作者：蔡于清
www.another-prj.com

在上一篇文档中我向大家介绍 MMU 的工作原理和对 s3c2410 MMU 部分操作进行了讲解。我们知道 MMU 存在的原因是为了支持虚拟存储技术，但不知道你发现了没有，虚拟存储技术的使用会降低整个系统的效率，因为与传统的存储技术相比，虚拟存储技术对内存的访问操作多了一步，就是对地址进行查表（查找映射关系），必须先从虚拟地址中分解出页号和页内偏移，根据页号对描述符进行索引（这就是一个查表过程）得到物理空间的首地址，这样做的代价是巨大的（其实这也正是时间效率与空间效率之间矛盾的一个体现），对某些嵌入式系统来说这简直就是恶梦。那么在引入了虚拟存储技术之后有没有方法在时间效率与空间效率这个矛盾之间取得一个平衡点呢？答案是有，我们可以通过一种技术从最大限度上降低这两者的矛盾，这种技术是 Caches(缓存)。也是我们本文要介绍的。

以下内容转载自中计报

Cache 的工作原理

Cache 的工作原理是基于程序访问的局部性。

对大量典型程序运行情况的分析结果表明，在一个较短的时间间隔内，由程序产生的地址往往集中在存储器逻辑地址空间的很小范围内。指令地址的分布本来就是连续的，再加上循环程序段和子程序段要重复执行多次。因此，对这些地址的访问就自然地具有时间上集中分布的倾向。

数据分布的这种集中倾向不如指令明显，但对数组的存储和访问以及工作单元的选择都可以使存储器地址相对集中。这种对局部范围的存储器地址频繁访问，而对此范围以外的地址则访问甚少的现象，就称为程序访问的局部性。

根据程序的局部性原理，可以在主存和 CPU 通用寄存器之间设置一个高速的容量相对较小的存储器，把正在执行的指令地址附近的一部分指令或数据从主存调入这个存储器，供 CPU 在一段时间内使用。这对提高程序的运行速度有很大的作用。这个介于主存和 CPU 之间的高速小容量存储器称作高速缓冲存储器(Cache)。

系统正是依据此原理，不断地将与当前指令集相关联的一个不太大的后继指令集从内存读到 Cache，然后再与 CPU 高速传送，从而达到速度匹配。

CPU 对存储器进行数据请求时，通常先访问 Cache。由于局部性原理不能保证所请求的数据百分之百地在 Cache 中，这里便存在一个命中率。即 CPU 在任一时刻从 Cache 中可靠获取数据的几率。

命中率越高，正确获取数据的可靠性就越大。一般来说，Cache 的存储容量比主存的容量小得多，但不能太小，太小会使命中率太低；也没有必要过大，过大不仅会增加成本，而且当容量超过一定值后，命中率随容量的增加将不会有明显地增长。

只要 Cache 的空间与主存空间在一定范围内保持适当比例的映射关系，Cache 的命中率还是相当高的。

一般规定 Cache 与内存的空间比为 4:1000，即 128kB Cache 可映射 32MB 内存；256kB Cache 可映射 64MB 内存。在这种情况下，命中率都在 90%以上。至于没有命中的数据，CPU 只好直接从内存获取。获取的同时，也把它拷进 Cache，以备下次访问。

Cache 的基本结构

Cache 通常由相联存储器实现。相联存储器的每一个存储块都具有额外的存储信息，称为标签(Tag)。当访问相联存储器时，将地址和每一个标签同时进行比较，从而对标签相同的存储块进行访问。Cache 的 3 种基本结构如下：

全相联 Cache

在全相联 Cache 中，存储的块与块之间，以及存储顺序或保存的存储器地址之间没有直接的关系。程序可以访问很多的子程序、堆栈和段，而它们是位于主存储器的不同部位上。
因此，Cache 保存着很多互不相关的数据块，Cache 必须对每个块和块自身的地址加以存储。当请求数据时，Cache 控制器要把请求地址同所有地址加以比较，进行确认。
这种 Cache 结构的主要优点是，它能够在给定的时间内去存储主存器中的不同的块，命中率高；缺点是每一次请求数据同 Cache 中的地址进行比较需要相当的时间，速度较慢。
直接映像 Cache
直接映像 Cache 不同于全相联 Cache，地址仅需比较一次。
在直接映像 Cache 中，由于每个主存储器的块在 Cache 中仅存在一个位置，因而把地址的比较次数减少为一次。其做法是，为 Cache 中的每个块位置分配一个索引字段，用 Tag 字段区分存放在 Cache 位置上的不同的块。
单路直接映像把主存储器分成若干页，主存储器的每一页与 Cache 存储器的大小相同，匹配的主存储器的偏移量可以直接映像为 Cache 偏移量。Cache 的 Tag 存储器(偏移量)保存着主存储器的页地址(页号)。
以上可以看出，直接映像 Cache 优于全相联 Cache，能进行快速查找，其缺点是当主存储器的组之间做频繁调用时，Cache 控制器必须做多次转换。
组相联 Cache
组相联 Cache 是介于全相联 Cache 和直接映像 Cache 之间的一种结构。这种类型的 Cache 使用了几组直接映像的块，对于某一个给定的索引号，可以允许有几个块位置，因而可以增加命中率和系统效率。
Cache 与 DRAM 存取的一致性
在 CPU 与主存之间增加了 Cache 之后，便存在数据在 CPU 和 Cache 及主存之间如何存取的问题。读写各有 2 种方式。
贯穿读出式(Look Through)
该方式将 Cache 隔在 CPU 与主存之间，CPU 对主存的所有数据请求都首先送到 Cache，由 Cache 自行在自身查找。如果命中，则切断 CPU 对主存的请求，并将数据送出；不命中，则将数据请求传给主存。
该方法的优点是降低了 CPU 对主存的请求次数，缺点是延迟了 CPU 对主存的访问时间。
旁路读出式(Look Aside)
在这种方式中，CPU 发出数据请求时，并不是单通道地穿过 Cache，而是向 Cache 和主存同时发出请求。由于 Cache 速度更快，如果命中，则 Cache 在将数据回送给 CPU 的同时，还来得及中断 CPU 对主存的请求；不命中，则 Cache 不做任何动作，由 CPU 直接访问主存。
它的优点是没有时间延迟，缺点是每次 CPU 对主存的访问都存在，这样，就占用了一部分总线时间。
写穿式(Write Through)
任一从 CPU 发出的写信号送到 Cache 的同时，也写入主存，以保证主存的数据能同步地更新。
它的优点是操作简单，但由于主存的慢速，降低了系统的写速度并占用了总线的时间。
回写式(Copy Back)
为了克服贯穿式中每次数据写入时都要访问主存，从而导致系统写速度降低并占用总线时间的弊病，尽量减少对主存的访问次数，又有了回写式。
它是这样工作的：数据一般只写到 Cache，这样有可能出现 Cache 中的数据得到更新而主存中的数据不变(数据陈旧)的情况。但此时可在 Cache 中设一标志地址及数据陈旧的信息，只有当 Cache 中的数据被再次更改时，才将原更新的数据写入主存相应的单元中，然后再接受再次更新的数据。这样保证了 Cache 和主存中的数据不致产生冲突。

…

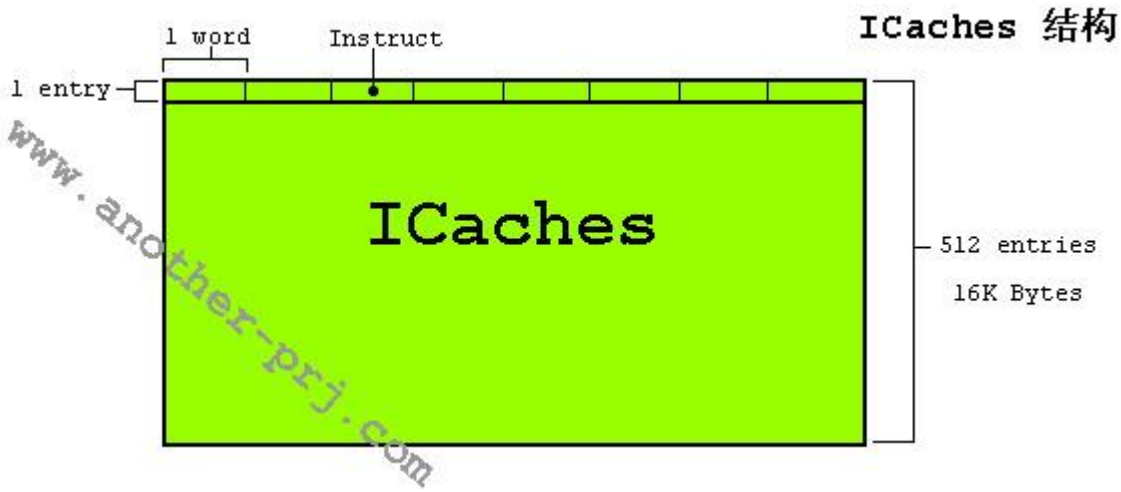
…..

你可以通过 http://www.chinaunix.net/jh/45/180390.html 阅读完全文

s3c2410 内置了**指令缓存（ICaches）,数据缓存（DCaches）,写缓存（write buffer），物理地址标志读写区（Physical Address TAG RAM),**CPU 将通过它们来提高内存访问效率。

我们先讨论**指令缓存（ICaches）。**

ICaches 使用的是虚拟地址，它的大小是 16KB,它被分成 512 行(entry)，每行 8 个字（8 words,32Bits）。



当系统上电或重起（Reset）的时候，ICaches 功能是被关闭的，我们必须往 Icr bit 置 1 去开启它，Icr bit 在 CP15 协处理器中控制寄存器 1 的第 12 位（关闭 ICaches 功能则是往该位置 0）。ICaches 功能一般是在 MMU 开启之后被使用的（为了降低 MMU 查表带来的开销）,但有一点需要注意，并不是说 MMU 被开启了 ICaches 才会被开启，正如本段刚开始讲的，ICaches 的开启与关闭是由 Icr bit 所决定的，无论 MMU 是否被开启，只要 Icr bit 被置 1 了，ICaches 就会发挥它的作用。

大家是否还记得 discriptor（描述符）中有一个 C bit 我们称之为 **Ctt**,它是指明该描述符描述的内存区域内的内容（可以是指令也可以是数据）是否可以被 Cache，若 Ctt=1,则允许 Cache,否则不允许被 Cache。于是 CPU 读取指令出现了下面这些情况：

1．如果 CPU 从 Caches 中读取到所要的一条指令（**cache hit**）且这条指令所在的内存区域是 Cacheble 的（该区域

所属描述符中 Ctt=1）,则 CPU 执行这条指令并从 Caches 中返回（不需要从内存中读取）。

2．若 CPU 从 Caches 中读取不到所要的指令（**cache miss**）而这条指令所在的内存区域是 Cacheble 的（同第 1 点），则 CPU 将从内存中

读取这条指令，同时，一个称为**“8-word linefill”的动作将发生**，这个动作是把该指令所处区域的 8 个 word 写进

ICaches 的某个 entry 中，这个 entry 必须是没有被锁定的（对锁定这个操作感兴趣的朋友可以找相关的资料进行了解）

3．若 CPU 从 Caches 中读取不到所要的指令（cache miss）而这条指令所在的内存区域是 UnCacheble 的（该区域所属描

述符中 Ctt=0），则 CPU 将从内存读取这条指令并执行后返回（不发生 linefill）

通过以上的说明，我们可以了解到 CPU 是怎么通过 ICaches 执行指令的。你可能会有这个疑问，ICaches 总共只有 512 个条目（entry），当 512 个条目都被填充完之后，CPU 要把新读取近来的指令放到哪个条目上呢？答案是 CPU 会把新读取近来的 8 个 word 从 512 个条目中选择一个对其进行写入，那 CPU 是怎么选出一个条目来的呢？这就关系到 ICaches 的**替换法则（replacemnet algorithm）**了。ICaches 的 replacemnet algorithm 有两种，一种是 **Random 模式**另一种 **Round-Robin 模式**，我们可以通过 CP15 协处理器中寄存器 1 的 RR bit 对其进行指定（0 = Random replacement 1 = Round robin replacement），如果有需要你还可以进行指令锁定（INSTRUCTION CACHE LOCKDOWN）。
关于替换法则和指令锁定我就不做详细的讲解，感兴趣的朋友可以找相关的资料进行了解。

接下来我们谈**数据缓存（DCaches）**和 **写缓存（write buffer）**
DCaches 使用的是虚拟地址，它的大小是 16KB,它被分成 512 行（entry），每行 8 个字（8 words,32Bits）。每行有两个修改标志位（dirty bits），第一个标志位标识前 4 个字，第二个标志位标识后 4 个字，同时每行中还有一个 TAG 地址（标签地址）和一个 valid bit。
与 ICaches 一样，系统上电或重起（Reset）的时候，DCaches 功能是被关闭的，我们必须往 Ccr bit 置 1 去开启它，Ccr bit 在 CP15 协处理器中控制寄存器 1 的第 2 位（关闭 DCaches 功能则是往该位置 0）。与 ICaches 不同，DCaches 功能是必须在 MMU 开启之后才能被使用的。
我们现在讨论的都是 **DCaches**,你可能会问那 **Write Buffer** 呢？他和 **DCaches** 区别是什么呢？其实 **DCaches** 和 **Write Buffer** 两者间的操作有着非常紧密的联系，很抱歉，到目前为止我无法说出他们之间有什么根本上的区别（-_-!!!），但我能告诉你什么时候使用的是 DCaches,什么时候使用的是 Write Buffer.系统可以通过 Ccr bit 对 Dcaches 的功能进行开启与关闭的设定，但是在 s3c2410 中却没有确定的某个 bit 可以来开启或关闭 Write Buffer…你可能有点懵…我们还是来看一张表吧，这张表说明了 DCaches,Write Buffer 和 CCr,Ctt (descriptor 中的 C bit),Btt(descriptor 中的 B bit)之间的关系，其中“Ctt and Ccr”一项里面的值是 Ctt 与 Ccr 进行逻辑与之后的值（Ctt&&Ccr）.

Table 4-1. Data Cache and Write Buffer Configuration

Ctt and Ccr	Btt	Data cache, write buffer and memory access behavior
0 (1)	0	Non-cached, non-buffered (NCNB) Reads and writes are not cached and always perform accesses on the ASB and may be externally aborted. Writes are not buffered. The CPU halts until the write is completed on the ASB. Cache hits should never occur. (2)
0	1	Non-cached buffered (NCB) Reads and writes are not cached, and always perform accesses on the ASB. Cache hits should never occur. Writes are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Reads may be externally aborted. Writes can not be externally aborted.
1	0	Cached, write-through mode (WT) Reads which hit in the cache will read the data from the cache and do not perform an access on the ASB. Reads which miss in the cache cause a linefill. All writes are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Writes which hit in the cache update the cache. Writes cannot be externally aborted.
1	1	Cached, write-back mode (WB) Reads which hit in the cache will read the data from the cache and do not perform an ASB access. Reads which miss in the cache cause a linefill. Writes which miss in the cache are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Writes which hit in the cache update the cache and mark the appropriate half of the cache line as dirty, and do not cause an ASB access. Cache write-backs are buffered. Writes (Cache write-misses and cache write-backs) cannot be externally aborted.

从上面的表格中我们可以清楚的知道系统什么时候使用的是 DCaches,什么时候使用的是 Write Buffer，我们也可以看到 DCaches 的写回方式是怎么决定的（write-back or write-througth）。

在这里我要对 Ctt and Ccr=0 进行说明，能够使 Ctt and Ccr=0 的共有三种情况，分别是

Ctt =0， Ccr=0

Ctt =1， Ccr=0

Ctt =0， Ccr=1

我们分别对其进行说明。

情况 1（Ctt =0， Ccr=0）：这种情况下 CPU 的 DCaches 功能是关闭的（Ccr=0），所以 CPU 存取数据的时候不会从 DCaches 里进行数据地查询，CPU 直接去内存存取数据。

情况 2（Ctt =1， Ccr=0）：与情况 1 相同。

情况 3（Ctt =0， Ccr=1）：这种情况下 DCaches 功能是开启的，CPU 读取数据的时候会先从 DCaches 里进行数据地查询，若 DCaches 中没有合适的数据，则 CPU 会去内存进行读取，但此时由于 Ctt =0（Ctt 是 descriptor 中的 C bit,该 bit 决定该 descriptor 所描述的内存区域是否可以被 Cache）,所以 CPU 不会把读取到的数据 Cache 到 DCaches(不发生 linefill).

到此为止我们用两句话总结一下 DCaches 与 Write Buffer 的开启和使用：

1．DCaches 与 Write Buffer 的开启由 Ccr 决定。

2．DCaches 与 Write Buffer 的使用规则由 Ctt 和 Btt 决定。

DCaches 与 **ICaches** 有一个最大的不同，ICaches 存放的是指令，DCaches 存放的是数据。程序在运行期间指令的内容是不会改变的，所以 ICaches 中指令所对应的内存空间中的内容不需要更新。但是数据是随着程序的运行而改变的，所以 DCaches 中数据必须被及时的更新到内存（这也是为什么 ICaches 没有写回操作而 DCaches 提供了写回操作的原因）。提到写回操作，就不得不提到 **PA TAG 地址（物理标签地址）**这个固件，它也是整个 Caches 模块的重要组成部分。

简单说 PA TAG 地址（物理标签地址）的功能是指明了写回操作必须把 DCaches 中待写回内容写到物理内存的哪个位置。不知道你还记不记得，DCaches 中每个 entry 中都有一个 PA TAG 地址（物理标签地址）,当一个 linefill 发生时，被 Cache 的内容被写进了 DCaches,同时被 Cache 的物理地址则被写入了 PA TAG 地址（物理标签地址）。除了 TAG 地址（标签地址），还有两个称为 **dirty bit（修改标志位）**的位出现在 DCaches 的每一个 entry 中，它们指明了当前 entry 中的数据是否已经发生了改变（发生改变简称为变“脏”，所以叫 dirty bit,老外取名称可真有意思-_-!!!）。如果某个 entry 中的 dirty bit 置位了，说明该 entry 已经变脏，于是一个写回操作将被执行，写回操作的目的地址则是由 PA TAG 地址（物理标签地址）索引到的物理地址。