The MIPS Cache Architecture

CurrentVersion: 0.4
Date: 2008-05-30
Author: comcat <jiankemeng@gmail.com>

版本历史

版本状态	作 者	参与者	起止日期	备注
0.1	comcat		08-03-21	初始化
0.2	comcat		08-04-02	完成 ch1
0.4	comcat		08-05-30	完成草稿

目　录

1. Cache 基本思想

CPU 访问寄存器中的数据要比访问内存中的数据快几个数量级，这个在大量数据运算时，CPU 的处理速度会受限于访存速度，为了平滑他们之间的速度差异，设计者根据程序局部性原理引入了Cache，CPU 访问它的速度介于寄存器与内存之间，起到了一个平滑作用。实现 Cache 的花费介于寄存器与内存之间，这又是一个解决性能与价格矛盾的折衷。

引入 Cache 的理论基础是程序局部性原理，包括时间局部性和空间局部性。即最近被CPU访问的数据，短期内 CPU 还要访问（时间）；被 CPU 访问的数据附近的数据，CPU 短期内还要访问（空间）。因此如果将刚刚访问过的数据缓存在 Cache 中，那下次访问时，可以直接从Cache中取，其速度可以得到数量级的提高。

Cache 与内存之间交换数据以“数据行”(Line)为单位，大小一般为 16 字节，32 字节或者 64 字节：

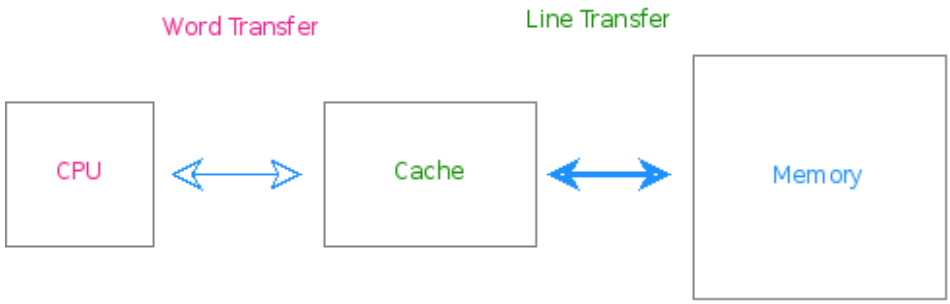


Figure 1: Cache and Memory

CPU 与数据 Cache 之间的数据交换则以字为单位（为获得较大的指令吞吐率，与指令 Cache 之间的数据交换则以数个字为单位，较常见的是128bit）

CPU 要访问的数据在Cache中有缓存则是为“命中” (Hit)，反之则为“缺失” (Miss)

Cache 以行大小(Line size)为单位分成若干个行，内存逻辑上亦可如此，其皆可按地址由低向高，依次编号。因此 Cache 与内存就会有个映射问题，如：第四个内存行中的数据被CPU访问时，该行将缓存于 Cache 中的哪一行？

1.1 相联方式

根据 Cache 相联方式的不同，Cache 有 3 类：直接相联，全相联，组相联。

直接相联 Cache 的映射规则为：

Cache行号 = 内存行号 % Cache总行数

假设某CPU之Cache共有2行，则Cache 与内存的映射关系为：

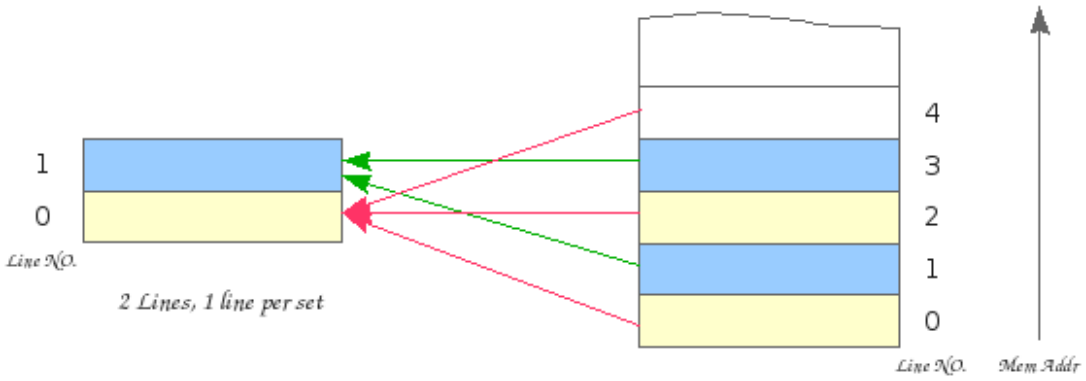


Figure 2: 直接相联 Cache 与 Memory 之间的映射

在直接相联 Cache 中，内存行都是映射到 Cache 的固定行（直接用地址高位索引，无需匹配）。这个在实际应用中，它的行冲突率会比较高，命中率会比较低。比如某段时间内 CPU 要不断的交叉访问2个映射到相同 Cache 行的数据行，那么尽管 Cache 尚有空间，可是 CPU 却不得不不停的来回填充Cache。它的优点在于结构简单、成本低。

相较于直接相联，全相联 Cache 是另一个极端，其允许任意内存行映射到 Cache 的任意行（不需索引，直接匹配）。缓存数据时，有空占空，当所有行被占用时，新入的行则需要替换掉一行，这个需要一个替换算法，常用的有 LRU(Least Recently Used)、Random、FIFO(First In First Out) 和 LFU(Least Frequently Used)。
全相联 Cache 之所有行集合成一个组 (set)，当 CPU 查找 Cache 时，CPU用地址同时与 Cache 中所有行的 Tag 相比较（匹配）。因为任意的内存行可以缓存于 Cache 的任意行，

因此其能充分利用 Cache 的存储空间，行冲突率会比较低，命中率较高，但实现起来硬件电路复杂，价格昂贵，一般用在对性能要求苛刻的场合，如 TLB。

组相联 Cache 则是直接相联与全相联的一个折衷，兼顾性能与价格。对 K 路组相联，首先对Cache 分组，每组含 K 个行，映射时先索引组，其规则为：

> Cache 组号 = 内存行号 % Cache总组数

尔后，内存行可以映射到该组内的任意行上。缓存数据时，有空占空，如组内所有行被占用，则使用替换算法(LRU, Random, FIFO, LFU)替换掉一行。

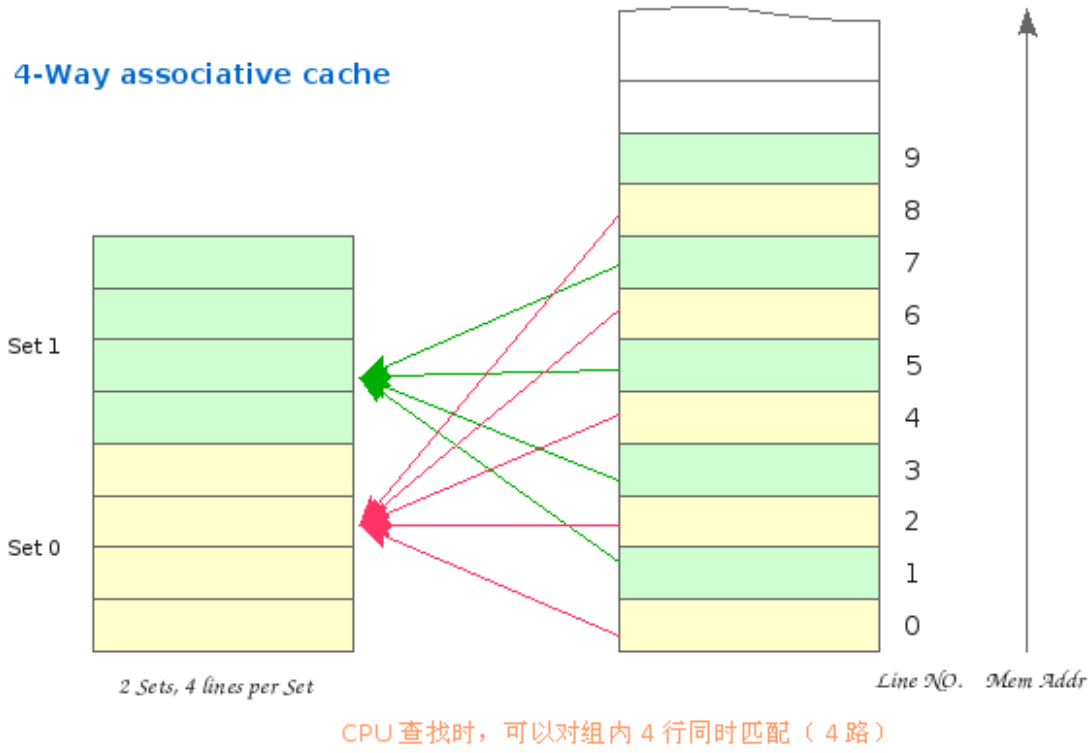


Figure 2: 4 路组相联 Cache 与 Memory 之间的映射

对于4路组相联Cache，其每组就有4行。可以编号为 W0 ~ W3，所有组(Set)的W0 的集合可以称为第 0 路，其大小为 Cache_size/ways。其他 3 路以此类推。

1.2 指令数据分离

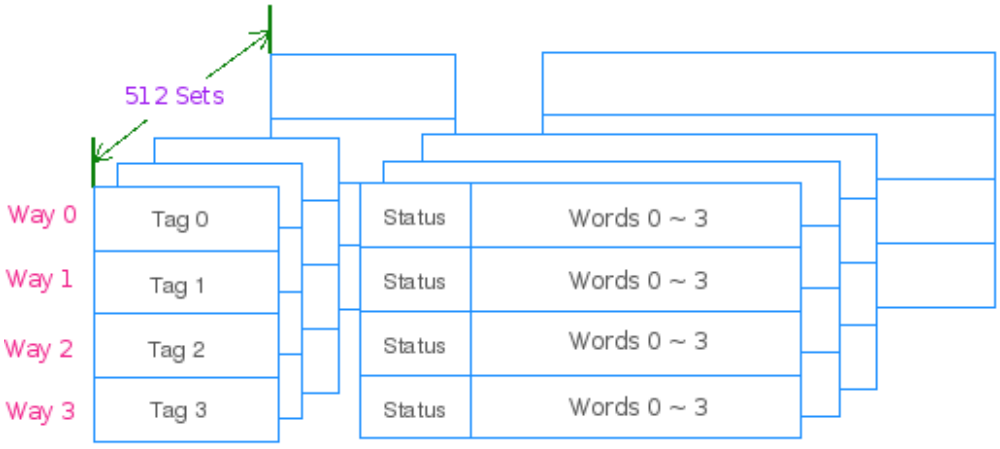
现代处理器的 L1 Cache 设计时基本采用哈佛结构，即将指令和数据分离，常称为 I-Cache、D-Cache，其各自有独立的读写端口（I-Cache 只读，不需写端口）。其对性能的提升是很明显的，指令读取和操作数据的读写可以在同一时钟周期内进行，且已缓存的指令，不会因 Cache 的刷新或行替换被清理掉，增加流水线取指段的延迟。
其缺点在于，当位于内存的指令块被更新时，I-Cache 中的数据不会被更新。当然在实际应用中，动态的修改指令段的情况极少出现，如果有，则更改指令后，软件要负责 flush I-Cache。

1.3 多级 **Cache**

高性能的处理器往往带有 L2 Cache，有的甚至带有 L3 Cache，其读写延迟依次增加，实现的成本依次降低。现代系统采用从 Register ---> L1 Cache ---> L2 Cache ---> L3 Cache ---> Memory ---> Mass storage 的层次结构，是为解决性能与价格矛盾所采用的实际可行的折中设计。

2. Cache 结构

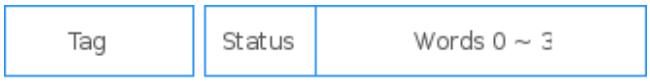
K 路组相联的 Cache 是目前使用最为广泛的 Cache 类型。如某 CPU 使用32KB大小、4路组相联、行大小为16 Bytes 的 L1 Cache，则该 Cache 每组有4行，共有512组。其结构如下图所示：



32KB, 4 Way, 16 Bytes line size Cache structure

其每组大小为 16B * 4 = 64B，每路大小为 32KB/4 = 8KB

该 Cache 的每行结构为：



Cache Line structure (16Bytes Line Size)

其由 Tag 域、Status 域和数据域组成，

Tag 域存放该行数据对应地址的高位。CPU在索引到该组 (set) 后，用相应地址与组内所有行的 Tag 相比较，以之区分具体的行。

数据域能容纳的字节数是为行大小 (Line Size)，其为 Cache 与内存之间数据交换的单位。

Status 域则为一些控制位信息（如Valid, Lock 以及Parity check 位等等），不同的 Cache 类型，不同的 Cache 实现 Status 域稍有不同，具体的可以参考相应 CPU 手册。

对于 K 路组相联的 Cache 因为当组内所有行被占用时，需要指定将要被替换的行，因此每组亦有一个用于路选 (Way-Select) 的数据域。不同的替换算法对该数据域的定义和使用亦不同，例如 PowerPC E500 内核的 Cache 实现使用PLRU (Pseudo LRU) 算法，8路的 Cache 路选域需要7位；而对于MIPS 4KE 的 Cache 实现，其使用LRU算法，4路的 Cache 路选域需要6位。关于这些算法的细节可以参考相关的资料。

某些 MIPS Cache 的实现（如MIPS 4KE, 24KE, 34K etc.）提供 Software Cache Testing 模式可以让程序员读出这个路选域的值，进而判断出当前 Cache 是否工作正常。这个特性在调试与Cache相关问题时非常有用。

所有路选数据域的集合是为 Way-Select Array.

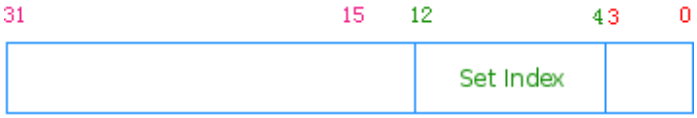
对于 I-Cache，因为其只读的特性，一致性的维护要容易的多，因此管理需要的控制域要较 D-Cache 少。如 4KE 的 D-Cache 实现，每组除了路选数据域外，还有 K 位的 Dirty 域，而 I-Cache 则不需要。具体的可以参考相应的CPU文档。

3. Cache 工作方式

3.1 概述

K 路组相联 Cache 的通常工作方式是：先组索引，然后用物理地址 (PA) 或虚拟地址 (VA) 同时与组内K 行之 Tag 域同时匹配，有匹配则为命中 (Hit)，尔后将命中的行中相应的字传给CPU。

以上述 Cache 为例，正常工作时其对地址的划分如下所示：

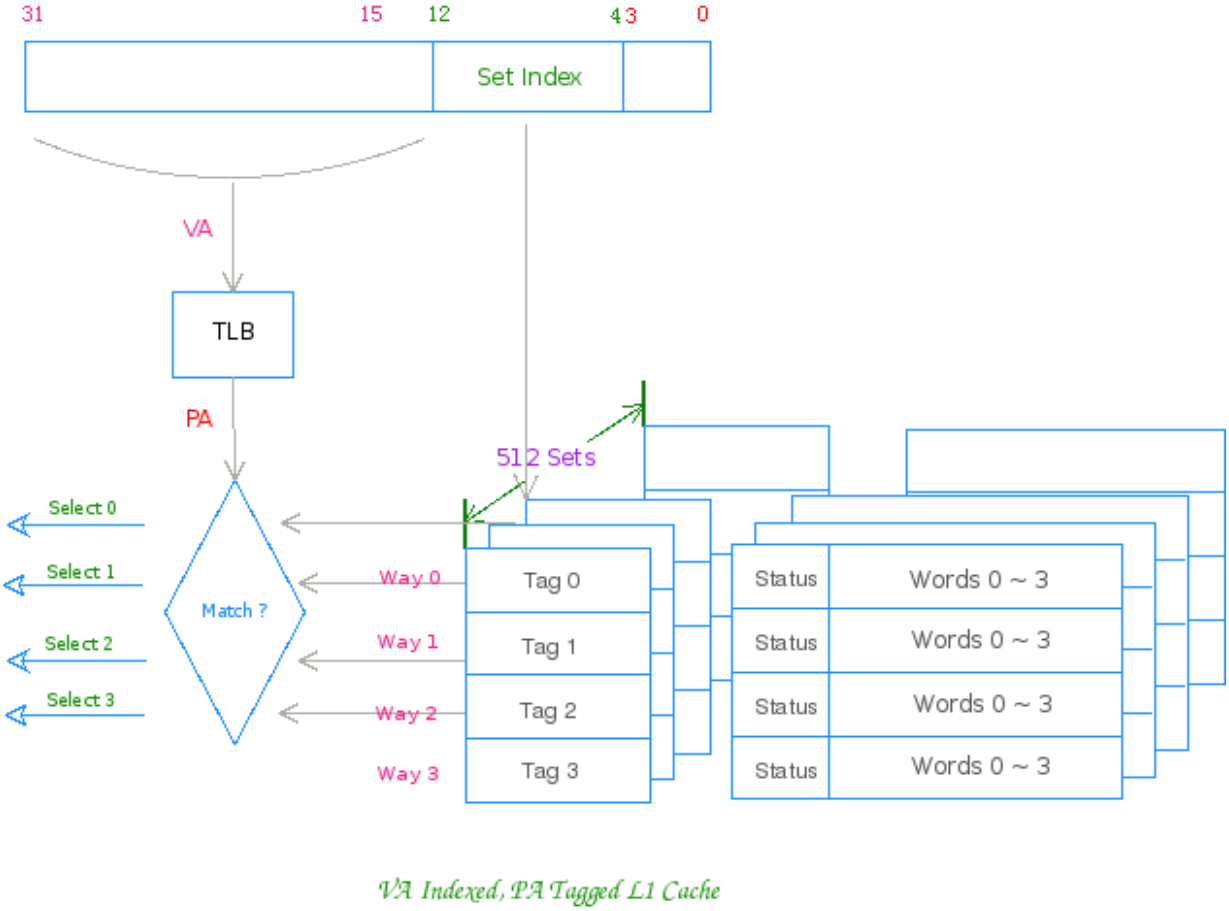


Addr[12:4] 用来索引组，9 bit 可索引 512 组
Addr[3:0] 用来行内索引字节，2^4 = 16 Bytes

现代体系结构的 CPU 多数皆有 MMU，因而其皆有虚拟地址空间与物理地址空间。指令流中的访存地址皆为 VA，在访存操作送给RAM控制器前 VA 要转换为 PA。Cache 的实现策略则要灵活的多，具体的实现可以用 VA 索引，PA 匹配；亦可 PA 索引，PA 匹配；乃至 VA 索引，VA 匹配 (SB1 I-Cache, 20KC I-Cache)；甚至 PA 索引，VA 匹配 (R6000)。

最为常见的MIPS L1 Cache 实现是 VA 索引，PA 匹配；这种实现可以在 VA经TLB转换为 PA 时，Cache控制器同时用VA索引 Cache 组，无需首先等待 VA 到 PA 的转换，尔后再索引、匹配，因此其性能优于 PA 索引，PA 匹配，但软件维护 Cache 一致性较后者复杂，且容易引起一些初看让人匪夷所思的问题。

VA 索引，PA 匹配 Cache 之工作方式图：



VA Indexed, PA Tagged L1 Cache

对于 L2 Cache，因为性能的要求已不再那么苛刻，为降低复杂度通常的实现皆是 PA 索引，PA 匹配。

正是因为MIPS L1 Cache 实现时，执着于对性能的追求，使得系统程序员负担加重 :) VA 索引，PA 匹配的方式极其容易出现 Cache 别名 (Aliases)问题，这个需要对内核中凡是有可能出现 Cache 别名的地方都要谨慎周密的加以考虑。PowerPC 的系统程序员比较幸福，L1 Cache 的实现皆是PA索引，PA 匹配，因此他们根本不需要考虑 Cache 别名问题。

3.2 Write-through or Write-back

在写命中(store hit)时，Cache的实现亦有两种策略：Write-back 和 Write-through. Write-through 的Cache，在 write hit 时，会将数据更新到 Cache 和 RAM. Write-back 的Cache，在 write hit 时，则仅将数据更新到 Cache 且将被更新的行标为 'dirty'，当该行被替换时控制器才将该行数据写回到内存。

对于 Write-back 的 Cache，在连续多次写数据时可以节约总线带宽，性能要好于 Write-through，但由于其缓存的数据往往是最新的，与内存中的数据多数时候是不一致的，因此需要软件来维护其一致性。

3.3 Read-allocate and Write-allocate

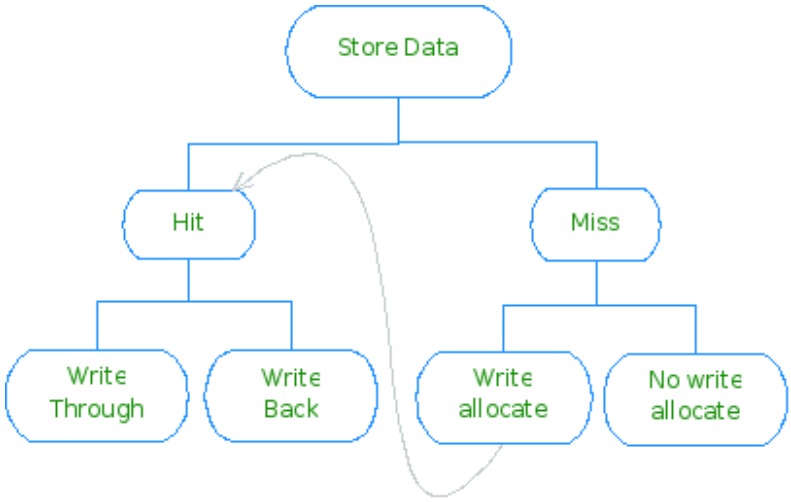
在写数据时 Cache miss，实际实现中有 2 种策略：Write-allocate 和 No write-allocate.前者的处理方式是先分配一行，后从 RAM 中读数据填充之（相当于一个 read miss refill 过程），最后才将数据写入 Cache（到此，亦会根据）. No read-allocate (Write-around)的处理方式则是绕过这一级 Cache（不分配 Cache line），直接将数据送到下一级 Cache/Memory. 有些 MIPS 的实现两种写策略皆支持。

读数据时 Cache miss，实际实现中亦有 2 种策略：Read-allocate 和 No read-allocate (Read through). 现代的实现一般皆为 Read-allocate，即：先从 Cache 中分配一行，后从 RAM 中读数据填充之，尔后将数据传给 CPU. No read-allocate则是直接从 RAM 取数据到CPU（不经Cache）。若非特别指出，读策略皆默认为 Read-allocate.

3.4 策略综合

读策略的采用比较简单，一般皆为 Read-allocate，一般不会特别指出，有些文档直接提 Write-through/Write-back with Read-allocate 是指此实现非 Write-allocate，即其写策略为 No write allocate.

下图为写策略的一个总结：



	Write Through	**Write Back**
Write allocate	On hits it writes to cache and main memory; On misses it updates the block in main memory and brings the block to the cache; Bringing the block to cache on a miss does not make a lot of sense in this combination because the next hit to this block will generate a write to main memory anyway	On hits it writes to cache setting “dirty” bit for the block, main memory is not updated; On misses it updates the block in main memory and brings the block to the cache; Subsequent writes to the same block, if the block originally caused a miss, will hit in the cache next time, setting dirty bit for the block. That will eliminate extra memory accesses and result in very efficient execution compared with Write Through with Write Allocate combination.

	Write Through	Write Back
No write allocate	On hits it writes to cache and main memory; On misses it updates the block in main memory not bringing that block to the cache; Subsequent writes to the block will update main memory because Write Through policy is employed. So, some time is saved not bringing the block in the cache on a miss because it appears useless anyway.	On hits it writes to cache setting “dirty” bit for the block, main memory is not updated; On misses it updates the block in main memory not bringing that block to the cache; Subsequent writes to the same block, if the block originally caused a miss, will generate misses all the way and result in very inefficient execution.

3.5 MIPS 实现实例

Au1100/Au1200 L1 D-Cache: Write-back with read-allocate (no write-allocate)

20Kc L1 D-Cache: Write-through with no write-allocate

Loongson2E/2F L1 D-Cache: Write back with no write-allocate

4. MIPS Cache 控制接口

MIPS 体系结构引入了如下的寄存器和指令作为软硬件的接口，便于软件管理 Cache:

4.1 TagHi and TagLo

TagHi 和 TagLo 用于暂存 Cache 行的 Tag 域数据，皆为 32 位 CP0 寄存器；其中 TagHi 只在物理地址长度超过 36 位时使用。执行 Index load tag 操作，则读出 Cache 行的 Tag 于这对寄存器中；执行 Index store tag 操作，则将寄存器中的数据写入对应的 Cache 行的 Tag 域。实际应用中，这两个寄存器主要在 Cache 初始化时用到，用在将未知状态的 Cache 行的 Tag 域置为 0。

MIPS 体系结构设计哲学就是能让软件做的事，尽可能让软件做。因此 Cache 的初始化（上电后 Cache 状态未知，一般都是由硬件来初始化），也留给了软件。参考的 Cache 初始化过程可能参看后面的第 6 章。

4.2 cache instruction

不同于 PowerPC 引入多条 dcbX, icbX 指令来管理 Cache，MIPS 体系结构只引入 cache 指令作为软件控制 cache 的统一接口，至于对哪个 Cache (L1 I, L1 D, L2, L3)、执行何种操作皆由一个整型值 ops 指定：

```
cache  ops, addr
```

4.2.1 ops

ops 编码进 4 字节的指令，占用 5 位，低 2 位指定 Cache 类型 (L1 I, L1 D, L2, L3)，高 3 位指定执行的操作。

其能指定的基本操作有：

Index Invalidate(Writeback)	000	
Index Load Tag	001	读 Cache 行之Tag
Index Store Tag	010	写 Cache 行之Tag
Hit Invalidate	100	
Fill, Hit Writeback Invalidate	101	
Hit Writeback	110	
Fetch and Lock	111	

MIPS32/64 体系结构规定必须实现3个：Index Invalidate(Writeback), Index Store Tag, Hit Writeback Invalidate，实际中最常用的也就是这3个操作。

为便于记忆可用如下的宏定义代替数字：

Index_Invalidate_I	0x00	(000 00)
Index_Writeback_Inv_D	0x01	(000 01)
Index_Writeback_Inv_SD	0x03	(000 11)
Index_Load_Tag_I	0x04	(001 00)
Index_Load_Tag_D	0x05	(001 01)
Index_Store_Tag_I	0x08	(010 00)
Index_Store_Tag_D	0x09	(010 01)
Hit_Invalidate_I	0x10	(100 00)
Hit_Invalidate_D	0x11	(100 01)
Hit_Invalidate_SD	0x13	(100 11)
Hit_Writeback_Inv_D	0x15	(101 00)
Hit_Writeback_Inv_SD	0x17	(101 11)

ARM 用 Coprocessor 寄存器写指令 MCR 来实现 Cache 的管理：

```
MCR    p15, 0, <Rd>, c7, <cN>, <opcode2>
```

<cN> 和 <opcode2> 组合，指定要做的操作，类似于 ops。

相较于 ARM，MIPS 的设计要简单、优美的多。

4.2.1 addr

addr 是为一个地址 (PA / VA)，用于选择针对 Cache 哪一行进行操作。有两种操作类型：Hit 和 Index。Hit 型的操作类似 Cache 的正常工作方式，其过程为： 先用 addr 的组选位（位12~4）索引组，同时将 VA 经 TLB 转换为 PA，然后匹配 Tag，命中 (Hit) 则对相应的行执行操作，否则什么也不做。

为了能刷新 Cache 时能精确访问到具体的行，MIPS 体系结构在 Hit 型的访问方式上又引入了 Index 型的方式：该方式无需匹配（不需要虚实地址转换），组选后直接用地址的中部（位14~13）进行组内索引（路选）：

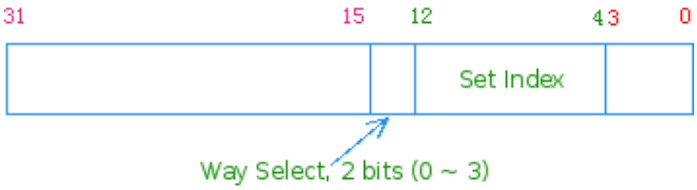


Figure 8: Indexed 方式地址分解

Index 型的操作，能使用虚拟地址 (VA) 精确操作组内的某一行，可用于不知道物理地址 (PA) 的场合。如在一个进程的内核上下文中，快速flush 整个 L1 D-Cache，按行 Index_Writeback_Inv_D dcache_size 大小的虚址空间即可。

cache 指令可以引起 TLB Refill 异常(ExeCode: TLBL)，TLB Invalid 异常，不会引起 TLB Modified 异常。

对于 Index 型的操作，addr 要使用不经 TLB 映射的(unmapped) 的地址，避免引起 TLB 异常。此仅对 PA 索引的 Cache 而言，VA 索引的 Cache，Index 型的操作根本不会走 TLB。

From BCM7410:

An indexed Cache inst. such as indexed invalidation, load tag and store tag must use an operand address in a unmapped segment, otherwise it may cause a unexpected TLB exception or it may apply to an erroneous Cache location.

From MIPS32 R1/R2:

For index operations (where the address is used to index the Cache but need not match the Cache tag) software should use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS.

From TX49:

TLB Refill and TLB Invalid exceptions can occur on any operation. For Index operations (where the physical address is used to index the cache but need not match the cache tag) unmapped addresses may be used to avoid TLB exceptions. This operation never causes TLB Modified or Virtual Coherency exceptions.

注：以上是针对 PA索引，PA匹配的情况

特别留意：有一部分 MIPS Cache 的实现使用地址的最低位 (LSB, Least Significant Bit) 进行路选，如东芝 TX49 系列的实现，其4 路组相联 Cache 索引方式访问使用 Addr[1:0] 作为路选位！[10] Page A-39

其它许多的 MIPS 实现并没有在其文档里非常清晰的注明这个，实际使用中要注意。 一个简单的方法就是查看Linux Kernel 里 struct cache_desc 结构中的 waybit 成员，若其值为 0，则其使用 LSB 进行路选 （参看下节）。如 NEC VR5500, R10000, Loongson2E 皆使用 LSB 作为其路选位。如果拿不准，可以写个测试函数，Index_Load_Tag_D 0x8000 0000 ~ 0x8000 0002 的 Tag 看其是否相同，更严谨的可以再将 0x8000 0001 处的 Tag 清为0，尔后再 Index_Load_Tag_D 0x8000 0000 ~ 0x8000 0001 的 2 个 Tag。

4.3 Linux/MIPS Cache 操作的初步封装

所有 Cache 操作的初步封装皆位于：[include/asm-mips/r4kcache.h]

其定义了两个宏， __BUILD_BLAST_CACHE 和 __BUILD_BLAST_CACHE_RANGE 分别用于批量生成形如下面的函数：

```
blast_xcachexx(void)                    # 以 Indexed 方式 flush 整个 Cache
blast_xcachexx_page(addr)               # Hit 方式 flush addr 处的一个 Page
blast_xcachexx_page_indexed(addr)       # 以 Indexed 方式 flush addr 处的一个Page

blast_xcache_range(start, end)          # Hit 方式 flush start 到 end 在 Cache 中的内容
protected_xcache_range(start, end)      # Hit、保护方式 flush start 到 end 在 Cache 中的内容
```

下面是一些预处理后的实例：

```
blast_icache16()
blast_dcache16()
blast_icache32_page()
blast_dcache32_page()
blast_icache128_page_indexed()
blast_dcache128_page_indexed()

blast_dcache_range()
blast_inv_dcache_range()
```

16, 32, 128 为行大小

这些函数会被位于 arch/mips/mm/c-r4k.c 中的形如 flush_cache_all/flush_cache_page/flush_cache_range/... 的函数所调用。

先看看所有 __BUILD_BLAST_CACHE 的调用：

```
396 __BUILD_BLAST_CACHE(d, dcache, Index_Writeback_Inv_D, Hit_Writeback_Inv_D, 16)
397 __BUILD_BLAST_CACHE(i, icache, Index_Invalidate_I, Hit_Invalidate_I, 16)
398 __BUILD_BLAST_CACHE(s, scache, Index_Writeback_Inv_SD, Hit_Writeback_Inv_SD, 16)
399 __BUILD_BLAST_CACHE(d, dcache, Index_Writeback_Inv_D, Hit_Writeback_Inv_D, 32)
400 __BUILD_BLAST_CACHE(i, icache, Index_Invalidate_I, Hit_Invalidate_I, 32)
401 __BUILD_BLAST_CACHE(s, scache, Index_Writeback_Inv_SD, Hit_Writeback_Inv_SD, 32)
402 __BUILD_BLAST_CACHE(i, icache, Index_Invalidate_I, Hit_Invalidate_I, 64)
403 __BUILD_BLAST_CACHE(s, scache, Index_Writeback_Inv_SD, Hit_Writeback_Inv_SD, 64)
404 __BUILD_BLAST_CACHE(s, scache, Index_Writeback_Inv_SD, Hit_Writeback_Inv_SD, 128)
```

用到的 Cache 操作仅有：Index_Invalidate_I/Hit_Invalidate_I, Index_Writeback_Inv_D/Hit_Writeback_Inv_D, Index_Writeback_Inv_SD/Hit_Writeback_Inv_SD。单从功能上看，OS 仅仅使用了索引方式的写回置无效（ICache 仅需置无效，无需写回）和命中方式的写回置无效（ICache

仅需置无效）。行大小为 16 和 32字节的各有一套函数；实际中尚未遇到64字节行大小的 L1 DCache，因此仅 L1 ICache 和 L2 需要一组函数；行大小 128 字节的 Cache 仅有 L2 Cache，故只定义 128 字节行大小的 SCache 即可。

宏 __BUILD_BLAST_CACHE 的定义如下：

```
342 /* build blast_xxx, blast_xxx_page, blast_xxx_page_indexed */
343 #define __BUILD_BLAST_CACHE (pfx, desc, indexop, hitop, lsize)        \
344 static inline void blast_##pfx##cache##lsize(void)        \
345 {                           \
346     unsigned long start = INDEX_BASE;           \
347     unsigned long end = start + current_cpu_data.desc.waysize;  \
348     unsigned long ws_inc = 1UL << current_cpu_data.desc.waybit; \
349     unsigned long ws_end = current_cpu_data.desc.ways <<       \
350                    current_cpu_data.desc.waybit;       \
351     unsigned long ws, addr;                 \
352                             \
353     __##pfx##flush_prologue                 \
354                             \
355     for (ws = 0; ws < ws_end; ws += ws_inc)           \
356         for (addr = start; addr < end; addr += lsize * 32)  \
357             cache##lsize##_unroll32(addr|ws,indexop);   \
358                             \
359     __##pfx##flush_epilogue                 \
360 }                           \
361                             \
362 static inline void blast_##pfx##cache##lsize##_page(unsigned long page) \
363 {                           \
364     unsigned long start = page;             \
365     unsigned long end = page + PAGE_SIZE;           \
366                             \
367     __##pfx##flush_prologue                 \
368                             \
369     do {                        \
370         cache##lsize##_unroll32(start,hitop);         \
371         start += lsize * 32;                \
372     } while (start < end);                  \
373                             \
374     __##pfx##flush_epilogue                 \
375 }                           \
376                             \
377 static inline void blast_##pfx##cache##lsize##_page_indexed(unsigned long page) \
378 {                           \
379     unsigned long indexmask = current_cpu_data.desc.waysize - 1;    \
380     unsigned long start = INDEX_BASE + (page & indexmask);      \
```

```
381     unsigned long end = start + PAGE_SIZE;              \
382     unsigned long ws_inc = 1UL << current_cpu_data.desc.waybit; \
383     unsigned long ws_end = current_cpu_data.desc.ways <<        \
384                            current_cpu_data.desc.waybit;        \
385     unsigned long ws, addr;                     \
386                                 \
387     __##pfx##flush_prologue                     \
388                                 \
389     for (ws = 0; ws < ws_end; ws += ws_inc)             \
390         for (addr = start; addr < end; addr += lsize * 32)  \
391             cache##lsize##_unroll32(addr|ws,indexop);   \
392                                 \
393     __##pfx##flush_epilogue                     \
394 }
395
```

__xxflush_prologue, __xxflush_epilogue 在非 MIPS MT (Multi-Thread) 平台上定义为空，忽略。

346 行的 INDEX_BASE 为常量：

```
31  #define INDEX_BASE  CKSEG0
```

32 位平台上 CKSEG0 为 0x8000 0000

current_cpu_data 为一个宏，预处理后展开成 cpu_data[smp_processor_id()]，cpu_data 之类型为 struct cpuinfo_mips; 定义于 [include/asm-mips/cpu-info.h]:

```
struct cpuinfo_mips {
    unsigned long       udelay_val;
    unsigned long       asid_cache;
#if defined(CONFIG_SGI_IP27)
//  cpuid_t     p_cpuid;    /* PROM assigned cpuid */
    cnodeid_t   p_nodeid;   /* my node ID in compact-id-space */
    nasid_t     p_nasid;    /* my node ID in numa-as-id-space */
    unsigned char   p_slice;    /* Physical position on node board */
#endif
    /*
     * Capability and feature descriptor structure for MIPS CPU
     */
    unsigned long       options;
    unsigned long       ases;
```

```
    unsigned int        processor_id;
    unsigned int        fpu_id;
    unsigned int        cputype;
    int         isa_level;
    int         tlbsize;
    struct cache_desc   icache; /* Primary I-cache */
    struct cache_desc   dcache; /* Primary D or combined I/D cache */
    struct cache_desc   scache; /* Secondary cache */
    struct cache_desc   tcache; /* Tertiary/split secondary cache */
#if defined(CONFIG_MIPS_MT_SMTC)
    /*
     * In the MIPS MT "SMTC" model, each TC is considered
     * to be a "CPU" for the purposes of scheduling, but
     * exception resources, ASID spaces, etc, are common
     * to all TCs within the same VPE.
     */
    int         vpe_id;  /* Virtual Processor number */
    int         tc_id;   /* Thread Context number */
#endif /* CONFIG_MIPS_MT */
    void            *data;  /* Additional data */
} __attribute__((aligned(SMP_CACHE_BYTES)));
```

该结构中有 4 个用来描述 Cache 的 struct cache_desc 结构：

```
struct cache_desc {
    unsigned int waysize;       /* Bytes per way */
    unsigned short sets;        /* Number of lines per set */
    unsigned char ways;         /* Number of ways */
    unsigned char linesz;       /* Size of line in bytes */
    unsigned char waybit;       /* Bits to select in a cache set */
    unsigned char flags;        /* Flags describing cache properties */
};
```

其中：

$$waysize = cache\_size / ways$$

$$sets = waysize / linesz$$

$$waybit = \log_2(waysize)$$

Cache 的基本参数 cache_size, ways 和 linesz 写在 cp0_configX 寄存器里，X 根据具体实现的不同为 0 ~ 2。这些成员在 probe_pcache() 和 setup_scache() 里设置，其执行的函数调用拓扑为：

```
start_kernel()
    |----- ...
```

```
|----- setup_arch()
|----- ...
|----- build_all_zonelists()
|----- page_alloc_init()
|----- ...
|----- trap_init()  ---> per_cpu_trap_init() ---> cpu_cache_init() ---> r4k_cache_init()
|----- ...
|----- mem_init()
|----- ...
`----- rest_init()


r4k_cache_init()
    |----- ...
    |----- probe_pcache()
    |----- setup_scache()
    |----- ...
    `----- ...
```

这两个函数定义在 [arch/mips/mm/c-r4k.c]，完成的基本操作即是读出 config0/config1/config2 的值，根据文档的描述移位计算出 cache_size, ways 和 line_size.

此处以上文描述的 L1 DCache（32KB、4路组相联、行大小为16 Bytes）为例来看看该宏展开后的函数 blast_dcache16() ，代码集中在 344~360：

```
start = 0x8000 0000
end = 0x8000 2000           -----> 8KB, 一路
ws_inc = 8KB
ws_end = 32KB

for (ws = 0; ws < ws_end; ws += ws_inc)
    for (addr = start; addr < end; addr += 16 * 32)
        cache16_unroll32(addr|ws,indexop);
```

可以看到，内层循环连续扫描了 8KB 大小的 Cache，实际上是对一路内的 512 行执行 indexop 操作 (Index_Writeback_Inv_D)，当 ws = 0 时，操作的是第一路，ws = 8KB 时操作的是第二路。

对用地址最低位进行路选的 Cache，waybit = 0，ws_inc = 1, ws_end = 4.

cache16_unroll32() 用于 cache indexop 32个连续 Cache 行，起始的第一行用 addr|ws 索引之。其定义于同一文件：

```
238 #define cache16_unroll32(base,op)                \
239   __asm__ __volatile__(                   \
240   "   .set push                  \n" \
241   "   .set noreorder                 \n" \
242   "   .set mips3                 \n" \
243   "   cache %1, 0x000(%0); cache %1, 0x010(%0)    \n" \
244   "   cache %1, 0x020(%0); cache %1, 0x030(%0)    \n" \
245   "   cache %1, 0x040(%0); cache %1, 0x050(%0)    \n" \
246   "   cache %1, 0x060(%0); cache %1, 0x070(%0)    \n" \
247   "   cache %1, 0x080(%0); cache %1, 0x090(%0)    \n" \
248   "   cache %1, 0x0a0(%0); cache %1, 0x0b0(%0)    \n" \
249   "   cache %1, 0x0c0(%0); cache %1, 0x0d0(%0)    \n" \
250   "   cache %1, 0x0e0(%0); cache %1, 0x0f0(%0)    \n" \
251   "   cache %1, 0x100(%0); cache %1, 0x110(%0)    \n" \
252   "   cache %1, 0x120(%0); cache %1, 0x130(%0)    \n" \
253   "   cache %1, 0x140(%0); cache %1, 0x150(%0)    \n" \
254   "   cache %1, 0x160(%0); cache %1, 0x170(%0)    \n" \
255   "   cache %1, 0x180(%0); cache %1, 0x190(%0)    \n" \
256   "   cache %1, 0x1a0(%0); cache %1, 0x1b0(%0)    \n" \
257   "   cache %1, 0x1c0(%0); cache %1, 0x1d0(%0)    \n" \
258   "   cache %1, 0x1e0(%0); cache %1, 0x1f0(%0)    \n" \
259   "   .set pop                   \n" \
260       :                       \
261       : "r" (base),                   \
262         "i" (op));
```

cache16 表示行大小为16 的 Cache；unroll32 表示仅对 32 个行操作。此外根据行的大小还定义了 cache32_unroll32, cache64_unroll32, cache128_unroll32.

其它的两个函数 blast_dcache16_page(page) blast_dcache16_page_indexed(page) 思想类似，只是前者用命中 (Hit)的方式 flush 一个页，不需要精确控制，代码相对要简洁些。

__BUILD_BLAST_CACHE_RANGE 定义如下：

```
406 /* build blast_xxx_range, protected_blast_xxx_range */
407 #define __BUILD_BLAST_CACHE_RANGE(pfx, desc, hitop, prot) \
408 static inline void prot##blast_##pfx##cache##_range(unsigned long start, \
409                         unsigned long end)  \
410 {                           \
```

```
411     unsigned long lsize = cpu_##desc##_line_size();        \
412     unsigned long addr = start & ~(lsize - 1);         \
413     unsigned long aend = (end - 1) & ~(lsize - 1);         \
414                             \
415     __##pfx##flush_prologue                 \
416                             \
417     while (1) {                     \
418         prot##cache_op(hitop, addr);              \
419         if (addr == aend)                 \
420             break;                  \
421         addr += lsize;                  \
422     }                           \
423                             \
424     __##pfx##flush_epilogue                 \
425 }
426
427 __BUILD_BLAST_CACHE_RANGE(d, dcache, Hit_Writeback_Inv_D, protected_)
428 __BUILD_BLAST_CACHE_RANGE(s, scache, Hit_Writeback_Inv_SD, protected_)
429 __BUILD_BLAST_CACHE_RANGE(i, icache, Hit_Invalidate_I, protected_)
430 __BUILD_BLAST_CACHE_RANGE(d, dcache, Hit_Writeback_Inv_D, )
431 __BUILD_BLAST_CACHE_RANGE(s, scache, Hit_Writeback_Inv_SD, )
432 /* blast_inv_dcache_range */
433 __BUILD_BLAST_CACHE_RANGE(inv_d, dcache, Hit_Invalidate_D, )
434 __BUILD_BLAST_CACHE_RANGE(inv_s, scache, Hit_Invalidate_SD, )
```

从 427~434 的使用来看，418 行处会用到两个宏：cache_op(addr,op) 和 protected_cache_op(addr, op):

```
33  #define cache_op(op,addr)                   \
34      __asm__ __volatile__(                   \
35      "   .set    push                \n" \
36      "   .set    noreorder               \n" \
37      "   .set    mips3\n\t               \n" \
38      "   cache   %0, %1                  \n" \
39      "   .set    pop             \n" \
40      :                           \
41      : "i" (op), "R" (*(unsigned char *)(addr)))

193 #define protected_cache_op(op,addr)             \
194     __asm__ __volatile__(                   \
195     "   .set    push            \n"     \
196     "   .set    noreorder       \n"     \
197     "   .set    mips3           \n"     \
198     "1: cache   %0, (%1)        \n"     \
199     "2: .set    pop         \n"     \
200     "   .section __ex_table,\"a\"   \n"     \
201     "   "STR(PTR)" 1b, 2b       \n"     \
```

```
202     "   .previous"              \
203     :                       \
204     : "i" (op), "r" (addr))
```

33 ~ 41 行没什么，就是在 C 语言环境里使用 cache 指令。200 ~ 201 行则将198 行的指令地址和紧随其后的指令地址放入 __ex_table 节中，__ex_table 是为一个异常处理表，内核设计时将其作为 ELF 文件的一节，32bit 平台上其每项为 8 字节，放两个地址，第一为可能引起异常的指令的地址，第二为相应的处理函数地址（指针），如 arch/mips/kernel/scall32-o32.S 中：

```
164     .section __ex_table,"a"
165     PTR 1b,bad_stack
166     PTR 2b,bad_stack
167     PTR 3b,bad_stack
168     PTR 4b,bad_stack
169     .previous
170
171     /*
172      * The stackpointer for a call with more than 4 arguments is bad.
173      * We probably should handle this case a bit more drastic.
174      */
175 bad_stack:
176     negu    v0              # error
177     sw  v0, PT_R0(sp)
178     sw  v0, PT_R2(sp)
179     li  t0, 1               # set error flag
180     sw  t0, PT_R7(sp)
181     j   o32_syscall_exit
```

PTR 在 32bit 平台上为 .word，64bit 平台上为 .dword，皆为汇编中用来定义一个字的。.word 0x80000000, 0x80004000 则在 data 节定义两个字。
可以看到引入宏 protected_cache_op 的目的旨在将其纳入异常处理系统，对这一操作增加一些保护。

至此，由该宏生成的一系列的 blast_xcache_range(start, end) protected_xcache_range(start, end) 函数就很容易理解了。

5. Cache Aliases Issue

这个问题应该算是 MIPS 世界里一个比较大的问题，内核为解决这个问题费了不少的劲，始终未能彻底地解决，由此引起的 Bug，现象诡异，极难调试。

5.1 简述

在使用 VA 索引，PA 匹配的 Cache 里，可能存在多个 VA 映射到同一个 PA 的情形（共享内存），这样的 Cache 是先用 VA 索引组，则他们可能会索引到不同的组（只要组索引位不一样），尽管尔后用于匹配的 PA Tag 相同，但因为不在同一组，因此无可避免的在 Cache 中会有两个镜像。
Cache Aliases 就是指同一物理行在 Cache 里有两个或多个“镜像”（位于不同的组）的情形。

5.2 现象

设 VA_1、VA_2 皆映射到 PA_0，VA_1 的组选位为 0，VA_2 的组选位为 3，某时用 VA_1 取值后向 VA_1 更新数据，则对 write back 的 Cache 最新的数据缓存于 Cache 的组 0 (Set 0)，尔后用 VA_2 访问同一内存空间，则其取到的值并非是最新的，这即是典型的 Cache Aliases 造成的存储不一致的问题。

5.3 判断

对于 VA 索引，PA 匹配的 Cache，如果 Way_Size > PAGE_SIZE，则存在 Cache Aliases 的问题。

在 Cache 的正常工作情况下，用于索引的 VA，只需用其低位索引到具体哪一组即可，因此其需要的 VA 位数为 $\log_2$(Way_Size)，对 32KB, 4-way, Line Size 16B 的 Cache，其用到的位数为 13 (VAddr[12:0]) 其中 VAddr[3:0] 用于行内索引，我们可以宏观的认为 VA 的低 13 位就是用于索引一路的 Cache （大小为 Way_Size）。

对于采用 PAGE_SIZE 为页大小的 OS 来说，VA 的低 $\log_2$(PAGE_SIZE) 位是与 PA 相同

的。当 Way_Size > PAGE_SIZE 时，即 $\log_2$(Way_Size) > $\log_2$(PAGE_SIZE)， 则用于索引一路 Cache 的 VA 低位较用于页内偏移的低位多了 $\log_2$(Way_Size) - $\log_2$(PAGE_SIZE)，为了分析的方便，往往将这个多出的位称为颜色位 (color bit)。对于上述的 Cache，若系统采用 4KB 的 PAGE_SIZE，则VAddr[12] 即为颜色位。

两个颜色位不一样的 VA 对应同一 PA，会索引到不同的组，因此就会出现 Aliases 问题。

若能保证用于索引的 VA 低位小于或等于 $\log_2$(PAGE_SIZE)，则用于索引的 VA 低位与 PA 的低位一致，即便多个 VA 对应同一 PA，因其用于索引的低位皆相同，其皆索引到相同的组，组内 PA Tag 一匹配，是不会出现多个“镜像”的。上述的 Cache，若系统采用 16KB 的 PAGE_SIZE，则 VAddr[13:0] 始终与物理地址一致，而用于组索引的仅VAddr[12:0] 而已，颜色位根本不会出现。

对 Way_Size > PAGE_SIZE，若 OS 在构建页表进行虚实地址映射时，能确保映射到同一 PA 的所有 VA 的颜色位相同，则也能彻底的解决问题。只是这样会增加 OS 存储管理子系统的负担，影响性能。

5.4 解决

解决的基本思想有两个，积极的和保守的。

积极的基本思想为：软件确保多个 VA（映射到同一 PA）皆索引到相同的组（组索引位相同）。最激进的是直接消除颜色位，这个可以通过提高 OS 的页大小来实现；温和点的是 OS 保证映射到同一 PA 的所有 VA 的颜色位相同，这个需要修改 Kernel 的存储管理子系统。

保守的解决方法是目前 Linux MIPS 所采用的主要方法，其是在所有可能出现 Cache Aliases 的地方 flush Cache。例如进程 Pi 在往 VA_1 (映射到PA_0) 更新数据后，数据没有立即被写回到内存，此时发生一个中断使 Pi 陷入内核态，此时Kernel 需要从 PA_0 读取数据，内核从用户进程读取数据往往用的是不需 TLB 映射的 KSEG0 空间地址访问之，因此内核用 (0x80000000 + PA_0) 获取用户进程之数据，此时就会出现 Cache Aliases，内核

取到的数据就会与实际的数据不一致，保守的做法就是：索引方式 flush VA_1 ，将 VA_1 对应在 Cache 中的数据行更新到内存，这样内核从 (0x80000000 + PA_0) 取得的数据就是最新的了。

5.4.1 焦点

内核中最易出现 Cache Aliases 的地方，主要位于如下几个函数：

```
void copy_to_user_page (struct vm_area_struct *vma, struct page *page,
                        unsigned long user_vaddr,
                        void *dst, void *src, int len)

called by access_process_vm(),  vma, page and user_vaddr is the source

void copy_from_user_page (struct vm_area_struct *vma, struct page *page,
                          unsigned long user_vaddr,
                          void *dst, void *src, int len)
```

MIPS 下，他们都定义于 [arch/mips/mm/init.c]。这些函数在内核里一般用在一个进程的内核态上下文内拷贝另一个进程的数据，例如内核中的 ptrace 机制，就频繁使用这些函数。

现象上，极易出现在 fork() 后，子进程在 COW (Copy On Write) 某个页时。其原因大致为，子进程要写某个页时，因为是共享的父进程的物理页，则内核首先为其分配一个页后，首先要将与父进程共享的页 Copy 过来，内核里不知为何子进程映射到这个页的 VA 往往不同于父进程，这就出现了典型的两个不同的 VA 指向同一个 PA，此时极其容易出现 Cache Aliases，数据的不一致，往往造成一些指针“飞”了，一旦使用这些指针访问数据，用户态则常会 "Segmentation fault" 或者 "Bus error"，内核态则常常直接 Oops/Kernel Panic。

5.4.2 Linux Cache Flush API

对各种需要 flush Cache 的情形，kernel 封装了一整套 API 来为所有的体系结构服务，关于这些 API 的详细描述可以参考源码中的 [Documentation/cachetlb.txt]，描述得很详细，MIPS 相关的声明则位于 [include/asm-mips/cacheflush.h]，适用于大部分 MIPS

平台的实现则位于 [arch/mips/mm/c-r4k.c]：

void **flush_cache_all** ()
void **__flush_cache_all** ()
void **flush_cache_mm (**struct mm_struct *mm)
void **flush_cache_dup_mm** (struct mm_struct *mm)
void **flush_cache_page** (struct vm_area_struct *vma,
unsigned long addr, unsigned long pfn)
void **flush_cache_range** (struct vm_area_struct *vma,
unsigned long start, unsigned long end)
void **flush_cache_vmap** (unsigned long start, unsigned long end)
void **flush_cache_vunmap** (unsigned long start, unsigned long end)
void **flush_icache_range** (unsigned long start, unsigned long end)
void **flush_icache_page** (struct vm_area_struct *vma, struct page *page)

void **flush_anon_page** (struct vm_area_struct *vma, struct page *page,
unsigned long vmaddr)
void **flush_dcache_page** (struct page *page)

MIPS 特别引入的：

void **flush_cache_sigtramp** (unsigned long addr)
void **flush_icache_all** ()
void **flush_data_cache_page** (unsigned long addr)
void **local_flush_data_cache_page** (unsigned long addr)

这些函数最终调用的都是 4.3 节描述的形如 blast_xxxxxxx () 的初步封装函数。择其要者概述如下：

5.4.2.1 `flush_cache_all()`

以索引方式刷新整个 DCache

5.4.2.2 `__flush_cache_all()`

在 flush_cache_all() 的基础上，额外以索引方式刷新 ICache 和 L2 Cache

5.4.2.3 `flush_data_cache_page(vaddr)`

以命中 (Hit) 方式刷新 PAGE_SIZE 大小的 DCache。因其是用Hit 方式，则需要保证当前所用之页表中 vaddr 中有相应的映射，该函数不负责页表的填充，直接

blast_dcachexx(vaddr)。

5.4.2.4 `flush_cache_page(vma, vaddr, pfn)`

该函数是对 ICache/DCache, Hit/Indexed 方式操作的高端封装，其会根据具体的情况选用合适的底层操作函数，如果发现是在一个进程的内核上下文中刷新另一个进程的 vaddr，则其会采用索引的方式来刷新，否则其会使用 Hit 的方式来刷新。vma 指向 vaddr 所属进程之 vm_area_struct 结构，pfn 为 vaddr 对应之物理页框号。

因 Cache 在缓存某个内存行时，在组内位置不固定，因此以 Indexed 方式刷新一行时要覆盖到组内所有行，这样就需要比 Hit 操作更多的代价。但因其无需走 TLB，可用在一个进程的内核上下文中刷新属于另一个进程的 vaddr。该函数的封装则是既考虑到了性能又兼顾了灵活性。

5.4.3 改进

针对 flush 过于频繁影响整体性能的情形，Linux MIPS 社区一直在做改进的努力。较大的改进是在 2006 年 12 月引入了 kmap_coherent/kunmap_coherent，重写了 copy_to_user_page/copy_from_user_page，又加了 copy_user_highpage。最早的 patch 之 commit 为 bcd02280，主要函数的定义皆位于 [arch/mips/mm/init.c]，可以如下命令查看之：

```
# git-clone git://git.linux-mips.org/pub/scm/linux mips_linux
# cd mips_linux/
# git show bcd02280
# git log -p arch/mips/mm/init.c
```

原始的邮件可以查看：http://www.uwsg.iu.edu/hypermail/linux/kernel/0610.2/1446.html

kmap_coherent 现在 KSEG2 分配一个与虚址 addr 具有相同颜色位的地址，而后将这个地址指向相同的物理页 page，在 TLB 里写入一个固定 (wired)的映射。其返回值就是这个新分配的、位于 KSEG2 高端的虚址：

```
void *kmap_coherent (struct page *page, unsigned long addr)
{
   enum fixed_addresses idx;
   unsigned long vaddr, flags, entrylo;
```

```
    unsigned long old_ctx;
    pte_t pte;
    int tlbidx;

    BUG_ON(Page_dcache_dirty(page));

    inc_preempt_count();                                    # 禁止抢占
    idx = (addr >> PAGE_SHIFT) & (FIX_N_COLOURS - 1);       # 取颜色位
#ifdef CONFIG_MIPS_MT_SMTC
    idx += FIX_N_COLOURS * smp_processor_id();
#endif
    vaddr = __fix_to_virt(FIX_CMAP_END - idx);              # 固定映射，定义于 asm/fixmap.h
    pte = mk_pte(page, PAGE_KERNEL);                        # 取 page 对应的页框号 PFN + Flags
#if defined(CONFIG_64BIT_PHYS_ADDR) && defined(CONFIG_CPU_MIPS32)
    entrylo = pte.pte_high;
#else
    entrylo = pte_val(pte) >> 6;
#endif

    ENTER_CRITICAL(flags);                                  # 关中断
    old_ctx = read_c0_entryhi();
    write_c0_entryhi(vaddr & (PAGE_MASK << 1));             # TLB 之 VPN + ASID
    write_c0_entrylo0(entrylo);                             # TLB 奇数项之 PFN + Flags
    write_c0_entrylo1(entrylo);                             # TLB 偶数项之 PFN + Flags
#ifdef CONFIG_MIPS_MT_SMTC
    set_pte(kmap_coherent_pte - (FIX_CMAP_END - idx), pte);
    /* preload TLB instead of local_flush_tlb_one() */
    mtc0_tlbw_hazard();
    tlb_probe();
    tlb_probe_hazard();
    tlbidx = read_c0_index();
    mtc0_tlbw_hazard();
    if (tlbidx < 0)
        tlb_write_random();
    else
        tlb_write_indexed();
#else
    tlbidx = read_c0_wired();
    write_c0_wired(tlbidx + 1);         # 分配一个固定的 TLB Entry
    write_c0_index(tlbidx);             # 设置 Index 为刚分配的 Entry
    mtc0_tlbw_hazard();
    tlb_write_indexed();                # 索引方式将 entryhi, entrylo0, entrylo1的值写入 Index 指定的 Entry
#endif
```

```
    tlbw_use_hazard();
    write_c0_entryhi(old_ctx);      # 恢复原来的 Entry_hi。TLB 在工作时会用 Entry_hi(ASID) 来区分
                                      TLB 项是否属于当前进程（属硬件范围）
    EXIT_CRITICAL(flags);           # 开中断

    return (void*) vaddr;
}
```

5.5 案例分析

5.5.1 现象

BCM56218 (BCM3302 MIPS core, 32KB 2-way DCache) SoC，Glibc_small rootfs on MTD，启动后，在最后运行 busybox 时，内核总会输出 "Segmentation fault" or "Bus error"：

```
.............
rting pid 590, console /dev/console: '/etc/init.d/rcS'
Bus error
Segmentation fault

Welcome to Wind River Linux

Please press Enter to activate this console.
Starting pid 595, console /dev/console: '/bin/sh'

Bus error
#
```

ls 一下，就会有如下的一些诡异现象：

```
# ls  /bin/
Segmentation fault

# ls  /
*** glibc detected *** -sh: malloc(): memory corruption: 0x33fb4af4 ***
Aborted

# ls  /proc
Unhandled kernel unaligned access[#2]:
Cpu 0
$ 0   : 00000000 1000fc00 00000000 65532043
```

```
$ 4   : 65532043 00000000 00000000 00000000
$ 8   : 812c2006 00001000 812c2000 fffffff4
$12   : 0000002e 00004000 00600000 0000006c
$16   : 00000000 65532043 812c5cf4 812efd38
$20   : 00000000 877ac120 004a02f8 812efd40
$24   : 80000000 33f222d0
$28   : 812ee000 812efcb0 00000001 8003f568
Hi    : 00000000
Lo    : 00000004
epc   : 8003f174 pid_task+0x1c/0x38     Not tainted
ra    : 8003f568 get_pid_task+0x2c/0x88
Status: 1000fc03    KERNEL EXL IE
Cause : 00000010
BadVA : 65532053
PrId  : 0002901a
Modules linked in:
Process sh (pid: 708, threadinfo=812ee000, task=8771f908)
Stack : 877ac120 812efd38 8009dd90 812efd38 812c2009 812a534c 800cab10 800900c0
        812c2009 812efd40 812efe60 812efd38 812c2009 812c5cf4 812efe60 800901b8
        811e3304 8009e564 812efe84 80330000 812c2009 812efd40 877a3dc4 812efe60
        00000000 00000000 004a02f8 00000000 80092358 8009290c 80193fb4 8002e788
        00008000 812efea5 877ac120 877a85ec 001313f5 00000003 812c2006 000012bf
        ...
Call Trace:
[<8003f174>] pid_task+0x1c/0x38
[<8003f568>] get_pid_task+0x2c/0x88
[<800cab10>] pid_revalidate+0x28/0x178
[<800901b8>] do_lookup+0x68/0x1fc
[<80092358>] __link_path_walk+0x5b0/0x1238
[<80093098>] link_path_walk+0xb8/0x1b4
[<800934fc>] do_path_lookup+0x140/0x36c
[<80094190>] __user_walk_fd+0x54/0x88
[<8008aa3c>] vfs_lstat_fd+0x28/0x68
[<8008aaa4>] sys_newlstat+0x28/0x50
[<8000d610>] syscall_trace_entry+0x70/0x90

Code: 03e00008  00001021  00021100 <8c630010> 00451023  00621023  1060fff9  2442ff34  03e00008
note: sh[708] exited with preempt_count 1
Segmentation fault

# ps ax
  PID  Uid     VCPU 0 Unable to handle kernel paging request at virtual address 73656d20, epc == 800bf6f0, ra
== 800bf6e8
mSize Stat CommaOops[#1]:
Cpu 0
$ 0   : 00000000 1000fc00 73656d20 7375621f
$ 4   : 81240568 73756220 00000000 87719854
$ 8   : 00000000 80370000 81253c50 ffffffff
```

```
$12   : 00000005 00000000 2aaa8290 7fcca278
$16   : 73656d18 877197fc 81240464 00000000
$20   : 81240560 00000020 00000020 81246e64
$24   : 00000000 801728e4
$28   : 81252000 81253e80 87719854 800bf6e8
Hi    : 00000000
Lo    : 00000d68
epc   : 800bf6f0 inotify_inode_queue_event+0x94/0x160     Not tainted
ra    : 800bf6e8 inotify_inode_queue_event+0x8c/0x160
Status: 1000fc03    KERNEL EXL IE
Cause : 80000008
BadVA : 73656d20
PrId  : 0002901a
Modules linked in:
Process ps (pid: 636, threadinfo=81252000, task=87792c90)
Stack : 00000000 800833d4 00000000 00000000 00000000 00000000 81240568 00000000
        875991d0 877197fc 81240464 00000000 87719854 00000020 00000000 00000000
        00000000 800c0148 81225000 7fcca8b5 80082fc0 00000000 81246e64 0000416d
        00000020 8123eb60 00000004 81246e64 81225000 7fcca8b5 80083488 11e1a2ff
        386d4414 11e1a2ff 386d4414 11e1a2ff 00000400 00000000 000003ff 7fcca958
        ...
Call Trace:
[<800bf6f0>] inotify_inode_queue_event+0x94/0x160
[<800c0148>] inotify_dentry_parent_queue_event+0xd8/0x190
[<80083488>] do_sys_open+0xa0/0x174
[<8000d610>] syscall_trace_entry+0x70/0x90


Code: afa20018  8e0200fc  2450fff8 <16820007> 8e030008  0802fddf  8fa40018  26220008  02208021
nd
    1 root       508 S    init
    2 root           SWN [ksoftirqd/0]
    3 root           SW  [watchdog/0]
    4 root           SW< [events/0]
    5 root           SW< [khelper]
    6 root           SW< [kthread]
   20 root           SW< [kblockd/0]
   31 root           SW  [pdflush]
   32 root           SW  [pdflush]
   33 root           SW< [kswapd0]
   34 root           SW< [aio/0]
   35 root           SW< [unionfs_siod/0]
Segmentation fault
```

执行其它的命令，亦会有类似的现象。

5.5.2 分析

通常的 "Segmentation fault" 系访问非法虚拟地址所致（页表没相应的映射）；"Bus error" 则常常是访问非对其的地址所致，例如从 0x8000 0003 读取一个字，大多数情况下内核会报 "Bus error"。

从几次内核 dump 出来的寄存器和栈来看，出现 "Segmentation fault" 的位置不固定，访问的非法地址差别较大，完全重现的概率微乎其微，因此从引起异常的指令，到函数，再到程序块的逆向分析思路根本行不通。印象中试了几次，几近抓狂。

后与 yshi 聊起，此 BSP 在 2.6.14 的版本时没有任何问题，更新到 2.6.21 后每次到运行 busybox 时，就会 hung 掉（此时 cpu_has_dc_aliases = 1），强制将 cpu_has_dc_aliases = 0 则出现上面描述的 "Segmentation fault" 现象，而且更重要的：将 kernel 配置成 uncached (CONFIG_UNCACHED=y) 后，系统能正常运行，没有任何问题，并表示十分怀疑是 Cache Aliases 引起的问题。yshi 提供的信息，给从正面突破打开了一个口子。

查看内核启动日志后发现该 MIPS core 使用 32KB 2-way 的 DCache，且其用虚址索引、物址匹配，内核默认使用 4KB 的页大小，则 way_size = 32KB/2 = 16KB > 4KB，显然存在 Cache Aliases。加上用 uncached 的内核起没有问题，因此初步判断是由 Cache Aliases 引起的。

查看代码发现 cpu_has_dc_aliases 这个变量应该为 1，指示说明这个处理器有 Cache Aliases 问题，如将其强制定义成 0，则 Kernel 中为 Cache Aliases 准备的额外的 Cache flush 皆会被绕过，造成大量的父子进程间数据不一致。

由此将强制 0 赋值去除，则问题转化成：在 2.6.14 没有任何问题，2.6.21 则启动到运行 busybox 时就 hung 了。

5.5.3 探索

用 SysRq 的 t 键查看 hung 时，系统所有的 Task，惊奇的发现 pid == 1 的 init “蒸发”了。

很自然的，kgdb 上，艰苦卓绝的跟踪后发现是在 run_init_process("/sbin/init") 中挂了，以行步进到 do_execve()，发现该函数能正确返回，继续跟踪发现在执行 syscall_exit 时挂的，眼见就能接近焦点，不幸的是，当单步到 restore_all 的第一条指令时， kgdb 的单步跟踪始终在不停的死循环，每次执行 si，PC 的值都不会变。尝试多次，无果。静心偶然想起，以前亦遇到过断点在异常返回代码中是 kgdb 常常不能正常工作，其原因大概是 kgdb 的实现亦依赖于内核的异常处理，应该是自己单步自己，搞出死循环了。

于是转而对比 2.6.21 相对于 2.6.14 所做的改进，期间和 yshi 闲聊，亦提到过 2.6.21 在一些地方使用 kmap_coherent 替换了 2.6.14 所用的 kmap_atomic，于是很快就找到了 5.4.3 节所提到的那个 patch。

稍作分析后，就尝试绕开所有使用 kmap_coherent 的代码（使用原 kmap_atomic + flush 的机制）发现没有 hung，可以进入 shell，但现象与将 cpu_has_dc_aliases 强制定义为 0 的现象类似，只是情况没那么糟糕，虽有 "Segmentation fault"、"Bus error" 但不太影响正常使用，显然是 kmap_coherent 导致了 hung。于是粗略判断为退回原有的机制，可能还有某些该 flush 的地方被疏漏了，如果将这些洞补上，应该就可以彻底消除这些讨厌的 "Segmentation fault" 了。

至此已有些眉目，考虑到市场的兄弟还等着发布，列了两个解决方向：

A. 保守的，也是最快的：参照 2.6.14，补上这些残余的疏漏；
B. 激进的，也是可能花时间很多的：使 kmap_coherent 工作正常。

5.5.4 解决

5.5.4.1 保守的

参照 [Documentation/cachetlb.txt] 的指引，和引入 kmap_coherent 的 patch，发现焦点集中在 3 个函数中：

```
copy_to_user_page()
copy_from_user_page()
copy_user_highpage()
```

kmap_coherent 就在这三个函数中被调用，且他们都会根据 cpu_has_dc_aliases 的值判断是否用 kmap_coherent，若其值为 0，则这些函数就会走另一条路径：

```
void copy_user_highpage(struct page *to, struct page *from,
    unsigned long vaddr, struct vm_area_struct *vma)
{
    void *vfrom, *vto;
    vto = kmap_atomic(to, KM_USER1);
    if (cpu_has_dc_aliases) {
        vfrom = kmap_coherent(from, vaddr);
        copy_page(vto, vfrom);
        kunmap_coherent(from);
    } else {
        vfrom = kmap_atomic(from, KM_USER0);
        copy_page(vto, vfrom);
        kunmap_atomic(vfrom, KM_USER0);
    }
    if (((vma->vm_flags & VM_EXEC) && !cpu_has_ic_fills_f_dc) ||
        pages_do_alias((unsigned long)vto, vaddr & PAGE_MASK))
        flush_data_cache_page((unsigned long)vto);
    kunmap_atomic(vto, KM_USER1);
    /* Make sure this page is cleared on other CPU's too before using it */
    smp_wmb();
}

void copy_to_user_page(struct vm_area_struct *vma,
    struct page *page, unsigned long vaddr, void *dst, const void *src,
    unsigned long len)
{
    if (cpu_has_dc_aliases) {
        void *vto = kmap_coherent(page, vaddr) + (vaddr & ~PAGE_MASK);
```

```
        memcpy(vto, src, len);
        kunmap_coherent(page);
    } else
        memcpy(dst, src, len);
    if ((vma->vm_flags & VM_EXEC) && !cpu_has_ic_fills_f_dc)
        flush_cache_page(vma, vaddr, page_to_pfn(page));
}

void copy_from_user_page(struct vm_area_struct *vma,
    struct page *page, unsigned long vaddr, void *dst, const void *src,
    unsigned long len)
{
    if (cpu_has_dc_aliases) {
        void *vfrom =
            kmap_coherent(page, vaddr) + (vaddr & ~PAGE_MASK);
        memcpy(dst, vfrom, len);
        kunmap_coherent(page);
    } else
        memcpy(dst, src, len);
}
```

改进的做法有点流氓 :) 引入了 cpu_use_kmap_coherent，让他们都走 else，然后对 else 中的程序块做了额外的 Cache flush 以消除因 Cache Aliases 引起的数据不一致问题：

```
--- a/arch/mips/mm/init.c
+++ b/arch/mips/mm/init.c
@@ -207,11 +207,13 @@
    void *vfrom, *vto;

    vto = kmap_atomic(to, KM_USER1);
-   if (cpu_has_dc_aliases) {
+   if (cpu_has_dc_aliases && cpu_use_kmap_coherent) {
        vfrom = kmap_coherent(from, vaddr);
        copy_page(vto, vfrom);
        kunmap_coherent(from);
    } else {
        vfrom = kmap_atomic(from, KM_USER0);
+       if(pages_do_alias((unsigned long)vfrom, vaddr & PAGE_MASK))
+           flush_cache_page(vma, vaddr, page_to_pfn(from));
        copy_page(vto, vfrom);
        kunmap_atomic(vfrom, KM_USER0);
@@ -230,13 +231,14 @@
    struct page *page, unsigned long vaddr, void *dst, const void *src,
    unsigned long len)
 {
```

```
-   if (cpu_has_dc_aliases) {
+   if (cpu_has_dc_aliases && cpu_use_kmap_coherent) {
        void *vto = kmap_coherent(page, vaddr) + (vaddr & ~PAGE_MASK);
        memcpy(vto, src, len);
        kunmap_coherent(page);
    } else
        memcpy(dst, src, len);
-   if ((vma->vm_flags & VM_EXEC) && !cpu_has_ic_fills_f_dc)
+   if ((vma->vm_flags & VM_EXEC) && !cpu_has_ic_fills_f_dc ||
+       pages_do_alias((unsigned long)dst, vaddr & PAGE_MASK))
        flush_cache_page(vma, vaddr, page_to_pfn(page));
 }

@@ -246,13 +248,16 @@
    struct page *page, unsigned long vaddr, void *dst, const void *src,
    unsigned long len)
 {
-   if (cpu_has_dc_aliases) {
+   if (cpu_has_dc_aliases && cpu_use_kmap_coherent) {
        void *vfrom =
            kmap_coherent(page, vaddr) + (vaddr & ~PAGE_MASK);
        memcpy(dst, vfrom, len);
        kunmap_coherent(page);
-   } else
+   } else {
+       if(pages_do_alias((unsigned long)src, vaddr & PAGE_MASK))
+           flush_cache_page(vma, vaddr, page_to_pfn(page));
        memcpy(dst, src, len);
+   }
 }

 EXPORT_SYMBOL(copy_from_user_page);

--- a/include/asm-mips/mach-bcm56218/cpu-feature-overrides.h
+++ b/include/asm-mips/mach-bcm56218/cpu-feature-overrides.h
@@ -11,6 +11,6 @@
 #define __ASM_MACH_BCM56218_CPU_FEATURE_OVERRIDES_H

 #define cpu_has_llsc       1
-#define cpu_has_dc_aliases 0
+#define cpu_use_kmap_coherent  0
```

关于如何如此 flush 的原因，可参考 Cache Aliases 节的描述。

5.5.4.2 激进的

此后轻松的盯了 kmap_coherent() 近一天，最后浮出水面的石头几乎让人崩溃，怀疑了很多不该怀疑的，可从来没有怀疑过 broadcom 的这个 MIPS 实现，它的 KSEG2 高端有一段地址空间，居然不经过 TLB，而 kmap_coherent 恰恰是用位于这个区间的地址来做 fixed map 的，经测试在清空所有 TLB 项的情形下，访问这段区域不会出现异常，且始终返回 0 值，感觉就像固定映射到了一个外设的内部 RAM（我称之为 black hole），要命的是 broadcom 没有任何的文档描述。下面这张图是以页为步进单位，多次访问得出：

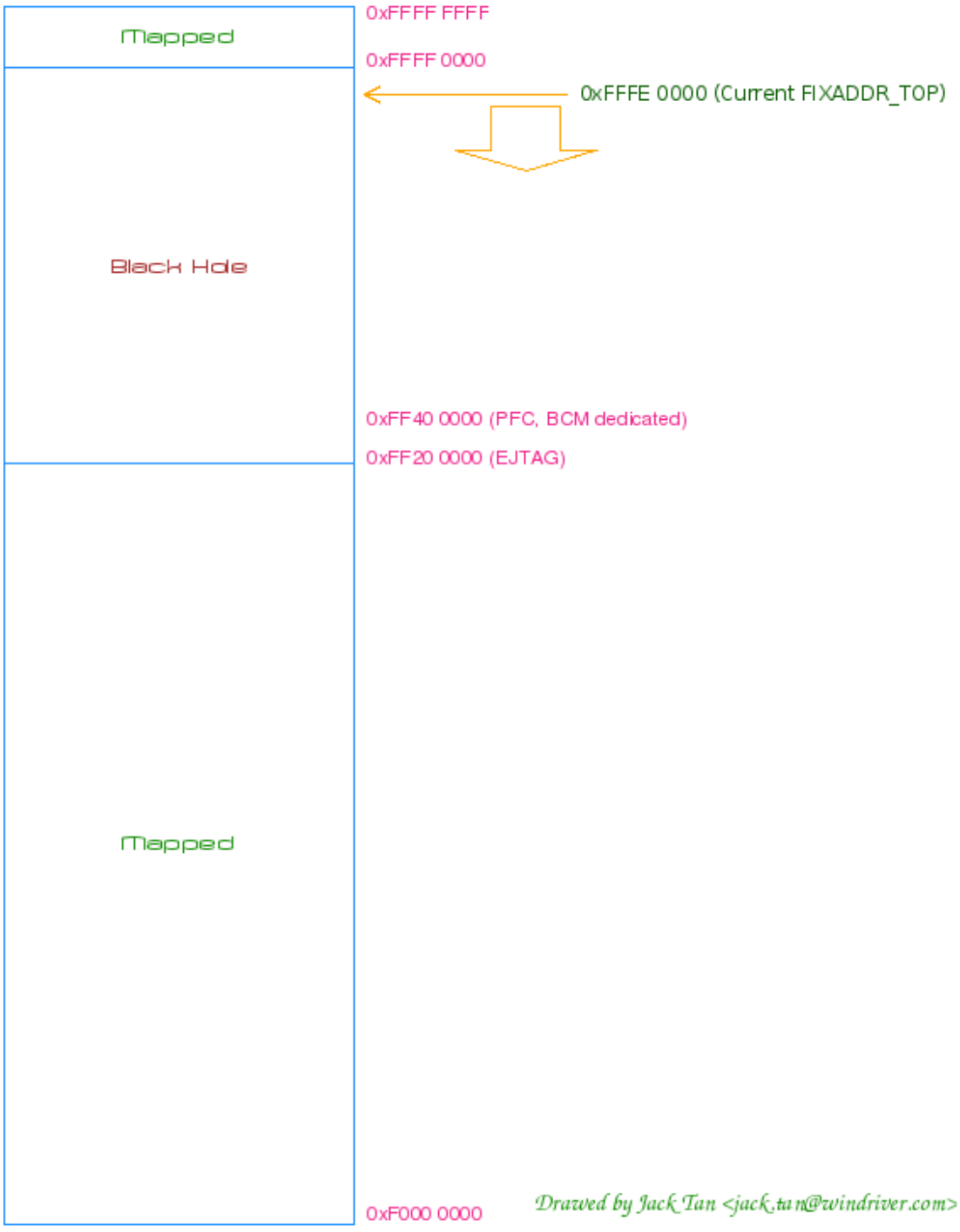


Figure 9: BCM3300's Kseg2 high address space

FIXADDR_TOP 往下 8 页左右，即是 kmap_coherent 用来作临时固定映射的。知道的这一点，修正的手段也就很容易了，只要把 FIXADDR_TOP 往下移到 Black Hole 区以下即可，最终的修正比较简洁：

```
--- a/include/asm-mips/fixmap.h.orig
+++ b/include/asm-mips/fixmap.h
@@ -79,6 +79,7 @@
 */
-#if defined(CONFIG_CPU_TX39XX) || defined(CONFIG_CPU_TX49XX)
+#if defined(CONFIG_CPU_TX39XX) || defined(CONFIG_CPU_TX49XX) || defined (CONFIG_BCM5621X)
#define FIXADDR_TOP    ((unsigned long)(long)(int)(0xff000000 - 0x20000))
#else
#define FIXADDR_TOP    ((unsigned long)(long)(int)0xfffe0000)
#endif
```

6. Cache initialized on MIPS

上电后 Cache 内部的状态未知，尤其是用于控制的 Tag/Status 域的状态都是未知的，这个需要在使用 Cache 前将其初始化为 0。x86、PowerPC 的实现都是用硬件来完成，MIPS 的实现不一样，这个过程需要软件来做（设计理念不一样）。

下面给出 Yamon 的实现。Yamon 是 MIPS 公司开发、维护的 Bootloader，其对符合 MIPS32 和 MIPS64 标准的 MIPS core 的实现支持较好，对不兼容此标准的一些 MIPS 实现，支持不太好。

MIPS 上电后，使用 Cache 前，必须Cache 初始化的代码必须放在 uncache 区 (KSEG1)。初始化的基本操作就是将 Cache 的 Tag 置为 0。

6.1 Yamon 对 I-Cache 初始化

a0 ----> cache size
a1 ----> line size

```
LEAF( sys_init_iCache )

    /* Clear TagLo and  TagHi.
     * Note : For 20Kc and 25Kf, we actually need to clear
     *       ITagLo and ITagHi, but they have the same
     *       register numbers (and select fields) as TagLo, TagHi.
     */
    MTC0(zero, C0_TagLo)      # mtc0 $0, c0_TagLo, 将 CP0 之 TagLo 置为 0
    MTC0(zero, C0_TagHi)      # mtc0 $0, c0_TagHi, 将 CP0 之 TagHi 置为 0

    beq   a0, zero, 2f              /* just in case Cache size = 0 */

    /* Calc an address that will correspond to the first Cache line */
    li       a2, KSEG0BASE       # KSEG0BASE == 0x8000 0000

    /* Calc an address that will correspond to the last Cache line  */
    addu       a3, a2, a0
    subu    a3, a1

    /* Loop through all lines, invalidating each of them */
1:
```

```
        Cache       ICache_INDEX_STORE_TAG, 0(a2)       /* clear tag */
#ifdef _ARCH_RBTX4939
        Cache       ICache_INDEX_STORE_TAG, 1(a2)       /* clear tag */
        Cache       ICache_INDEX_STORE_TAG, 2(a2)       /* clear tag */
        Cache       ICache_INDEX_STORE_TAG, 3(a2)       /* clear tag */
#endif

        bne       a2, a3, 1b
        addu       a2, a1
2:
        jr       ra
        nop

END( sys_init_iCache )
```

6.2 Yamon 对 D-Cache 初始化

a0 ----> cache size
a1 ----> line size
a2 ----> core id

```
LEAF( sys_init_dCache )

        li    a3, MIPS_20Kc
        beq   a3, a2, 1f
        nop
        li    a3, MIPS_25Kf
        beq   a3, a2, 1f
        nop

        MTC0( zero, C0_TagLo )
        MTC0( zero, C0_TagHi )
        b     2f
        nop
1:
        /* 20Kc/25Kf : Use DTagLo and DTagHi for data Cache */          #  DTagLo 与 TagLo 具有相同的 CP0 寄存器号，
                                                                            mtc0/mfc0 操作时，使用选择子区分：
                                                                            mtc0 $0, $28, 2
        MTC0_SEL_OPCODE( R_zero, R_C0_DTagLo, R_C0_SelDTagLo )      # R_C0_SelDTagLo 为选择子，整数，文档有定义
        MTC0_SEL_OPCODE( R_zero, R_C0_DTagHi, R_C0_SelDTagHi )      # R_C0_SelDTagHi 为选择子
2:
        beq   a0, zero, 2f                  /* just in case Cache size = 0 */

        /* Calc an address that will correspond to the first Cache line */
        li       a2, KSEG0BASE
```

```
        /* Calc an address that will correspond to the last Cache line  */
        addu        a3, a2, a0
        subu    a3, a1

        /* Loop through all lines, invalidating each of them */
1:

        Cache       DCache_INDEX_STORE_TAG, 0(a2)       /* clear tag */
#ifdef _ARCH_RBTX4939
        Cache       DCache_INDEX_STORE_TAG, 1(a2)       /* clear tag */
        Cache       DCache_INDEX_STORE_TAG, 2(a2)       /* clear tag */
        Cache       DCache_INDEX_STORE_TAG, 3(a2)       /* clear tag */
#endif

        bne       a2, a3, 1b
        addu       a2, a1
2:
        jr       ra
        nop

END( sys_init_dCache )
```

6.3 4KE Cache 初始化参考实现

以下代码在实际系统上经过测试，可以直接使用。MIPS32 的实现应该皆可使用。

```
.macro  enable_cached
    /* set config */
    mfc0    k1, CP0_CONFIG
    srl     k1, 3
    sll     k1, 3
    ori     k1, 0x3
    mtc0    k1, CP0_CONFIG
    ssnop
    ssnop
    ssnop
    sll     zero, 3
.endm

.macro  get_icache_size
    /* fetch cp0_config1 value */
    mfc0    k0, CP0_CONFIG, 1

    /* get icache bytes per way */
    move    t0, k0
    srl     t0, t0, 22
```

```
    andi    t0, 0x7
    addu    t0, 0xa
    xor     s0, s0
    addu    s0, s0, 1
    sll     s0, t0

    /* get icache ways */
    move    s1, k0
    srl     s1, s1, 16
    andi    s1, 0x7
    addu    s1, 1

    /* get the icache size */
    mul     s0, s1
    mflo    s0
.endm

.macro  get_dcache_size

    /* get dcache bytes per way */
    move    t0, k0
    srl     t0, t0, 13
    andi    t0, 0x7
    addu    t0, 0xa
    li      s0, 1
    sll     s0, t0

    /* get dcache ways */
    move    s1, k0
    srl     s1, s1, 7
    andi    s1, 0x7
    addu    s1, 1

    /* get the dcache size */
    mul     s0, s1
    mflo    s0

.endm

.macro  setup_cache

    /* clear taglo/taghi */
    mtc0    zero, CP0_TAGLO
    ssnop
    ssnop
    ssnop
    sll     zero, 3
```

```
    /* setup icache */
    get_icache_size
    lui     s2, 0x8000
    addu    s3, s2, s0
21:
    cache   Index_Store_Tag_I, 0(s2)
    addiu   s2, 16
    ble     s2, s3, 21b
    nop

    /* setup dcache */
    get_dcache_size
    lui     s2, 0x8000
    addu    s3, s2, s0
22:
    cache   Index_Store_Tag_D, 0(s2)
    addiu   s2, 16
    ble     s2, s3, 22b
    nop

    enable_cached
.endm
```

7. Write Buffer

7.1 概述

现代的处理器往往引入 Write/Store Buffer 来提升存储系统的性能。Write Buffer 的位置位于 L1 D-Cache 与 下一级存储器之间，下图是 Open SPARC T1 的框图：

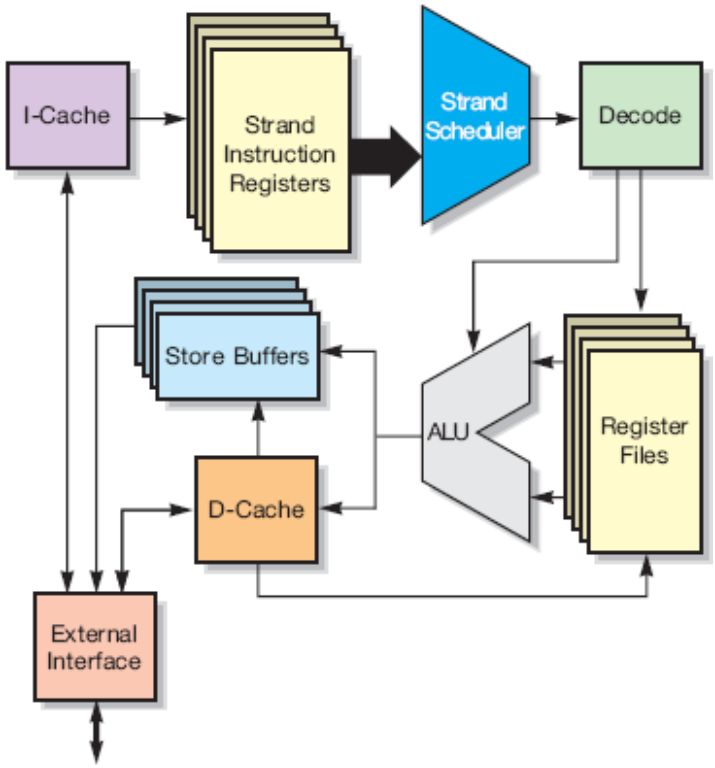


FIGURE 2-1 SPARC Core Block Diagram

引入 Write Buffer，其优点在于当 DCache 的脏行被替换，数据写回时，此时 Cache 无需等待可以继续工作，即：其认为数据回写入 Write Buffer 即认为此回写操作已完成，无需等待下一级存储器的缓慢响应。

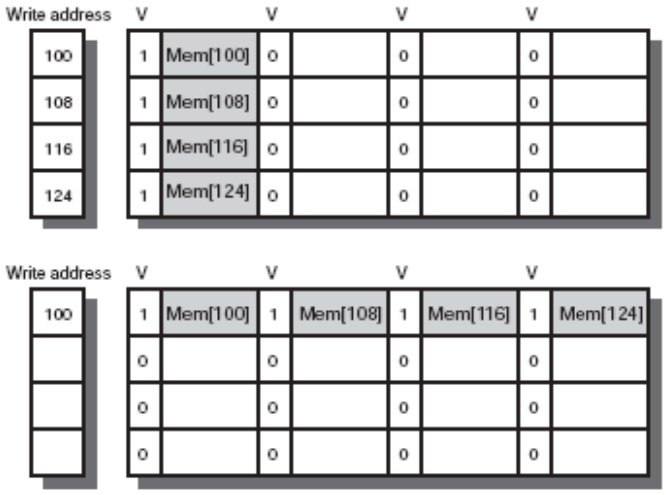
Write-through 之 Cache 亦复如是。

对不经 Cache 的 store 操作，其亦认为数据回写入 Write Buffer 即认为此次写操作已完成，很多手册上将这个技术称为 "Uncached accelerated"。

7.2 结构和工作方式

与 Cache 相似，其每个 Entry 亦分地址和数据，数据区的容量一般于 DCache 之行大小相同。当 Cache 数据写回时，WB 即为其分配一个 Entry，将地址和数据写入，若 WB 满，则 Cache 亦需要等待。

有一些牛B 的实现还引入了“写合并” (Write merging)，即：后写的数据发现 WR 里有一个有效的 Entry，其地址恰好落在其范围内，则将后入的数据与其合并：



上面的 WB 没用写合并，下面的 WB 是使用写合并的。其皆有 4 个 Entry，每个 Entry 能存储 64 字节，每 8 字节具有一个有效位 (V)，表示这个单元是否被占用。对后者，4 次（每次 8 字节）写被合并到一个 Entry 里。而对前者，每次 8 字节，连续 4 次的写，则将其所有 Entry 占用了，如此时再来一个写，则尽管每行尚有 3/4 的空间未用，但还是要等待。

此外每次写多个字常常快于一个字一个字的写，其更能充分利用总线带宽。

7.3 控制接口

因为 Write Buffer 亦会“缓存”通过 uncached 地址写出的数据，则写向设备 I/O 寄存器的数据不会立即写到设备内，因此若软件需要 I/O 操作立时生效，则需主动刷新 Write Buffer。

为了支持这个操作，MIPS ISA 引入了 'sync' 指令，处理器对这个的指令的响应如下：

> 该指令后的指令停止进入流水线（延期）直到该指令前的所有指令被提交（完成），所有 Load/Store 指令对内存的访问请求在该指令发射前全部完成。

宏观上看，其实际上就是一 Memory barrier

有些实现引入了更精细的控制，比如 Cavium 的 MIPS 多核实现，其将 Load/Store 操作详细的区分为 L2/DRAM Load/Store，I/O Load/Store，IOBDMA Load/Store，并在指令集引入 syncs, syncw, syncws 和 syniobdma，针对不同的组合对其进行刷新。

软件对 Write Buffer 的操作仅此而已，其不能精确读取 WB 的内容。

Appendix: Debug Tips

A.1 SysRq Tips

A.1.1 On telnet console

1. `Ctrl + ]`
2. `telnet> send brk`
3. `t` `# show tasks`

A.1.2 On minicom console

1. `Ctrl + A, F` `# send brk`
2. `p` `# show registers`

A.1.3 Add a new SysRq mapping

In drivers/char/sysrq.c:

```
static void sysrq_handle_dumptlb(int key, struct tty_struct *tty)
{
    dump_tlb_all();                         # implemented in arch/mips/lib-32/dump_tlb.c
}

static struct sysrq_key_op sysrq_dumptlb_op = {
    .handler     = sysrq_handle_dumptlb,
    .help_msg    = "dumpTLB",
    .action_msg = "Dump TLB",
    .enable_mask     = SYSRQ_ENABLE_DUMP,
};
```

Then add the pointer of sysrq_dumptlb_op in sysrq_key_table:

```
static struct sysrq_key_op *sysrq_key_table[36] = {
    ......
    &sysrq_reboot_op,       /* b */
    &sysrq_crashdump_op,        /* c & ibm_emac driver debug */
```

```
    &sysrq_showlocks_op,        /* d */
    &sysrq_term_op,         /* e */
    &sysrq_moom_op,         /* f */
    /* g: May be registered by ppc for kgdb */
    &sysrq_dumptlb_op,              /* g */
    NULL,               /* h */
    &sysrq_kill_op,         /* i */
    NULL,   /* j */
    &sysrq_SAK_op,          /* k */
    ......
};
```

Then you can use the 'g' to trigger off the sysrq_handle_dumptlb

A.2 Dump TLB

dump_tlb_all() is implemented in [arch/mips/lib-32/dump_tlb.c]:

```
void dump_tlb(int first, int last)
{
    unsigned int pagemask, c0, c1, asid;
    unsigned long long entrylo0, entrylo1;
    unsigned long entryhi;
    int i;

    asid = read_c0_entryhi() & 0xff;

    printk("\n");
    for (i = first; i <= last; i++) {
        write_c0_index(i);
        BARRIER();
        tlb_read();                                  # 读 TLB 的第 i + 1 项，数据写入 entryhi/lo0/lo1
        BARRIER();
        pagemask = read_c0_pagemask();               # 读取 pagemask 的值
        entryhi = read_c0_entryhi();                 # 取刚刚读出的 TLB 项之 entryhi 部分
        entrylo0 = read_c0_entrylo0();               # 取第 i + 1 项之 entrylo0 部分（偶数页）
        entrylo1 = read_c0_entrylo1();               # 取第 i + 1 项之 entrylo1 部分（奇数页）

        /* Unused entries have a virtual address in KSEG0.  */
        if ((entryhi & 0xf0000000) != 0x80000000
            && (entryhi & 0xff) == asid) {           # 只打印属于当前进程的 TLB 项，asid 值为 tlb read
                                                         前预先保存
            /*
             * Only print entries in use
             */
            printk("Index: %2d pgmask=%s ", i, msk2str(pagemask));

            c0 = (entrylo0 >> 3) & 7;
            c1 = (entrylo1 >> 3) & 7;
                                                     # 下面的代码，对读出的数据稍作修饰，增加点可读性
            printk("va=%08lx asid=%02lx\n",
                   (entryhi & 0xffffe000), (entryhi & 0xff));
            printk("\t\t\t[pa=%08Lx c=%d d=%d v=%d g=%Ld]\n",
                   (entrylo0 << 6) & PAGE_MASK, c0,
                   (entrylo0 & 4) ? 1 : 0,
                   (entrylo0 & 2) ? 1 : 0, (entrylo0 & 1));
            printk("\t\t\t[pa=%08Lx c=%d d=%d v=%d g=%Ld]\n",
                   (entrylo1 << 6) & PAGE_MASK, c1,
                   (entrylo1 & 4) ? 1 : 0,
```

```
                (entrylo1 & 2) ? 1 : 0, (entrylo1 & 1));
            printk("\n");
        }
    }

    write_c0_entryhi(asid);
}

void dump_tlb_all(void)
{
    dump_tlb(0, current_cpu_data.tlbsize - 1);
}
```

为了能看到 TLB 的全貌，常常将上面的红色代码移去，让其打印出所有 TLB 的项。

下面是 MIPS 4KEc （16 项的 TLB）上的一个参考输出：

```
Index:  0 pgmask=4kb va=0040c000 asid=58
            [pa=017f4000 c=3 d=0 v=1 g=0]       <---------- vaddr = 0x0040 c000
            [pa=00000000 c=0 d=0 v=0 g=0]       <----------  奇数页没用

Index:  1 pgmask=4kb va=7fe2c000 asid=58
            [pa=09c93000 c=3 d=1 v=1 g=0]
            [pa=00000000 c=0 d=0 v=0 g=0]

Index:  2 pgmask=4kb va=00432000 asid=58
            [pa=01711000 c=3 d=1 v=1 g=0]       <---------- vaddr = 0x0043 2000   偶数页
            [pa=0170b000 c=3 d=1 v=1 g=0]       <---------- vaddr = 0x0043 3000   奇数页

Index:  3 pgmask=4kb va=2aaa8000 asid=58
            [pa=0170f000 c=3 d=1 v=1 g=0]
            [pa=01710000 c=3 d=1 v=1 g=0]

Index:  4 pgmask=4kb va=0041c000 asid=58
            [pa=01727000 c=3 d=0 v=1 g=0]
            [pa=00000000 c=0 d=0 v=0 g=0]

Index:  5 pgmask=4kb va=2c3c2000 asid=58
            [pa=01799000 c=3 d=0 v=1 g=0]
            [pa=0179a000 c=3 d=0 v=1 g=0]

Index:  6 pgmask=4kb va=2c320000 asid=58
            [pa=017db000 c=3 d=0 v=1 g=0]
            [pa=017dc000 c=3 d=0 v=1 g=0]

Index:  7 pgmask=4kb va=00444000 asid=58
            [pa=0172f000 c=3 d=0 v=1 g=0]
            [pa=09c9a000 c=3 d=1 v=1 g=0]

Index:  8 pgmask=4kb va=00420000 asid=58
            [pa=0172b000 c=3 d=0 v=1 g=0]
            [pa=0172c000 c=3 d=0 v=1 g=0]

Index:  9 pgmask=4kb va=00434000 asid=58
            [pa=09c90000 c=3 d=1 v=1 g=0]
            [pa=09c98000 c=3 d=1 v=1 g=0]

Index: 10 pgmask=4kb va=00414000 asid=58
            [pa=0171f000 c=3 d=0 v=1 g=0]
            [pa=01720000 c=3 d=0 v=1 g=0]
```

```
Index: 11 pgmask=4kb va=2c3b2000 asid=58
        [pa=09c6a000 c=3 d=0 v=1 g=0]
        [pa=09c6b000 c=3 d=0 v=1 g=0]

Index: 12 pgmask=4kb va=0040a000 asid=58
        [pa=00000000 c=0 d=0 v=0 g=0]
        [pa=017f3000 c=3 d=0 v=1 g=0]

Index: 13 pgmask=4kb va=2c368000 asid=58
        [pa=017bb000 c=3 d=0 v=1 g=0]
        [pa=017bc000 c=3 d=0 v=1 g=0]

Index: 14 pgmask=4kb va=00412000 asid=58
        [pa=00000000 c=0 d=0 v=0 g=0]
        [pa=0171e000 c=3 d=0 v=1 g=0]

Index: 15 pgmask=4kb va=00418000 asid=58
        [pa=01723000 c=3 d=0 v=1 g=0]
        [pa=01724000 c=3 d=0 v=1 g=0]
```

A.3 Testing KSEG2

Following functions is to investigate the Kseg2 high address space. If the address is mapped, accessing the address will trigger off an Oops like this: " Unable to handle kernel paging request at virtual address f4000000, epc == xxx ..." (There is no TLB entry for this address and no page table entry in current process's page table)

```
static void dump_mem(unsigned long addr, int len)
{
    unsigned long start, end;
    start = addr & 0xfffffffc;
    end = start + len;

    for(; start < end; start += 32)
    {
        printk(KERN_ERR "0x%08x:\t%08x %08x %08x %08x %08x %08x %08x %08x\n",
                (u32)start, *(u32 *)start, *(u32 *)(start + 4),
                *(u32 *)(start + 8), *(u32 *)(start + 12),
                *(u32 *)(start + 16), *(u32 *)(start + 20),
                *(u32 *)(start + 24), *(u32 *)(start + 28)
            );
    }
    return;
}

static void sysrq_handle_test_kseg2(int key, struct tty_struct *tty)
{
    u32 start = 0xff200000, end = 0xfffff000;
    for( ; start <= end; start += 0x1000)
    {
        dump_tlb_all();
        dump_mem(start, 32*10);
    }
}

static struct sysrq_key_op sysrq_test_kseg2_op = {
    .handler    = sysrq_handle_test_kseg2,
    .help_msg   = "showKseg2",
    .action_msg = "Investigate Kseg2",
    .enable_mask    = SYSRQ_ENABLE_DUMP,
};
```

A.4 验证 Cache Aliases

VA_1、VA_2 映射到同一 PA，期间使用 VA_1 更新过数据，后陷入内核态，内核使用 VA_2 访问这块数据，则可用 memcmp((void *)VA1, (void *)VA2, PAGE_SIZE) 比较之，如返回值为 1 则说明 Cache Aliases

A.5 Dump Cache Tag

```
/* dump all tags within set sn */
void dump_dcache_tag(int sn)
{
    const u32 ws_inc = 1UL << current_cpu_data.dcache.waybit;
    const u32 ws_end = current_cpu_data.dcache.ways << current_cpu_data.dcache.waybit;
    u32 addr_cool = 0x80000000 + (sn << __ffs(current_cpu_data.dcache.linesz));

    int i = 0;
    u32 taglo = 0;

    /* dump all line's tag in set sn*/
    i = 0;
    printk("<0> Set %d: ", sn);
    do {
        asm (
            "addu   $8, %2, %1\n\t"
            "cache  0x05, 0($8)\n\t"
            "mfc0   %0, $28\n\t"
            : "=r"(taglo)
            : "r"(i), "r"(addr_cool)
            : "$8"
        );

        printk("0x%08x  ", taglo);
        i += ws_inc;

    } while (i < ws_end);
    printk("\n");

}

void dump_dcache_tag_all()
{
    int sets = current_cpu_data.dcache.sets;
    int i = 0;

    for (; i < sets; i++)
        dump_dcache_tag(i);
}

static void sysrq_handle_dump_dc_tag(int key, struct tty_struct *tty)
{
    dump_dcache_tag_all();
}
```

```
static struct sysrq_key_op sysrq_dump_dc_tag_op = {
    .handler    = sysrq_handle_dump_dc_tag,
    .help_msg   = "dumpDCacheTag",
    .action_msg = "Dump DCache Tag",
    .enable_mask    = SYSRQ_ENABLE_DUMP,
};
```

Example Output:

```
telnet> send brk
SysRq : Dump DCache Tag
 Set 0: 0x002b4080  0x003260c0
 Set 1: 0x002b40c0  0x003260c0
 Set 2: 0x002fd080  0x0026d080
 Set 3: 0x0026d080  0x002b40c0
 Set 4: 0x002fd080  0x016df080
 Set 5: 0x016df080  0x002ba0c0
 Set 6: 0x002f90c0  0x09c9a080
 Set 7: 0x002f90c0  0x01473080
 Set 8: 0x0147c080  0x002f90c0
 Set 9: 0x09c9a0c0  0x002f90c0
 Set 10: 0x002f90c0  0x09c9a080
 Set 11: 0x09c9a080  0x002f90c0
 Set 12: 0x0030b080  0x002f90c0
......
......
 Set 251: 0x002facc0  0x002f9cc0
 Set 252: 0x002facc0  0x002f9cc0
 Set 253: 0x002f9cc0  0x002facc0
 Set 254: 0x0026cc80  0x002facc0
 Set 255: 0x002f9cc0  0x002facc0
```

A.6 Testing DCache Tag Multiple Hit

以下的函数人为地将 Cache 的某个组的所有行的 Tag 强制写为相同的值，然后访问一个索引到这个组的地址，人造一个多命中的情况 ;)

```
static void sysrq_handle_dc_multihit(int key, struct tty_struct *tty)
{
    const u32 addr_cool = 0x80400000;
    const u32 ws_inc = 1UL << current_cpu_data.dcache.waybit;
    const u32 ws_end = current_cpu_data.dcache.ways << current_cpu_data.dcache.waybit;

    int i = 0;
    u32 taglo = 0;

    /* dump all line's tag in set 0*/
    printk("<0>----------------------- Before invalidated ---------------------- \n");
    do {
        asm (
            "lui    $3, 0x8040\n\t"
            "addu    $8, $3, %1\n\t"
            "cache    0x05, 0($8)\n\t"
            "mfc0    %0, $28, 2\n\t"
            : "=r"(taglo)
            : "r"(i)
            : "$8", "$3"
        );

        printk("<0> taglo_%d = 0x%08x\n", i/ws_inc, taglo);
        i += ws_inc;

    } while (i < ws_end);

    printk("<0> -------------- Invalidated all lines in set 0 ------------------ \n");

    asm ("mtc0    $0, $28, 2\n\t");    // set TagLo to zero
    /* clear all line's tag in set 0 */
    do {
        asm (
            "lui    $8, 0x8040\n\t"
            "mtc0    $0, $28, 2\n\t"
            "addu    $3, $8, %0\n\t"
            "cache 0x01, 0($3)\n\t"        /* invalidate write-back a line */
            "cache 0x09, 0($3)\n\t"        /* clear this line's tag to zero */
            :
```

```
            : "r"(i)
            : "$8", "$3"
        );

        i += ws_inc;

    } while (i < ws_end);

    /* dump all line's tag in set 0*/
    i = 0;
    do {
        asm (
            "lui    $3, 0x8040\n\t"
            "addu    $8, $3, %1\n\t"
            "cache    0x05, 0($8)\n\t"
            "mfc0    %0, $28, 2\n\t"
            : "=r"(taglo)
            : "r"(i)
            : "$8", "$3"
        );

        printk("<0> taglo_%d = 0x%08x\n", i/ws_inc, taglo);
        i += ws_inc;

    } while (i < ws_end);

    printk("<0> ----------------- Cache a line into set 0 ---------------------- \n");

    /* cache a line into set 0 */
    u32 tmp = *(u32 *) addr_cool;
    printk("<0> *(u32 *)0x%08x = 0x%08x\n", addr_cool, tmp);

    /* dump all line's tag in set 0*/
    i = 0;
    do {
        asm (
            "lui    $3, 0x8040\n\t"
            "addu    $8, $3, %1\n\t"
            "cache    0x05, 0($8)\n\t"
            "mfc0    %0, $28, 2\n\t"
            : "=r"(taglo)
            : "r"(i)
            : "$8", "$3"
        );

        printk("<0> taglo_%d = 0x%08x\n", i/ws_inc, taglo);
        i += ws_inc;
```

```
} while (i < ws_end);

printk("<0> -- Get a validated line, read its Tag, write it to all lines -- \n");

/* find a validated line, read its tag */
i = 0;
do {
   asm (
      "lui    $3, 0x8040\n\t"
      "addu    $8, $3, %1\n\t"
      "cache    0x05, 0($8)\n\t"
      "mfc0    %0, $28, 2\n\t"
      : "=r"(taglo)
      : "r"(i)
      : "$8", "$3"
   );

   printk("<0> taglo = 0x%08x\n", taglo);
   i += ws_inc;

} while ((((taglo >> 6) & 3) == 0) && i < ws_end);    /* cache line is invalide */

/* set all other line's tag to the validated line's tag */
do {
   asm (
      "lui    $8, 0x8040\n\t"
      "addu    $3, $8, %0\n\t"
      "cache 0x09, 0($3)\n\t"        /* store tag */
      :
      : "r" (i)
      : "$8", "$3"
   );

   i += ws_inc;

} while (i < ws_end);

/* dump all line's tag in set 0*/
i = 0;
do {
   asm (
      "lui    $3, 0x8040\n\t"
      "addu    $8, $3, %1\n\t"
      "cache    0x05, 0($8)\n\t"
      "mfc0    %0, $28, 2\n\t"
      : "=r"(taglo)
      : "r"(i)
      : "$8", "$3"
```

```
        );

        printk("<0> taglo_%d = 0x%08x\n", i/ws_inc, taglo);
        i += ws_inc;

    } while (i < ws_end);

    /* multi-hit cache line */
    printk("<0> ---------- Multi-Hit cache line ---------- \n");
    tmp = *(u32 *) addr_cool;
    printk("<0> *(u32 *)0x%08x = 0x%08x\n\n", addr_cool, tmp);

    printk("<0> ---------- After Multi-Hit ---------- \n");
    /* dump all line's tag in set 0*/
    i = 0;
    do {
        asm (
            "lui    $3, 0x8040\n\t"
            "addu    $8, $3, %1\n\t"
            "cache    0x05, 0($8)\n\t"
            "mfc0    %0, $28, 2\n\t"
            : "=r"(taglo)
            : "r"(i)
            : "$8", "$3"
        );

        printk("<0> taglo_%d = 0x%08x\n", i/ws_inc, taglo);
        i += ws_inc;

    } while (i < ws_end);
    printk("<0> ---------- \n\n");
}

static struct sysrq_key_op sysrq_dc_multihit_op = {
    .handler    = sysrq_handle_dc_multihit,
    .help_msg    = "DCache Multi-Hit Testing",
    .action_msg    = "DCacheMultiHitTest",
    .enable_mask    = SYSRQ_ENABLE_DUMP,
};
```

Cache 多命中的情形，MIPS 上的结果往往是取其中的一个：

4KEc 平台，8KB, 2-way, linesize 16 bytes L1 DCache:

```
telnet> send brk
SysRq : DCacheMultiHitTest
---------- Before invalidated ----------
```

```
taglo_0 = 0x003260c0
taglo_1 = 0x002b4080

-------------- Invalidated all lines in set 0 ------------------
taglo_0 = 0x00000000
taglo_1 = 0x002b4080    -----> 不知为何总是残留一行，Invalidate 对其似乎不起作用，不过幸运的是后面的写成功了

----------------- Cache a line into set 0 ----------------------
*(u32 *)0x80000000 = 0x3c1b802f
taglo_0 = 0x00000080
taglo_1 = 0x002b4080

-- Get a validated line, read its Tag, write it to all lines --
taglo = 0x00000080
taglo_0 = 0x00000080
taglo_1 = 0x00000080

---------------------- Multi-Hit cache line ---------------------
*(u32 *)0x80000000 = 0x802b6188

------------------------ After Multi-Hit ------------------------
taglo_0 = 0x002b4080
taglo_1 = 0x00000080
-----------------------------------------------------------------
```

对 Cache 组内多命中的情形，比较可靠的设计是一旦出现多命中，则将组内所有行直接置无效，重新从内存里取数据。

Reference:

[1] Computer Organization and Architecture 6th Edtion
[2] Computer Architecture A Quantitative Approach 4th Edtion
[3] See MIPS Run 2nd Edtion
[4] http://www.linux-mips.org/wiki/Caches
[5] 4KE Software User Manual
[6] 24KE Processor Core Family Software User’s Manual
[7] 20Kc Processor Core User’s Manual
[8] Loongson2E/2F User Manual
[9] SB1 User Manual
[10] 64-Bit TX System RISC TX49/H2, TX49/H3, TX49/H4 Core Architecture Rev1.0
[11] VR5500A User's Manual
[12] Au1100/Au1200 Processor Data Book
[13] MIPS R4000 Microprocessor User’s Manual 2nd Edition
[14] ARM Architecture Reference Manual 2nd Edtion
[15] Linux PowerPC 详解
[16] PowerPC e500 Core Family Reference Manual
[17] OpenSPARC T1 Microarchitecture Specification