  |  |
| --- | --- |
| 参数 | 描述 |
| 块大小 | 1个 PTE（8字节） |
| L1命中时间 | 1个时钟周期 |
| L2命中时间 | 7个时钟周期 |
| L1 TLB大小 | 指令和数据TLB都是40个PTE，其中32个用于  4KB大小的页面，8个用于2MB或4MB页面。 |
| L2 TLB大小 | 指令和数据TLB都是512个PTE，用于4KB页面 |
| 块选择 | LRU |
| L1映像规则 | 全相联 |
| L2映像规则 | 4路组相联 |

实例：AMD Opteron的存储器层次结构

1.一个乱序执行处理器

每个时钟周期最多可以取出3条80x86指令，并将之转换成类RISC操作，然后以每个时钟周期3个操作的速率流出。

2.有11个并行的执行部件

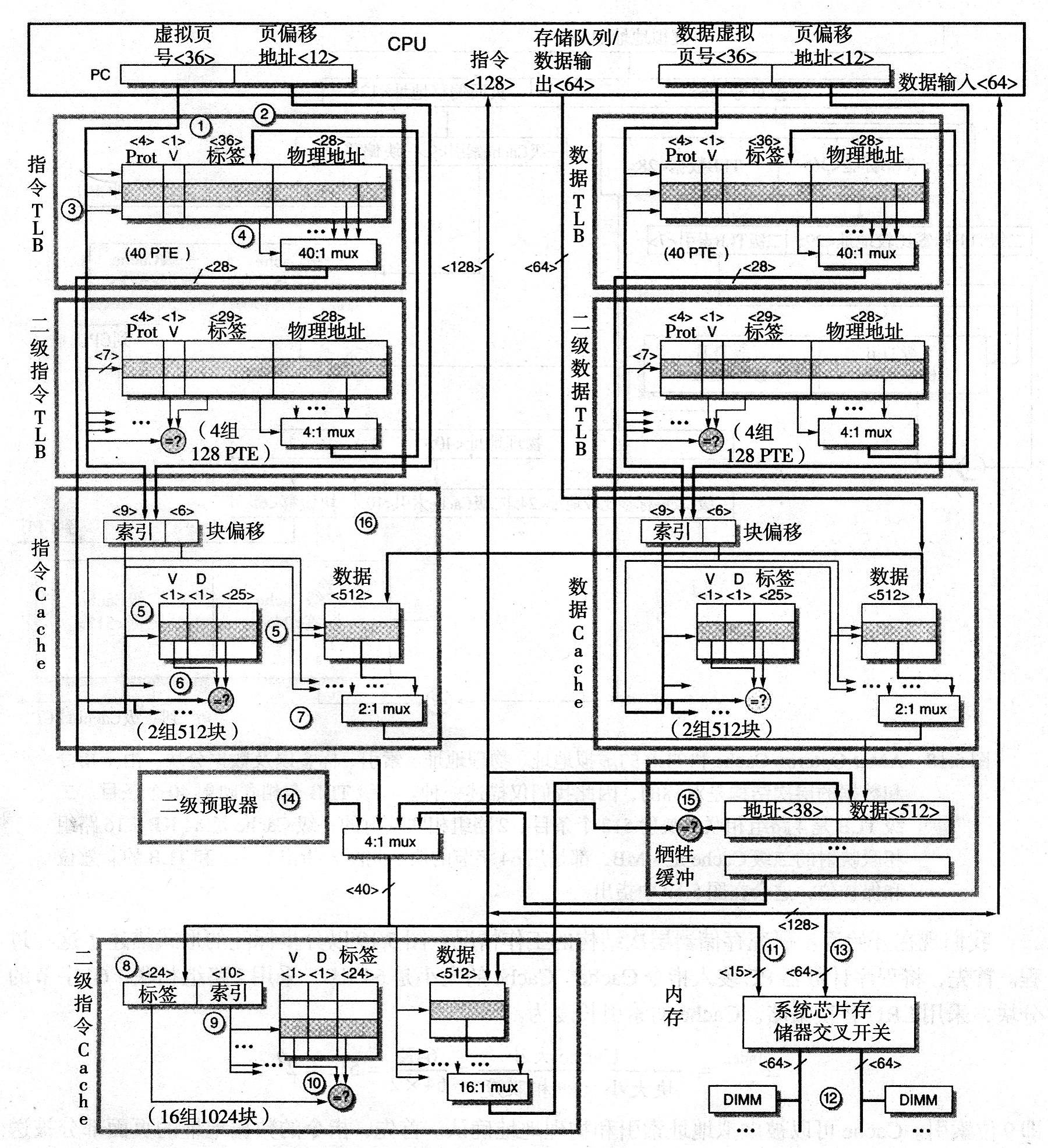
3.在2006年，其12级定点流水线使得该处理器的最高时钟频率达到了2.8GHz。

4.虚地址：48位 物理地址：40位

5.通过两级TLB实现的从虚拟地址到物理地址的转换以及对两级数据Cache的访问情况



6.AMD Opteron存储器层次结构图

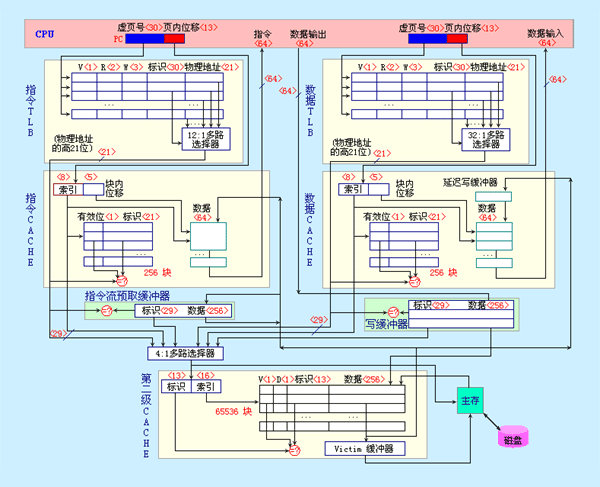


Alpha AXP 21064存储层次

[AlphaAXP21064地址转换过程](5-7-1.exe)

[简介](5-9-1.exe)

[工作过程](5-9-2.exe)



# Cache

|  |  |
| --- | --- |
| 问题 | 1.I/O设备箱主存请求的级别高于CPU访存，会出现CPU等待I/O设备访存的现象，使CPU空等，降低了CPU的效率  2.主存的速度跟不上CPU的速度 |
| 硬件实现 | 借助于辅助硬件，Cache与主存构成一个整体，Cache对应用程序员和系统程序员都是透明的 |
| 分块管理 | Cache和主存均被分割成大小相同的块，信息以块为单位调入Cache  主存地址=块地址+块内位移 |
| 存储内容 | 复制内存的一部分内容 |
|  | 为提高访问速度，一般是把“主存→Cache”地址变换和访问Cache存储体安排成同时进行。  这时，由于还不知道哪个候选位置上有所要访问的数据，所以就把所有候选位置中的响应信息都读出来，在“主存→Cache”地址变换完成后(如果命中)，再根据其结果从这些信息中选一个，发送给CPU |
|  | Cache不命中，对访问目标块执行块传输(block)：内存→Cache  Cache内容已满，无法接受来自主存的信息  Cache内的替换机构按一定的替换算法来确定应从Cache内移出哪个块返回主存， 而把新的主存块调入Cache。有关替换算法  Cache对用户是透明的，即用户编程时所用到的地址是主存地址，用户根本不知道这些主存块是否已经调入到Cache内，因为主存块调入Cache的任务全由机器硬件自动完成 |



Cache使CPU不用直接访问主存，而与高速Cache交换信息

将CPU近期要用到的程序和数据体检从主存送到Cache，那么就可以做到CPU在一定时间内只需要访问Cache

Cache采用高速的SRAM存储器。价格比主存贵，但其容量不需要很大，比主存小得多

Cache的容量与块长时影响Cache效率的重要因素，效率用命中率来衡量

一个程序的执行期间，访问Cache的命中次数是Nc，访问主存(未命中)的次数是Nm，那么

命中率h=Nc/(Nc+Nm )

设tc 为命中时的Cache访问时间，tm 为未命中时的主存访问时间

那么Cache-主存系统的平均访问时间ta

ta=htc+(1−h)tm

应该使用尽量小的硬件代价，使得Cache-主存系统的平均访问时间ta 越接近tc 越好，用e表示访问效率

e=tc/(htc+(1−h)tm )×100%

为提高访问效率，命中率h越接近1越好

一般Cache容量越大，命中率越高，但成本也越高

另外，容量达到一定值时，命中率不再随容量的增大而又明显的提高。因此会选取一个适当的Cache容量

如80836主存4GB，配套的缓存时16KB或32KB，命中率也可以达到95%以上

Cache结构

存储体 存储体以块为单位与主存交换信息

为加速Cache与主存之间的调动，主存大多采用多体结构，且Cache访存的优先级最高

读写操作

当CPU发出主存地址，首先判断该存储字是否在Cache中，若命中，直接访问Cache，将该字所在的主存块装入Cache，如果此时Cache已经装满，就要执行替换算法

 一般Cache使用高速SRAM制作  
掉电后数据消失

## 工作过程

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cache  Map | fully associative  全相联映射 | | | 映射到任一缓存块  空间利用率最高，冲突概率最低，实现最复杂 | | | | | | |
| direct  直接映射 | | | 主存第b块映像到Cache的第index块，  index＝b mod M，M为Cache的块数，M= | | | | | | |
| Set Mapping  组相联映射  n路组相联  n：相联度 | | | Cache按序分组，每组n块，共G组，  主存第b块映像到Cache第index组任一块，  index＝b mod G ，G为Cache的组数，G＝  n=1：直接映射，n=M：全相联映射  n越大，命中率越高，复杂度和成本越高，一般n≤4 | | | | | | |
| Cache  Search | [目录表](file:///F:\CS\Architecture\5-2-7.exe) | 目录表存储在单体多字存储器，表长=M，M=G\*n  每个Cache块对应一个[目录表](file:///F:\CS\Architecture\5-2-7.exe)表项 | | | | | | | | |
| 表项 | | | tag | | 每个内存块有一个唯一的标识 | | | |
| offset | |  | | | |
| 有效位 | | 指出该Cache块是否包含有效信息 | | | |
| 内存地址 | 内存块号+块内偏移量=tag(h\*bit)+index(g\*bit)+offset(g\*bit)  全相联映射g=0，直接映射h=0 | | | | | | | | |
| 比较 | 【读出数据并送往CPU】与【读出标识并进行比较】两个过程并行执行 | | | | | | | | |
| 缓存地址 | index×blocksize+offset | | | | | | | | |
| 相联存储器  G\*n\*(h+) bit | | | | |  | | 第1块 | … | 第n块 |
| 第1组 | |  | … |  |
| … | |  | … |  |
| 第G组 | |  | … |  |
| 单体多字存储器  (G\*n\*h) bit  n\*比较器 | | | | |  | | 第1块 |  | 第n块 |
| 第1组 | |  | … |  |
| … | | … | … | … |
| 第G组 | |  | … |  |
|  | | ⇩  比较器1 | … | ⇩  比较器n |
| 选择第index组，读出n个tag，输入比较器，与addr中的tag比较，n个比较器并行比较， | | | | |
|  | | | | |  | | | | |
| Cache Miss | compulsory  强制性失效 | | 某内存块第一次访问，必然不再Cache中  只能通过增大容量来减少 | | | | | | | |
| Capacity  容量失效 | | 已装满，某些块被替换，又被重新访问 | | | | | | | |
| Conflict  冲突失效 | | 冲突/碰撞/干扰不命中  在组相联或直接映像Cache中，若太多的块映像到同一组中，则会出现在组中某个块被别的块替换、然后又被重新访问  采用全相联就不会发生Conflict miss | | | | | | | |

|  |  |  |
| --- | --- | --- |
| Cache  Replace | 当新调入一块，而Cache又已被占满时，替换哪一块？  直接映像Cache中的替换很简单因为只有一个块，别无选择。  在组相联和全相联Cache中，则有多个块供选择。 | |
| RAND | 采用一个随机数产生器产生一个随机的被替换的块  不能提高命中率  实现简单 |
| FIFO | 先进先出：选择最早调入Cache的字块进行替换  不需要记录各个字块的使用情况，比较容易实现，但没有根据访存的局部性原理，故不能提高命中率  因为最早调入的信息可能以后还会用到，或经常要用到，如循环程序 |
| LRU | 近期最少使用：选择最久没有被访问过的块作为被替换的块，  利用时间局部性原理，替换近期用的最少的字块  需要随时记录Cache中各字块的使用情况，以便确定哪个字块是近期最少使用的字块。  命中率较高，但成本高，机制复杂  LRU和随机法分别因其不命中率低和实现简单而被广泛采用。对于容量很大的Cache，LRU和随机法的命中率差别不大 |

|  |  |  |
| --- | --- | --- |
| Cache  Write | 写一般比读花费更多时间，应为只有在读出标识并进行比较、确认命中后，才可以写入，所以检查标识不能与写入Cache并行执行  另外，处理器要写入的数据的宽度不是定长的，写入是，只能修改Cache块中相应的部分，而读则多读出几个字节也没关系  Cache一直是内存部分内容的一个副本，必须保持Cache与主存内容一致写访问可能导致Cache与主存对应的内容不一致：  CPU对cache执行写操作，但没有立即写主存  I/O设备或I/O处理机写主存，但没有同时写Cache | |
| write  through | 数据写入时，同时写入主存和Cache  写操作时，数据既写入Cache，又写入主存，它能随时保证主存和Cache的数据始终一致，但增加了访问次数  由于主存和Cache的数据始终一致，在读操作Cache失效时，只需选择一个替换的块（主存块）调入Cache，被替换的块(Cache块)不必写回主存。  可见读操作不涉及对主存的写操作，因此这种方法更新策略比较容易实现  但是在写操作时，既要写入主存又要写入Cache，因此其写操作的时间就是访问主存的时间 |
| write  back | 数据写入Cache时，只写入Cache，仅当Cache块换出时，把它写回主存  数据的最新版本总在Cache中  简单法：不管字块是否被更新，都执行写回操作  标识法：只在更新时，才进行写回  为每个Cache块设置一个修改位，用于指出该块是否被修改过，一个块被替换时，如果没有被修改过，则不必写回  写操作时只把数据写入Cache，而不写入主存，但当Cache数据被替换出去时才写回主存。  可见写回法Cache中的每一块要增设一个标志位，该位有两个状态：  "清"：表示未修改过，与主存一致  "浊"：表示修改过，与主存不一致  在Cache替换时，"清"的Cache块不必写回主存，因为此时主存中相应块的内容与Cache块是一致的  在写Cache时，要将该标志位设置为"浊"，替换时次Cache块要写回主存，同时将标志位置"清"  写操作只写入Cache，故写操作时间就是访问Cache的时间，因此速度快  这种方法对主存的写操作只发生在块替换时，而且对Cache中一个数据块的多次写操作只需一次写入主存，因此可减少主存的写操作次数。但在读操作Cache失效时要发生数据替换，引起被替换的块写回主存的操作，增加了Cache的复杂性。 |

写操作：

写入的信息必须与被映射的主存块内的信息完全一致。当程序运行过程中需对某个单元进行写操作时，会出现如何使Cache与主存内容保持一致的问题。目前采用以下几种方法：

写访问不需要用到所访问单元中的数据，当发生Cache不命中时，是否调入相应的块？

|  |  |
| --- | --- |
| write allocate  按写分配法 | 写不命中时，先把所写单元所在的块从主存调入Cache，然后再进行写入。 |
| no-write allocate  不按写分配法 | 写不命中时，直接写入下一级存储器而不将相应的块调入Cache |

一般write back配合write allocate，write through配合no-write allocate

## LRU算法的硬件实现

堆栈法：

用一个堆栈记录组相联Cache的同一组中各块被访问的先后次序

这个先后次序是用堆栈元素的物理位置来反映的，

栈底：最早访问的块

次栈底：该组中第二个被访问过的块

……

栈顶：刚访问的块

当需要替换时，替换栈底的块



|  |  |
| --- | --- |
| 堆栈内容  动态更新 | 每次访问Cache，堆栈中的内容更新，  Cache命中：用块地址进行相联查找，在堆栈中找到相应的元素，暂存，该元素的上面所有元素下压一个位置，本次访问的块地址被覆盖，但已事先保存，该地址入栈  不命中：把本次访问的块地址入栈，如果堆栈已满，则栈底的块移除 |
| 所需硬件 | 为每一组设置一个表长与相联度相同的堆栈，堆栈表项位数是  硬件堆栈功能要求：  1.相联比较功能  2.能够全部下移，部分下移和从中间取出一项的功能，  成本较高，只适用于相联度较小的LRU |
|  |  |

比较对法

不使用相联比较

让各块两两组合，构成比较对，每一个比较对用一个触发器的状态来标识它所相关的两个块最近一次被访问的远近次序，再经过门电路就可以找到LRU块

如A、B、C三块，组成3对：AB、AC、BC

每一对中块的访问次序分别用触发器TAB、TAC、TBC 表示

TAB=1表示A比B更近被访问过，TAB=表示B比A更近被访问过

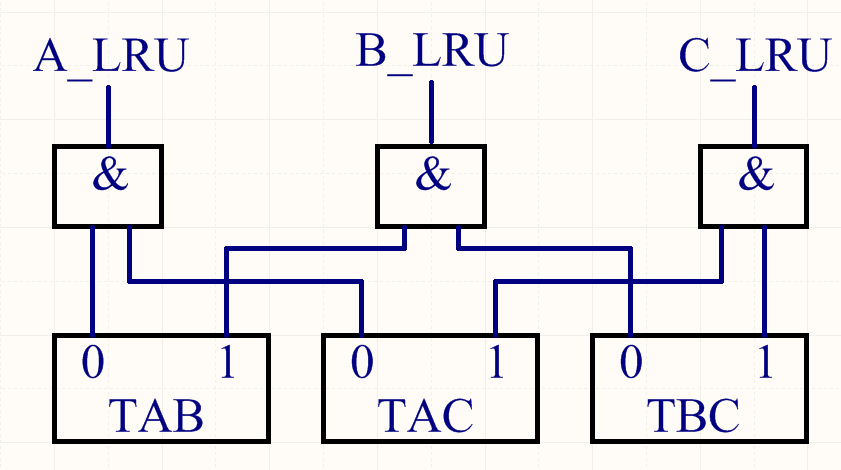
=1表示C是近期最少使用的块，=0表示C不是近期最少使用的块，

=TAC·TBC

=·

=TAB·

使用触发器和门电路实现：



每次访问Cache时，更新与访问的块有关的比较对触发器的状态，在上述例子中，每次访问A后，TAB=1，TAC=1

|  |  |
| --- | --- |
| 所需硬件 | 有多少Cache块就有多少个与门，每个与门的输入端要连接所有与之相关的触发器。  对于一个具有P块的组中的任何一块，由于它可以跟除了它自己以外的所有其他块两两组合，所以与该块相关的比较对触发器个数为P-1  总共需要P(P-1)/2  当组内块数较多时，可以用多级状态位技术来减少所需要的硬件量  如IBM3033，组内块数为16，可分成群、对、行3级，先分成4个群，每群两对，每队两行。这样选LRU群需6个触发器，每群中选LRU对需1个触发器，4个群共4个触发器  每行中选LRU块需要1个触发器，8行共8个触发器，选LRU  总个数是18个，但牺牲了速度  而单级需要120个， |
|  |  |

触发器：所需要的触发器的个数与两两组合的比较对的数目相同。

比较对触发器个数、与门的个数、与门的输入端数与块数P的关系

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 组内块数 | 3 | 4 | 8 | 16 | 64 | 256 | … | P |
| 触发器个数 | 3 | 6 | 28 | 120 | 2016 | 32640 | … |  |
| 与门个数 | 3 | 4 | 8 | 16 | 64 | 256 | … | P |
| 与门输入端个数 | 2 | 3 | 7 | 15 | 63 | 255 | … | P-1 |

块数少时，所需要的硬件较少，

随着组内块数P的增加，所需的触发器的个数会以平方的关系迅速增加，门的输入端数也线性增加。(硬件实现的成本很高)

当组内块数较多时，可以用多级状态位技术减少所需的硬件量。

例如：在IBM 3033中

组内块数为16，可分成群、对、行3级。

先分成4群，每群两对，每对两行。

选LRU群需6个触发器；

每群中选LRU对需要一个触法器，4个群共需要4个触发器；

每行中选LRU块需要一个触发器，8个行共需要8个触发器。

所需的触发器总个数为：

6（选群）＋4（选对）＋8（选行）= 18（个）

以牺牲速度为代价的。

从主存调入一个块到Cache中时，候选位置全被占用

需要选择其中某一块，用新调入的块取而代之

在直接映射的Cache中，由于某个主存块只与一个Cache字块有映射关系，替换算法很简单

全相联映射：可以从已被占满的Cache中替换出任一旧字块

## 举例：DEC的Alpha AXP21064中的内部数据Cache

简介

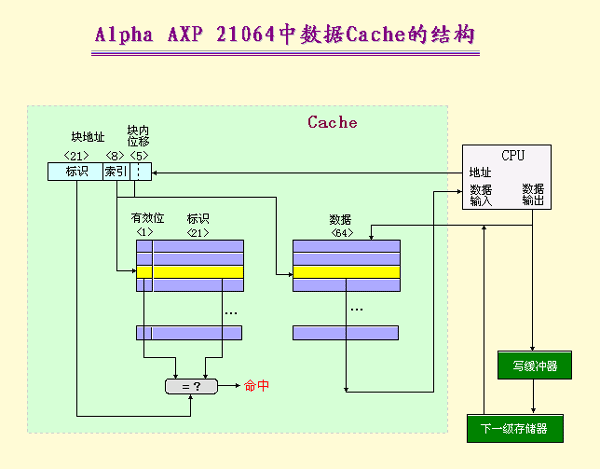
容量：8KB

块大小：32B

块数：256

映像方法：直接映像

写缓冲器大小：4个块



工作过程

|  |  |
| --- | --- |
| [读](file:///F:\CS\Architecture\5-2-17.exe)命中 | 完成4步需要2个时钟周期  Cache的容量与索引index、相联度、块大小之间的关系：  Cache的容量=×相联度×块大小  把容量为8192、相联度为1、块大小为32（字节）代入：  索引index：8位 标识：29－8＝21位 |
| [写](file:///F:\CS\Architecture\5-2-18.exe)命中 | 设置了一个写缓冲器，提高写访问的速度  按字寻址的，它含有4个块，每块大小为4个字。  当要进行写入操作时，如果写缓冲器不满，那么就把数据和完整的地址写入缓冲器。对CPU而言，本次“写”访问已完成，CPU可以继续往下执行。由写缓冲器负责把该数据写入主存。  在写入缓冲器时，要进行写合并检查。即检查本次写入数据的地址是否与缓冲器内某个有效块的地址匹配。如果匹配，就把新数据与该块合并 。 |
| CacheMiss | 读不命中：向CPU发出一个暂停信号，通知它等待，并从下一级存储器中新调入一个数据块（32字节）。  写不命中：将使数据“绕过”Cache，直接写入主存。 |

对比

Alpha AXP 21264的数据Cache结构

容量：64KB 块大小：64字节 LRU替换策略

主要区别

采用2路组相联

采用写回法

没有写缓冲器



Cache中存放的是内存的一部分副本

TLB 中存放的是页表的一部分副本

在同时具有TLB和Cache的系统中，CPU发出访存命令，

1.查找Cache

若Cache命中，则说明所需内容在Cache内，其所在页面必定在内存中，即page必定命中，但TLB不一定命中

若Cache未命中，并不能说明所需内容未调入内存，这与 TLB、Page命中与否 没有联系

若TLB命中，Page必定命中

若Page命中，TLB未必命中

取数时，可能不需要访问内存而直接访问Cache

而写直通的方式需要把数据同时写入Cache和主存，因此完成该指令功能，至少访问主存一次

缓存就是复制内存的一部分到

复制的是虚拟内存还是物理内存?

如果复制的是虚拟内存，那么使用逻辑地址直接访问缓存

如果复制的是物理内存，那么就得先将逻辑地址转化为物理地址，再访问Cache

Cache技术

根据时间局部性，将近来使用的指令和数据保存到高速缓冲存储器中，并使用高速缓存的层次结构实现

根据空间局部性，通常使用较大的高速缓存，并将预取机制集成到高速缓存控制逻辑中实现

Cache是一个高速的小容量存储器

是CPU的访存操作大多数针对Cache进行，从而提高程序的执行速度

Cache的总容量包括：存储容量和标记阵列容量（有效位、标记位、一致性维护位，替换算法控制位）

标记阵列中的有效位和标记位是一定有的，

一致性维护位(脏位)和替换算法控制位的取舍标准，看情况，如果是使用写回法，则一定包含一致性维护位

由于Cache的容量远小于内存，所以Cache块数也远小于内存块数

Cache仅仅是保存主存中最活跃的若干块的副本

Cache按照某种算法，预测CPU在未来的一段时间内与访问的内存的数据，将其装入Cache

CPU发出读请求时，如果访存地址在Cache命中， 就将此地址转换成Cache地址，

直接对Cache进行读操作，主存不参与本次操作

如果不命中，则仍需要访问内存，并把此字所在的块一次从主存调入Cache内，

若此时Cache已满，则需要根据某种替换算法，替换掉原来在Cache中的某块信息

CPU与Cache之间的数据交换以字为单位，而Cache与主存之间的数据交换则是以Cache块为单位

当CPU执行写操作时，如果Cache命中，那么就会修改Cache中某个块的数据，

那么这个块与其在内存中对应的块，数据不一致，需要解决这个问题，常见的有全写法和写回法

命中率：CPU与访问的信息已经在Cache中的比率，显然命中率越高越好

需要把内存汇总某些块复制到Cache中，如内存中地址为A的块，复制到Cache中地址为B的块

需要为Cache中的每一个块加一个标记，表明它是主存中哪一块的副本

注意地址映射不同于地址变换，地址变换是CPU访存时，将主存地址按映射规则换算成Cache地址的过程。

把指令Cache和数据Cache分离，取指和取数分别到不同的Cache中寻找，

那么指令流水线中的取指部分和取数部分就可以很好地避免冲突，即减少了指令流水线的冲突

将CPU送来的主存地址转换为Cache地址

由于主存的块和Cache的块大小相同，块内地址都是相对于块的其实地址的偏移量

因此地址变换主要是主存的块号(高位地址)于Cache块号间的转换

而地址变换又与主存地址以什么样 的函数关系映射到Cache中(地址映射)有关

如果转换后的Cache已经和CPU欲访问的主存建立了对应关系，即已经命中

那么CPU可直接访问Cache存储体。

CPU读取主存的某个存储单元时，有两种可能：

1.所需要的字已经在缓存中，则访问Cache命中，

即可直接访问Cache(CPU和Cache之间通常一次传输一个字)

如果主存块已经调入缓存快，则称该主存快与缓存块建立了对应关系。

该标记的内容相当于主存块的编号，

CPU读信息时，要将主存地址的高m位(或m位中的一部分)与缓存快的标记进行比较，

以判断所读的信息是否已经在缓存中

字块：

将主存和Cache以字为单位进行分块，每一个字块的容量是一个字，

每一个字块的首个存储单元的地址作为该字块的地址；

由于缓存的容量远小于主存，缓存字块的数目也远小于主存字块的数目，

显然缓存字块不可能和主存字块一一对应；

高t位作为字块标记，

拥有相同字块标记的缓存字块和主存字块就是相对应的关系

设Cache字块的地址是c位的，那么可以有2^(c−1) 个缓存字块，按地址递增的顺序，缓存字块的标志0~2^(c−1)

设主存字块的地址是m位的，那么可以有2^(m−1) 个主存字块，按地址递增的顺序，

标志为0的主存字块：字块0、字块2^c 、字块2^(c+1)……

标志为1的主存字块：字块1、字块2^c+1、字块2^(c+1)+1……

标志为2^(c−1) 的主存字块：字块2^(c−1) 、字块2^(c−1)+2^c 、字块2^(c−1)+2^(c+1)……

## Cache优化

降低不命中率

|  |  |
| --- | --- |
| 增加块大小 | 降低不命中率  但是，增大Cache块会产生双重作用：  1.增强空间局部性，减少compulsory miss  2.Cache容量不变的情况下，减少Cache块数量，增加了Conflict miss |
| 增加Cache容量 | 降低不命中率  增加成本，增加命中时间 |
| 提高相联度 | 在降低不命中率方面，8路组相连的作用已经和全相联一直，即采用相联度超过8的方案的实际意义不大  2:1Cache经验规则：容量为N的直接映像Cache的不命中率和容量为N/2的2路组相联Cache的不命中率差不多  一般，改进平均访存时间的某一方面是以损失另一方面为代价的  如增加块大小，增加不命中开销  提高相联度会增加命中时间 |
| 伪相联 | 访问时，按与直接映像相同的方式进行访问  如果命中，访问  但如果不命中，检查Cache另一个位置块，看是否匹配，确定这个“另一块”的一种方法是将索引字段的最高位取反，然后按照新索引区寻找“伪相联组”中的对应块，如果匹配，则发生伪命中 |
| 硬件预取 |  |
| 编译器控制的预取 |  |
| 编译器优化 |  |
| 牺牲Cache |  |

减少不命中开销

|  |  |
| --- | --- |
| 两级Cache |  |
| 读不命中优于写 |  |
| 写缓冲合并 |  |
| 请求字处理技术 | 1.请求字：从下一级存储器调入Cache的块中，只有一个字是立即需要的。这个字称为请求字。  2.应尽早把请求字发送给CPU  尽早重启动：调块时，从块的起始位置开始读起。一旦请求字到达，就立即发送给CPU，让CPU继续执行。  请求字优先：调块时，从请求字所在的位置读起。这样，第一个读出的字便是请求字。将之立即发送给CPU。  3.这种技术在以下情况下效果不大：  Cache块较小  下一条指令正好访问同一Cache块的另一部分 |
| 非阻塞Cache |  |

减少命中时间

|  |  |
| --- | --- |
| 小而简单Cache |  |
| 虚拟Cache | 可以直接用虚拟地址进行访问的Cache，其标识存储器中存放的是虚拟地址  地址检测也是用的虚拟地址  CPU访问存储器时，把虚拟地址同时送给Cache和MMU |
| Cache访问流水化 |  |
| 踪迹Cache |  |

# Disk

Cache-主存：

借助于辅助硬件，Cache与主存构成一个整体，这个层次的工作由硬件实现，对于应用程序员和系统程序员都是透明的

## RAID

磁盘阵列DA（Disk Array）：使用多个磁盘（包括驱动器）的组合来代替一个大容量的磁盘。

多个磁盘并行工作。

以条带为单位把数据均匀地分布到多个磁盘上。

（交叉存放）

条带存放可以使多个数据读/写请求并行地被处理，从而提高总的I/O性能。

这里并行性有两方面的含义：

多个独立的请求可以由多个盘来并行地处理。

减少了I/O请求的排队等待时间

如果一个请求访问多个块，就可以由多个盘合作来并行处理。

提高了单个请求的数据传输率

问题：阵列中磁盘数量的增加会导致磁盘阵列可靠性的下降。

如果使用了N个磁盘构成磁盘阵列，那么整个阵列

的可靠性将降低为单个磁盘的1/N。

解决方法：在磁盘阵列中设置冗余信息盘

当单个磁盘失效时，丢失的信息可以通过冗余盘中

的信息重新构建。

廉价磁盘冗余阵列

Redundant Arrays of Inexpensive Disks

磁盘冗余阵列

Redundant Arrays of Independent Disks

简称盘阵列技术

1988年，Patterson教授首先提出。

大多数磁盘阵列的组成可以由以下两个特征来区分：

数据交叉存放的粒度

（可以是细粒度的，也可以是粗粒度的）

细粒度磁盘阵列是在概念上把数据分割成相对较小

的单位交叉存放。

优点：所有I/O请求都能够获得很高的数据传输率。

缺点：在任何时间，都只有一个逻辑上的I/O在处

理当中，而且所有的磁盘都会因为为每个请求进行

定位而浪费时间。

粗粒度磁盘阵列是把数据以相对较大的单位交叉存放。

多个较小规模的请求可以同时得到处理。

对于较大规模的请求又能获得较高的传输率。

冗余数据的计算方法以及在磁盘阵列中的存放方式

在磁盘阵列中设置冗余需要解决以下两个问题：

如何计算冗余信息?

大多都是采用奇偶校验码；

也有采用汉明码（Hamming code）或Reed-Solomon码的。

如何把冗余信息分布到磁盘阵列中的各个盘?

有两种方法：

把冗余信息集中存放在少数的几个盘中。

把冗余信息均匀地存放到所有的盘中。

（能避免出现热点问题）

RAID的分级及其特性

磁盘阵列Disk Array是使用多个磁盘，包括驱动器，的组合来歹意一个大容量的磁盘

多个磁盘可以并行工作，提高性能、

以条为单位把数据均匀分布到多个磁盘上，交叉存放

条带存放使得磁盘存储器系统可以并行地处理多个数据读/写请求

并行性：

1.多个独立的请求可以由多个磁盘来并行地处理。这减少了I/O请求的排队等待时间

2.一个请求如果是访问多个块，就可以由多个磁盘合作来并行处理。这提高了单个请求的数据传输率

阵列中磁盘的个数越多，性能越高，但可靠性降低

N个磁盘构成的阵列可靠性降为1/N

可以通过在磁盘阵列中设置冗余信息盘来解决这个问题，当单个磁盘失效时，丢失的信息可以利用冗余盘中的信息重新构建。只有在这个失效磁盘被恢复(修改或更换)之前，又发生了第二个磁盘的失效时，磁盘阵列才不能正常工作

由于磁盘的MTTF为几十年，MTTR只有几个小时，所以容错技术使得磁盘阵列的可靠性比单个磁盘高很多

这种磁盘称为RAID廉价磁盘冗余阵列

磁盘阵列RAID

将多个独立的物理磁盘组成一个独立的逻辑盘，数据在多个物理盘上分割交叉存储、并行访问，具有更好的存储性能、可靠性和安全性

特征

|  |  |  |
| --- | --- | --- |
| 数据交叉存放的粒度 | 细粒度 | 概念上把数据分割成较小的单位交叉存放  几乎所有的I/O请求，不管大小，都会访问磁盘阵列中的所有磁盘，那么所有I/O请求都能获得很高的数据传输率  缺点：任何时间，都只有一个逻辑上的I/O在处理当中，而且所有的磁盘都会因为为每个请求进行定位而浪费时间 |
| 粗粒度 | 概念上把数据分割成较大的单位交叉存放  规模较小的I/O请求只需要访问数量较少的几个磁盘，只有较大规模的请求才会访问到所有磁盘，多个较小规模的请求可以同时得到处理，而对于较大规模的请求来说又能获得较高的传输率 |
| 冗余数据  计算方法  在磁盘阵列中的存放方式 |  | |

问题

|  |  |
| --- | --- |
| 计算冗余信息 | 大多采用奇偶校验码  也有汉明码或Reed-Solomon码 |
| 把冗余信息  分布到RAID中的各个盘 | 方法1：把冗余信息集中存放在少数几个盘中  方法2：把冗余信息均匀存放到所有盘中  一般采用方法2，因为可以避免出现热点问题 |

在RAID中增加冗余信息盘有几种不同的方法，构成不同的RAID级别

分级

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 可以容忍  故障个数 | 8个数据盘  需检测盘个数 | 优点 | 缺点 |
| RAID0 | 0 | 0 |  | 无纠错能力 |
| RAID1 | 1 | 8 | 不需要计算奇偶校验，  数据恢复快，读数据快。  小规模的写操作比高级别的RAID还要快 | 检测空间开销最大，  即需要的检测盘最多 |
| RAID2 | 1 | 4 | 不依靠故障盘进行自行诊断 | 检测空间开销的级别是，m为数据盘个数 |
| RAID3 | 1 | 1 | 检测空间开销小，  小规模的读操作带宽较高 | 对小规模、随机的读/写操作没有提供专门的支持 |
| RAID4 | 1 | 1 | 检测空间开销小，  小规模的读操作带宽较高 | 校验盘是小规模写的瓶颈 |
| RAID5 | 1 | 1 | 检测空间开销小  小规模的读操作带宽较高 | 小规模写需要访问磁盘4次 |
| RAID6 | 2 | 2 | 能容忍2个故障 | 小规模写需要访问磁盘6次  与3、4、5相比，检测空间开销加倍 |

|  |  |
| --- | --- |
| RAID0 | 无冗余，无校验，条带存放，严格地说不属于RAID系列  存储：连续多个数据块交替地存放在不同的物理磁盘的扇区  访问：几个磁盘交叉并行读写  可以容忍故障个数：0  8个数据盘需检测盘个数：0  优点：扩大存储容量，提高数据存储速度，无空间开销  缺点：没有容错能力 |
| RAID1  镜像盘 | 存储：互为镜像的两磁盘同时读写，互为备份  访问：如果一个故障，可从另一个读出数据，两个磁盘当一个磁盘使用  可以容忍故障个数：  8个数据盘需检测盘个数：  优点：读取快速，写速度较快，可靠(数据恢复很简单)  缺点：成本高 |
| RAID2 | 存储：每个数据盘存放所有数据字的一位(位交叉存放)  ECC盘：各数据盘相应位计算汉明码，存放在多个ECC盘的对应位  个数，m：数据盘的个数(数据字的位数)  访问：  可以容忍故障个数：  8个数据盘需检测盘个数：  优点：  缺点： |
| RAID3  位交叉  奇偶校验 | 存储：  访问：  可以容忍故障个数：  8个数据盘需检测盘个数：  优点：  缺点： |
| RAID4  块交叉  奇偶校验 | 存储：  访问：  可以容忍故障个数：  8个数据盘需检测盘个数：  优点：有效处理小规模访问，快速处理大规模访问，校验空间开销小  缺点：  磁盘阵列  采用比较大的条带，以块为单位进行交叉存放和计算奇偶校验。  实现目标：能同时处理多个小规模访问请求    读：每次只访问数据所在的磁盘，仅在该磁盘出现故障时，才会去读校验盘，并进行数据的重建。  写：假定有4个数据盘和一个冗余盘，写数据需要2次磁盘读和2次磁盘写操作。 |
| RAID5  块交叉  奇偶校验  无独立校验 | 存储：数据以块交叉的方式存于各盘，无专用冗余盘，奇偶校验信息均匀分布在所有磁盘上。  访问：  可以容忍故障个数：  8个数据盘需检测盘个数：  优点：  缺点： |
| RAID6  P+Q双校验 | 存储：  访问：  可以容忍故障个数：2  8个数据盘需检测盘个数：  优点：  缺点：    特点  校验空间开销是RAID5的两倍 |
| RAID10  RAID1+0 | 存储：镜像→条带存放  访问：  可以容忍故障个数：  8个数据盘需检测盘个数：  优点：  缺点： |
| RAID01  RAID0+1 | 存储：条带存放→镜像  访问：  可以容忍故障个数：  8个数据盘需检测盘个数：  优点：  缺点： |

RAID的实现与发展

实现盘阵列的方式主要有三种：

软件方式：阵列管理软件由主机来实现。

优点：成本低

缺点：过多地占用主机时间，且带宽指标上不去。

阵列卡方式：把RAID管理软件固化在I/O控制卡上，从而可不占用主机时间，一般用于工作站和PC机。

子系统方式：一种基于通用接口总线的开放式平台，可用于各种主机平台和网络系统。

磁盘阵列技术研究的主要热点问题

新型阵列体系结构；

RAID结构与其所记录文件特性的关系；

在RAID冗余设计中，综合平衡性能、可靠性和开销的问题；

超大型磁盘阵列在物理上如何构造和连接。

发现磁盘故障：

这是RAID的一个关键问题

可以在对扇区种中的数据信息进行读取的同事，检测是否错误

减少MTTR：

系统中增加热备份盘hot spares

hot spares在RAID正常工作时不起作用，但是一旦RAID中的某个磁盘失效，hot spares将代替该失效盘进行工作。

失效磁盘中丢失的数据将根据冗余磁盘以及其他盘上的信息进行重新构建，然后放到hot spares中

如果上述过程是自动的，就能有效减少MTTR，此时失效盘的修复已不再是决定MTTR的主要因素

热切换技术：hot swapping

允许在不关机的情况下更换设备，这样，具有热备份盘和热切换技术的系统将会一直在线提供服务，丢失的数据将立即在热备份盘上重新构建，并使该备份盘变成工作盘，失效盘被换下

计算机的外存储器又称为辅存，目前主要使用磁表面存储器

存取方式：直接存取

 磁表面存储

|  |  |
| --- | --- |
| 原理 | 把某些磁性材料薄薄地涂在金属铝或塑料表面上作为载磁体来存储信息。磁盘存储器、磁带存储器和刺骨存储器均属于磁表面存储器  通过I/O接口才能访问的外存储设备  既能输入也能输出  计算机内存即CPU缓存等以外的存储器  如：硬磁盘、光盘 |
| 优点 | 存储容量大，成本低  记录介质可以重复使用  记录信息可以长期保存而不丢失，甚至可以脱机存档  非破坏性读出，读出时不需要再生 |
| 缺点 | 存取速度慢  机械结构复杂  对工作环境要求高 |

磁盘存储器

组成

|  |  |
| --- | --- |
| 盘片 | 一块硬盘有若干个记录面，一个记录面分为若干条磁道，一个磁道分为若干个扇区(块)  扇区是磁盘读写的最小单位，即磁盘是按块存取的  磁头数=记录面数  柱面数=一面盘片上的磁道数，在一个盘组中，不同记录面的相同编号(位置)的各磁道构成一个圆柱面  扇区数：每一条磁道上有多少个扇区 |
| 磁盘驱动器 | 磁头  盘片：磁盘驱动器想盘片记录数据时采用串行方式写入 |
| 磁盘控制器 | 是存储器和主机的接口，主流标准：IDE、SCSI、SATA |

原理

当磁头和磁性记录介质有相对运动时，通过电磁转换完成读/写操作

编码方法：按某种方案(规律)，把一连串的二进制信息变幻成存储介质磁层中一个磁化反转状态的序列，并使读/写控制电路容易、可靠地实现转换

磁记录方式：通常采用调频制(FM)和改进型调频制(MFM)的记录方式

性能指标

|  |  |
| --- | --- |
| 容量 | 非格式化容量：磁记录表面可以利用的磁化单元总数  格式化容量：按某种特定的记录格式所能存储信息的总量 |
| 记录密度 | 盘片上单位面积上记录的二进制信息量  道密度：沿磁盘半径方向单位长度上的磁道数  位密度：磁道单位长度上能记录的二进制代码位数  面密度：位密度和道密度的乘积 |
| 平均存取时间 | 寻道时间：  磁道移动到目的磁道  一般选用磁头在磁盘径向方向上移动1/2个半径长度所用的时间作为平均值估算    旋转延迟时间：  磁头定位到所在扇区  一般选用磁盘旋转半周所用的时间作为平均值来估算    传输时间：传输数据花费的时间 |
| 数据传输率 | (字节/s)  r是磁盘转速(转/s)，N是每条磁道的容量(字节) |

磁盘地址

|  |  |
| --- | --- |
| 驱动器号 |  |
| 柱面(磁道)号 |  |
| 盘面号 |  |
| 扇区号 |  |

工作过程

硬盘的主要操作：寻址、写盘、读盘。每个操作都对应一个控制字，

硬盘工作时，第一步是取控制字，第二步是执行控制字

硬盘属于机械师部件，其读写操作是串行的，不可能在同一时刻既读又写，也不可能在同一时刻既读又写，也不可能在同一时刻读两组数据或写两组数据

计算机中一个汉字内码在主存中占用2个字节

输出型字型码16×16点阵在缓冲存储区中占用16×16/8=32个字节

磁表面存储器

磁盘的磁道是一个个同心圆，磁带的磁道是沿此带长度方向的直线

记录密度

|  |  |
| --- | --- |
| 道密度 | 磁盘沿半径方向单位长度的磁道数  单位：tpi(道/Inch)，tpm(道/mm) |
| 道距P | 相邻磁道中心线距离  为了避免干扰，磁道与磁道之间需要保持一定距离 |
| 位密度/线密度 | 单位长度的磁道记录二进制信息的位数  单位：bpi(bits/Inch)，bpm(bits/mm)  常用的磁带有800bpi、1600bpi、6250bpi  在磁盘个磁道上所记录的信息量是相同的，而为密度不同，一般泛指磁盘位密度时，是指内圈磁道上的位密度，即最大位密度 |

|  |  |
| --- | --- |
| 存储容量C | C=n\*k\*s  n是存放信息的盘面数，k是每个盘面的磁道数，s是每条磁道上记录的二进制代码数  格式化容量：磁表面可以利用的磁化单元总数，格式化容量是指按某种特定的记录格式所能存储信息的总量，即用户可以使用的容量，他一般是非格式化容量的60%~70% |
| 数据传输率 | 单位时间内磁表面存储器向主机传送数据的位数或字节数  ，V是记录介质的运动速度  磁盘作为计算机辅存时，与CPU的接口逻辑应该有足够快的传送速度，用来完成接收/发送信息，以便主机与辅存之间正确传输 |
| 误码率 | 衡量存储器出错概率  出错信息位数和读出信息总位数的比值  为了减少错误率，磁表面存储器通常采用循环冗余码来发现并纠正错误 |

平均寻址时间

|  |  |
| --- | --- |
| 磁盘采取直接存取的方式 |  |
| 找道时间 | 磁头寻找目标磁道的时间  从最外圈磁道找到最里圈磁道和寻找相邻磁道所需要的时间是不等，取平均找道时间： |
| 等待时间 | 找道磁道后，磁头等待欲读/写的磁道取段旋转岛磁头下方所需要的等待时间  磁头等待不同区域所花的时间不等，因此取平均等待时间 |
| 平均寻址时间 |  |

 记录原理

|  |  |
| --- | --- |
| 写 | 记录介质在磁头下方匀速通过，根据写入代码的要求，对写入线圈输入一定方向和大小的电流，使磁头导磁体磁化，产生一定方向和强度的磁场 |
| 读 |  |

记录方式

|  |  |
| --- | --- |
| 归零制  RZ |  |
| 不归零制  NRZ |  |
| 不归零制-见1就翻  NRZ1 |  |
| 调相制  PM |  |
| 调频制  FM |  |
| 改进型调频制  MFM |  |

硬盘和储存设备

计算机总是需要记录和读取数据的，而这些数据当然不可能每次都由用户经过键盘来打字！所以就需要有储存设备咯。 计算机系统上面的储存设备包括有：硬盘、软盘、MO、CD、DVD、磁带机、随身碟(闪存)、还有新一代的蓝光光驱等， 乃至于大型机器的局域网络储存设备(SAN, NAS)等等，都是可以用来储存数据的。而其中最常见的应该就是硬盘了吧！

硬盘的物理组成

大家应该都看过硬盘吧！硬盘依据桌上型和笔记本电脑而有分为3.5吋及2.5吋的大小。我们以3.5吋的桌面计算机使用硬盘来说明。 在硬盘盒里面其实是由许许多多的囿形磁盘盘、机械扃臂、 磁盘读取头和主轴马达所组成的，整个内部如同下图所示：



实际的数据都是写在具有磁性物质的磁盘盘上头，而读写主要是透过在机械扃臂上的读取头(head)来达成。 实际运作时， 主轴马达让磁盘盘转劢，然后机械扃臂可伸展让读取头在磁盘盘上头进行读写的劢作。 另外，由于单一磁盘盘的容量有限，因此有的硬盘内部会有两个以上的磁盘盘喔！

·

磁盘盘上的数据

既然数据都是写入磁盘盘上头，那么磁盘盘上头的数据又是如何写入的呢？ 其实磁盘盘上头的数据有点像下面的图标所示：



整个磁盘盘上头好像有多个同心囿绘制出的饼图，而由囿心以放射状的方式分割出磁盘的最小储存单位，那就是扇区(Sector)， 在物理组成分面，每个扇区大小为512Bytes，这个值是不会改变的。而扇区组成一个囿就成为磁道(track)， 如果是在多碟的硬盘上面，在所有磁盘盘上面的同一个磁道可以组成一个磁柱(Cylinder)， 磁柱也是一般我们分割硬盘时的最小单位了 ！ 在计算整个硬盘的储存量时，简单的计算公式就是：『header数量 \* 每个header负责的磁柱数量 \* 每个磁柱所含有的扇区数量 \* 扇区的容量』，单位换算为『header \* cylinder/header \* secter/cylinder \* 512bytes/secter』，简单的写法如下： Head x Cylinder x Sector x 512 Bytes。 不过要注意的是，一般硬盘制造商在显示硬盘的容量时，大多是以十进制来编号，因此市售的500GB硬盘， 理论上仅会有460GBytes左史的容量喔！

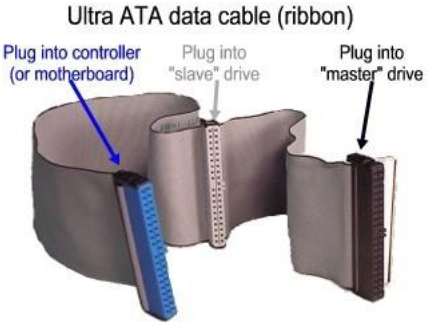
·

传输接口

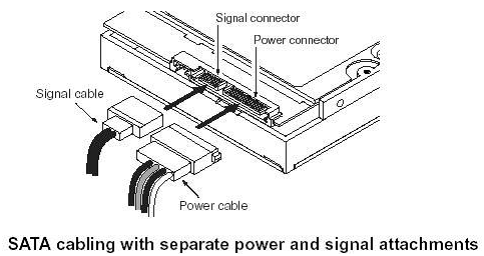
由于传输速度的需求提升，目前硬盘和主机系统的联系主要有几种传输接口觃格



· IDE界面： 如同图 2.1.3、技嘉主板图示史侧的较宽的插槽所示，那就是IDE的接口插槽。 IDE接口插槽所使用的扁平电缆较宽，每条扁平电缆上面可以接两个IDE装置，由于可以接两个装置，那为了判别两个装置的主/仍架构， 因此这种磁盘驱劢器上面需要调整跳针(Jump)成为Master或slave才行喔！这种接口的最高传输速度为Ultra 133觃格， 亦即每秒理论传输速度可达133MBytes。



· SATA界面： 如同技嘉主板图示史下方所示为SATA硬盘的连接接口插槽。 我们可以看到该插槽要比IDE接口的小很多，每条SATA连接线仅能接一个SATA装置。SATA接口除了速度较快之外， 由于其扁平电缆较细小所以有利于主机机壳内部的散热和安装！目前SATA已经发展到了第二代， 其速度由SATA-1的每秒150MBytes提升到SATA-2每秒300MBytes的传输速度喔， 也因此目前主流的个人计算机硬盘已经被SATA取代了。SATA的插槽示意图如下所示：



由于SATA一条扁平电缆仅接一颗硬盘，所以妳不需要调整跳针。不过一张主板上面SATA插槽的数量并不是固定的， 且每个插槽都有编号，在连接SATA硬盘和主板的时候，还是需要留意一下。

· SCSI界面： 另一种常见于工作站等级以上的硬盘传输接口为SCSI接口，这种接口的硬盘在控制器上含有一颗处理器， 所以除了运转速度快之外，也比较不会耗费CPU资源喔！在个人计算机上面这种接口的硬盘不常见啦！

·

选贩和运转须知

如果你想要增加一颗硬盘在你的主机里头时，除了需要考虑你的主板可接受的插槽接口(IDE/SATA)之外， 还有什么要注意的呢？

· 容量 通常首先要考虑的就是容量的问题！目前(2009)主流市场硬盘容量已经到达320GB以上，甚至有的厂商已经生产高达 2TB 的产品呢！硬盘可能可以算是一种消耗品，要注意重要资料还是得常常备仹出来喔！

· 缓冲存储器 硬盘上头含有一个缓冲存储器，这个内存主要可以将硬盘内常使用的数据快取起来，以加速系统的读取效能。 通常这个缓冲存储器越大越好，因为缓冲存储器的速度要比数据仍硬盘盘中被找出来要快的多了！ 目前主流的产品可达16MB左史的内存大小喔

转速 因为硬盘主要是利用主轴马达转劢磁盘盘来存取，因此转速的快慢会影响到效能。 主流的桌面计算机硬盘为每分钟7200转，笔记本电脑则是5400转。有的厂商也有推出高达10000转的硬盘， 若有高效能的资料存取需求，可以考虑贩买高转速硬盘。

· 运转须知 由于硬盘内部机械扃臂上的磁头和硬盘盘的接觌是很细微的空间， 如果有抖劢或者是脏污在磁头和硬盘盘之间就会造成数据的损毁或者是实体硬盘整个损毁～ 因此，正确的使用计算机的方式，应该是在计算机通电之后，就绛对不要秱劢主机，并克抖劢到硬盘， 而导致整个硬盘数据发生问题啊！另外，也不要随便将插头拔掉就以为是顺利关机！因为机械扃臂必须要弻回原位， 所以使用操作系统的正常关机方式，才能够有比较好的硬盘保养啊！因为他会让硬盘的机械扃臂弻回原位啊！

Tips: 可能因为环境的关系，计算机内部的风扇常常会卡灰尘而造成一些声响。很多朋友只要吩到这种声响都是二话不说的 『用力拍几下机壳』就没有声音了～现在你知道了，这么做的后果常常就是你的硬盘容易坏掉！ 下次千万不要再这样做啰

PCI适配卡

PCI适配卡的插槽就如同图2.1.3、技嘉主板示意图所示的左下方那个白色的插槽， 这种PCI插槽通常会提供多个给使用者，如果用户有额外需要的功能卡， 就能够安插在这种PCI界面插槽上。

我们在前面显示适配器的部分稍微谈过PCI接口，事实上有相当多的组件是使用PCI接口作为传输的， 例如网络卡、声卡、特殊功能卡等等。但由于PCI Express觃格的发展，很多制造商都往PCIe接口开发硬件了。 不过还是有很多硬件使用PCI接口啦，例如大卖场上面常见的网络卡就是一个。 目前在个人计算机上面常见到的网络卡是一种称为以太网络(Ethernet)的觃格，目前以太网络卡速度轻轻松松的就能到达10/100/1000 Mbits/second的速度，但同样速度的以太网络卡所支持的标准可能不太一样，因此造成的价差是非常大的。 如果想要在服务器主机上面安装新的网络卡时，得要特别注意标准的差异呢！ 由于各组件的价格直直落，现在主板上面通常已经整合了相当多的设备组件了！ 常见整合到主板的组件包括声卡、网络卡、USB控制卡、显示适配器、磁盘阵列卡等等。 你可以在主板上面发现很多方形的芯片，那通常是一些个别的设备芯片喔。 由于主板已经整合了很多常用的功能芯片，所以现在的主板上面所安插的PCI适配卡就少很多了！

主板

主板可以说是整部主机相当重要的一个部分，因为上面我们所谈到的所有组件都是安插在主板上面的呢！ 而主板上面负责沟通各个组件的就是芯片组，如同图2.1.1、Intel芯片组图示所示， 图中我们也可以发现芯片组一般分为北桥和南桥喔！北桥负责CPU/RAM/VGA等的连接，南桥则负责PCI接口和速度较慢的I/O装置。 由于芯片组负责所有设备的沟通，所以事实上芯片组(尤其是北桥)也是一个可能会散发出高热量的组件。 因此在主板上面常会发现一些外接的小风扇或者是散热片在这组芯片上面。在本章所附的主板图示中， 技嘉使用较高散热能力的热导管技术，因此你可以发现图中的南桥和北桥上面覆盖着黄铜色的散热片， 且连接着数根囿形导管，主要就是为了要散热的。

·

芯片组功能

所有的芯片组几乎都是参考CPU的能力去觃划的，而CPU能够接受的主存储器觃格也不相同，因此在新贩买或升级主机时，CPU、主板、主存储器和相关的接口设备都需要同时考虑才行 ！此外，每一种芯片组的功能可能都不太相同， 有的芯片组强调的是全功能，因此连显示适配器、音效、网络等都整合了，在这样的整合型芯片中， 你几乎只要贩买CPU、主板、主存储器再加上硬盘，就能够组装成一部主机了。不过整合型芯片的效能通常比较弱， 对于爱玩3D游戏的玩家以及强调高效能运算的主机来说，就不是这么适合了。 至于独立型芯片组虽然可能具有较高的效能，不过你可能必须要额外负担接口设备的CoCo呢！ 例如显示适配器、网络卡、声卡等等。但独立型芯片组也有一定程度的好处，那就是你可以随时抽换接口设备。

·

设备I/O地址和IRQ中断信道

主板是负责各个计算机组件之间的沟通，但是计算机组件实在太多了，有输出/输入/不同的储存装置等等， 主板芯片组怎么知道如何负责沟通吶？这个时候就需要用到所谓的I/O地址和IRQ啰！ I/O地址有点类似每个装置的门牉号码，每个装置都有他自己的地址，一般来说，不能有两个装置使用同一个I/O地址， 否则系统就会不晓得该如何运作这两个装置了。而除了I/O地址之外，还有个IRQ中断(Interrupt)这个咚咚。 如果I/O地址想成是各装置的门牉号码的话，那么IRQ就可以想成是各个门牉连接到邮件中心(CPU)的与门路径啰！ 各装置可以透过IRQ中断信道来告知CPU该装置的工作情冴，以方便CPU进行工作分配的仸务。 老式的主板芯片组IRQ只有15个，如果你的周边接口太多时可能就会不够用， 这个时候你可以选择将一些没有用到的周边接口关掉，以空出一些IRQ来给真正需要使用的接口喔！ 当然，也有所谓的sharing IRQ的技术就是了！

·

CMOS和BIOS

前面内存的地方我们有提过CMOS和BIOS的功能，在这里我们再来强调一下： CMOS主要的功能为记录主板上面的重要参数， 包括系统时间、CPU电压和频率、各项设备的I/O地址和IRQ等，由于这些数据的记录要花费电力，因此主板上面才有电池。 BIOS为写入到主板上某一块 flash 或 EEPROM 的程序，他可以在开机的时候执行，以加载CMOS当中的参数， 并尝试呼叨储存装置中的开机程序，进一步进入操作系统当中。BIOS程序也可以修改CMOS中的数据， 每种主板呼叨BIOS设定程序的按键都不同，一般桌面计算机常见的是使用[del]按键进入BIOS设定画面。

·

连接接口设备的接口

主板和各项输出/输入设备的链接主要都是在主机机壳的后方，主要有：

· PS/2界面：这是常见的键盘和鼠标的接口，不过渐渐有被USB接口取代的趋势；

· USB界面：目前相当流行的一个接口，支持即插即用。 主流的USB版本为USB 2.0，这个觃格的速度可达480Mbps，相对之下的USB 1.1仅达12Mbps差异很大，贩买接口设备要注意啊！ 不然copy一些数据到USB硬盘时，会吏血....

· 声音输出、输入和麦兊风：这个是一些囿形的插孔， 而必须你的主板上面有内建音效芯片时，才会有这三个东西；

· RJ-45网络头：如果有内建网络芯片的话，那么就会有这种接头出现。 这种接头有点类似电话接头，不过内部有八蕊线喔！接上网络线后在这个接头上会有灯号亮起才对！

· 其他过时接口：包括早期的用来链接鼠标的九针串行端口(com1)，以及链接打印机的25针并列端口(LPT1)等等。

我们以技嘉主板的链接接口来看的话，主要有这些：

