第3章

Cache缓存和数据一致性

与其他处理器一样，基于C66x内核的处理器也存在内核处理能力与存储器容量不匹配的问题。越靠近内核，存储器的通信带宽要求越高，但容量也就越小；越远离内核，处理器容量越大，但带宽也就越小。C66x处理器内核使用寄存器，其用到的存储器从内到外依次是L1（L1P和L1D）、L2 SRAM、MSM SRAM（L3）、DDR3。如前所述，L1和L2位于C66x内核中，L3位于处理器中（C66x内核外面），DDR3位于处理器外。

为了缓解处理器内核和外部存储器的矛盾，采用了Cache机制来实现外部数据在靠近处理器内核的存储器中保留一份拷贝，处理器内核经常与该数据拷贝交互数据，而不是直接和外部存储器交互数据。

本章首先介绍了为什么使用Cache、Cache存储器结构概览、Cache基础知识，然后对C66x的各个Cache进行了详细介绍，并介绍了使用Cache、数据一致性、片上Debug支持和运行中改变Cache配置等内容，最后介绍了如何优化Cache性能和一些设计建议。

3.1 为什么使用Cache

从DSP应用的角度，拥有一个大容量、快速的片上存储器是非常重要的。然而，处理器性能的提升比存储器发展的步伐更快，导致在内核与存储器速度间出现了一个性能缺口。越靠近内核内存速度越快，但容量也就越小。

Cache的机制是基于位置原理设计的，在讲述Cache机制前先介绍一下位置原理。

所谓位置原理，即假设如果一个存储器位置被引用，则其相同或相邻位置非常可能会很快又被引用。在一段时间内访问存储器的位置被指为时间位置，涉及相邻存储器的位置被指为空间位置。通过利用存储器访问位置原理，Cache缓存减少平均存储器访问时间。

基于位置原理，在一小段时间内，通常一个程序从相同或相邻存储器位置重用数据。

如果数据从一个慢速存储器映射到一个快速Cache存储器，在另一组数据替代前，尽可能经常访问Cache中的数据以提高数据访问效率。

3.2 C64x 和 C66x DSP 之间的 Cache 区别

对于使用过 C64x 内核的程序员来说，C66x 内核 Cache 的概念与 C64x 内核中的相似，但也有很大不同。本节介绍 C66x 内核与 C64x 内核之间的 Cache 区别，主要有以下几点。

1. 存储器尺寸和类型

对于 C66x 器件，每个 L1D 和 L1P 在 Cache 之外实现 SRAM。Cache 的尺寸是用户配置的，可以被设置成 4KB、8KB、16KB 或 32KB。可用的 SRAM 数量是器件相关的，并在器件特性数据手册中明确。而对于 C64x 器件，Cache 被设计成尺寸为固定的 16KB。C66x 器件相对于 C64x 器件，L2 的尺寸增加了。

2. 写缓冲

对于 C66x 器件，写缓冲的宽度增加到 128 位；对于 C64x 器件，宽度是 64 位。

3. Cache 能力

对于 C66x 器件，外部存储地址的 Cache 能力设置（通过 MAR 位）只影响 L1D 和 L2 Cache 缓存；也就是说，到外部存储器地址的程序取指令（program fetch）总是被 Cache 缓存进来。不管 Cache 能力设置状况。这和 C64x 器件上的情况不一样，在 C64x 器件上 Cache 能力设置影响所有 Cache，即 L1P、L1D 和 L2。

对于 C66x 器件，外部存储地址的 Cache 能力控制覆盖整个外部地址空间。对于 C64x 器件，外部存储地址的 Cache 能力控制只覆盖地址空间的一个子集。

4. Snooping 协议

在 C66x 器件上的 Snooping Cache 一致性协议直接发送数据到 L1D Cache 和 DMA。C64x 器件通过 invalid 和 writeback Cache Line 来维持一致性。由于减少了由 invalidate 导致的 Cache 缺失开支，C66x Snooping 机制更加有效。

与 C64x 器件一样，Snoop 协议在 C66x 器件中不维护 L1P Cache 和 L2 SRAM 之间的一致性，程序员负责维护其一致性。

5. Cache 一致性操作

对于 C66x 器件，L2 Cache 一致性操作总是操作在 L1P 和 L1D，即使 L2 Cache 功能被禁用。这与 C64x 器件情况不同，其需要明确调用 L1 一致性操作。

C66x 器件支持一整套的区域和全局（Range and Global）L1D Cache 一致性操作，而 C64x 器件只支持 L1D 区域 invalidate 和 writeback-invalidate 操作。

在 Cache 尺寸上有改变，C66x 器件在初始设置一个新尺寸前，自动 writeback-invalidate Cache。而 C64x 器件需要执行一个完整的 writeback-invalidate 程序（虽然这些是被一部分 CSL 函数处理的）。

对于 C66x 器件，L2 Cache 不包括 L1D 和 L1P，两者不相关。这意味着一个行从 L2 驱逐（evict），不会导致相应的行在 L1P 和 L1D 被驱逐。不相关的优势在于：由于程序取指令导致的 L2 中的行分配不会从 L1D Cache 驱逐数据；由于数据访问导致 L2 中的行分配不会从 L1P 驱逐程序代码，这减少 Cache 缓存缺失的数量。

以下介绍 C66x Cache 存储器结构概览、Cache 基础知识并详细介绍各级 Cache。

3.3 Cache 存储器结构概览

C66x DSP 存储器由内部两级基于 Cache 的存储器和外部存储器组成。L1P 和 L1D 都可以被配置成 SRAM 和 Cache，Cache 最大可以达到 32KB。所有 Cache 和数据路径自动被 Cache 控制器管理，如图 3.1 所示。1 级存储器通过核访问，不需要阻塞。2 级存储器可以被配置，并可被分成 L2 SRAM 和 Cache。外部存储器可以为几 MB 大小。

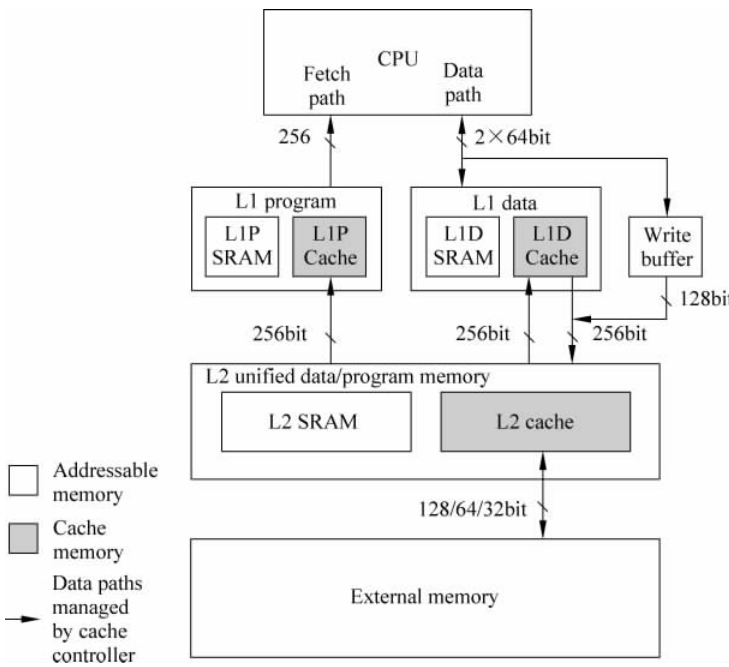

图 3.1 C66x DSP Cache 概览

C6678 器件上电配置如下：

复位后 L1P 被配置为全为 Cache，大小为 32KB。

复位后 L1D 被配置为全为 Cache，大小为 32KB。

复位后 L2 全是 SRAM，Cache 的容量可以被配置为 32KB、64KB、128KB、256KB 或全为 Cache。

访问时间取决于接口和使用的存储器技术。

3.4 Cache 基础知识

通常，Cache 可以分为直接映射 Cache（direct-mapped Caches）和组相联 Cache（set-associative Caches）两种类型。本节介绍 Cache 的一些基本知识。

为了较好理解 Cache 机制，首先介绍几个 Cache 的基本概念。

（1）Cache Line（Cache 行）：Cache 处理的最小单位。Cache Line 的尺寸要比内存存取

的数据尺寸要大，一个行的大小为一个行尺寸(Line Size)。例如，C66x 内核可以访问单个字节，而 L1P Cache 行尺寸为 32B，L1D Cache 行尺寸为 64B，L2 Cache 行尺寸为 128B。但是，如果发生一次读失效，则 Cache 会将整条 Cache Line 的数据读入。

(2) Line Frame(行帧)：Cache 中用于存储 Cache Line 的位置，包含被 Cache 的数据(1 行)、一个关联的 Tag 地址和这一行的状态信息。这一行的状态信息包括是否 Valid(有效)、Dirty(脏)和 LRU 状态。

(3) Set(集)：Line Frame 的一个集合。直接映射的 Cache 中一个 Set 包含一个 Line Frame，n 路组相联的 Cache 每个 Set 包含 n 个 Line Frame。

(4) Tag(标签)：Cache 中被 Cache 的物理地址的高位作为一个 Tag 存储在 Line Frame 中，在决定 Cache 是否命中的时候，Cache 控制器会查询 Tag。

(5) Valid(有效)：当 Cache 中的一个 Line Frame 保存了从下一级存储器取的数据，那么这个 Line Frame 的状态就是 Valid 的，否则，这个 Line Frame 的状态就是无效的(Valid = 0)。

(6) Invalidate(失效)：是将 Cache 中标记为 Valid 的 Line Frame 状态标记为无效的过程，受影响的 Cache Line 内容被废弃。为了维持数据一致性，与 writeback 组合成 writeback-invalidate，先将标记为 Dirty 的行写回到保存有这个地址的下一级存储器，再标记该行为无效状态。

(7) Dirty(脏)和 Clean(干净)：当一个 Cache Line 是 Valid 并包含更新后的数据，但还未更新到下一层更低的内存，则在 Line Frame 的 Dirty 位标志该 Cache Line 为脏的。一个 Valid 的 Cache Line 与下一层更低的内存一致，则 Line Frame 的 Dirty 位标志该 Cache Line 是 Clean 的(Dirty = 0)。

(8) Hit(命中)和 Miss(缺失)：当请求的内存地址的数据在 Cache 中，那么 Tag 匹配并且相应的 Valid 有效，则称为 Hit，数据直接从 Cache 中取给 DSP。相反，如果请求的内存地址的数据不在 Cache 中，Tag 不匹配或相应的 Valid 无效，则称为 Miss。

(9) Victim Buffer(Victim 缓冲)：Cache 中的一条 Cache Line 为新的 Line 腾出空间的过程称为驱逐(Evict)，被驱逐的 Cache Line 被称为 Victim(Line)。当 Victim Line 是 Dirty 的时，为了保持数据一致性，数据必须写回到下一级存储器中。Victim Buffer 保存 Vitim 直到它们被写回到下一级存储器中。

(10) Miss Pipelining(缺失流水)：对连续的缺失进行流水操作，提高对缺失处理的效率，降低阻塞(Stall)周期。

(11) Touch：对一个给定地址的存储器操作，被称为 Touch 那个地址。Touch 也可以指的是读数组元素或存储器地址的其他范围，唯一目的是分配它们到一个特定级别 Cache 中。一个内核中心循环用作 Touch 一个范围的内存，是为了分配它到 Cache 中，经常被称为一个 Touch 循环。Touch 一个数组是软件控制预取数据的一种形式。

3.4.1 直接映射 Cache——L1P Cache

直接映射 Cache 的工作原理可以参照 C66x L1P Cache。任何时候内核访问 L2 SRAM 或外部空间中的指令，指令都被调入 L1P Cache。

1. 读缺失

如果一个程序从地址 0020h 取出，假设那个 Cache 是完全无效的，意味着 Cache 中没有

Cache Line 包含该数据的缓存，这就是一个读缺失。

一个行帧的有效状态被 Valid（V）位指示：Valid 位为 0 表示相应的 Cache Line 是无效的，也就是说，不包含被 Cache 缓存的数据。

当核请求读地址 0020h，Cache 控制器把这个地址分为三块（Tag、Set 和 Offset），如图 3.2 所示。

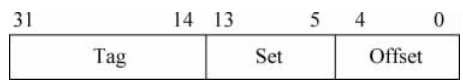

图 3.2 地址分块

Set 部分（bits 13～5）指示地址映射到哪一个 Set（如果是直接映射 Cache，一个 Set 等于一个行帧）。对于地址 0020h，Set 部分检测为 1。然后控制器检测 Tag（bits 31～14）和 Valid 位。由于我们假设 Valid 位为 0，控制器寄存器是一个缺失，也就是说被请求的地址没有包含在 Cache 中。一个缺失也意味着：为了容纳请求地址的行，一个行帧会被分配。然后控制器从存储器取行（0020h～0039h），并存数据到行帧 1。地址的 Tag 部分存储在 Tag RAM 中，Valid 位变成 1 用以指示该 Set 包含有效数据。取出的数据同时也发送给核，访问结束。一个地址的 Tag 部分之所以必须被存储，这是因为当地址 0020h 再次被访问时会更清楚该地址已经被 Cache 缓存。

2. 读命中

Cache 控制器把地址分割为三个部分：Tag、Set 和 Offset，如图 3.2 所示。Set 部分决定地址映射到哪一个 Set；存储的 Tag 部分用于与请求的地址 Tag 部分比较。这个比较是必要的，因为存储器中多个行映射同一 Set，通过 Tag 可以判断出请求的地址是否映射到 Cache 中。如果访问地址 4020h 也映射到同一个 Set，Tag 部分会不同，因而访问会是一个缺失。如果地址 0020h 被访问，Tag 比较为真且 Valid 位为 1，那么控制器寄存器为一个命中，并发送 Cache Line 中的数据到核，该访问结束。

3.4.2 Cache 缺失的类型

在组相联被讨论之前，最好理解不同类型的 Cache 缺失。Cache 最大的目的是减少平均存储器访问时间。从存储器到 Cache 取一个行帧的数据，对于每个缺失，都会有损失。因而，对于最常使用的 Cache Line，在被其他行替换前，要尽可能多地重复使用。这样一来，初始损失影响最小且平均存储器访问时间变得最短。

Cache 使用相同行帧来存储冲突的 Cache Line，替换一个行帧将导致从 Cache 中驱逐另一个行帧。如果后续驱逐的行帧又被访问，那么访问会缺失且这个行帧必须再次从低速存储器取出。因而，只要一个行帧还会被使用，应避免它被驱逐。

1. 冲突和容量缺失

一个 Set 对应的数据已经被 Cache 缓冲，随后同一个 Set 的其他存储器位置被访问，就会由于冲突导致驱逐，这个类型的缺失被称为冲突缺失。一个冲突缺失的产生是因为一个 Cache Line 在它被使用前因为冲突被驱逐，更深层次的原因可能是因为 Cache 容量被耗尽，从而导致冲突发生。

如果 Cache 容量被耗尽，当缺失发生时，Cache 中的所有行帧被分配，这就是一个容量缺失。如果一个数据组超过重用 Cache 容量，容量缺失发生。当容量耗尽，新行访问从数组

开始逐步替代旧行。

确认一个缺失的原因有助于选择相应措施避免缺失。冲突缺失意味着数据访问合乎Cache大小，但是Cache Line因为冲突被驱逐。在这种情况下，我们可能需要改变存储器布局，以便数据访问被分配到存储器中Cache没有冲突的地址中。或者，从硬件设计上，我们可以创建多个Set保持两个或更多行。因而，存储器的两个行映射到相同Set可以都被保持在Cache中，相互不驱逐。这就是组相联的Cache。为了避免容量缺失，需要减少一次操作数据的数量。

2. 强制性缺失

第三类缺失是强制性缺失或首次引用缺失。当数据第一次传入，在Cache中没有该数据的缓存，因而肯定发生该类型Cache缺失。与其他两种缺失不同，这种缺失不刻意避免，因而是强制的。

3.4.3 组相联 Cache

组相联Cache具有多路Cache以减少冲突缺失的可能性。C66x L1D Cache是一个2路组相联的Cache，具有4KB、8KB、16KB或32KB容量，并且Cache行尺寸为64字节。L1D Cache的特点在表3.1中描述。表3.2提供了L1D缺失阻塞特征。

表3.1 L1D Cache特点

特　征	C66x DSP	C64x DSP
组织	2路组相联	2路组相联
协议	读分配 Read Allocate，Writeback	读分配 Read Allocate，Writeback
内核访问时间	1周期	1周期
容量	4KB、8KB、16KB或32KB	16KB
行尺寸	64字节	64字节
替换策略	最近经常使用(LRU)	最近最少使用(LRU)
写缓冲	4 × 128位	4 × 64位
外部存储器容量	可配置	可配置

表3.2 L1D缺失阻塞特征

参　数	L2类型			
	0 Wait-State，2×128-bit Banks		1 Wait-State，4×128-bit Banks	
	L2 SRAM	L2 Cache	L2 SRAM	L2 Cache
单个读缺失	10.5	12.5	12.5	14.5
2并行读缺失(流水)	10.5+4	12.5+8	12.5+4	14.5+8
M连续的读缺失(流水)	10.5+3×(M−1)	12.5+7×(M−1)	12.5+3×(M−1)	14.5+7×(M−1)
M连续的并行读缺失(流水)	10.5+4×(M/2−1)+3×M/2	12.5+8×(M/2−1)+7×M/2	12.5+4×(M−1)	14.5+8×(M/2−1)+7×M/2
在读缺失时Victim缓冲清空	破坏缺失流水最大11个周期阻塞	破坏缺失流水最大11个周期阻塞	破坏缺失流水最大10个周期阻塞	破坏缺失流水最大10个周期阻塞
写缓冲流出速度	2周期/条目	6周期/条目	2周期/条目	6周期/条目

与直接映射 Cache 相比，2 路组相联 Cache 的每个 Set 由两个行帧组成：一个行帧在路 0；另一个行帧在路 1。存储器中的一条 Cache Line 仍然映射一个 Set，不过现在可以存入两个行帧中的任一条。从这个意义上讲，一个直接映射的 Cache 也可以被看成一个 1 路 Cache。组相联的 Cache 架构如图 3.3 所示。与直接映射类似，除了两个 Tag 比较不一样（组相联的 Cache 中多路都进行 Tag 比较），Cache 命中和缺失的机理相似。

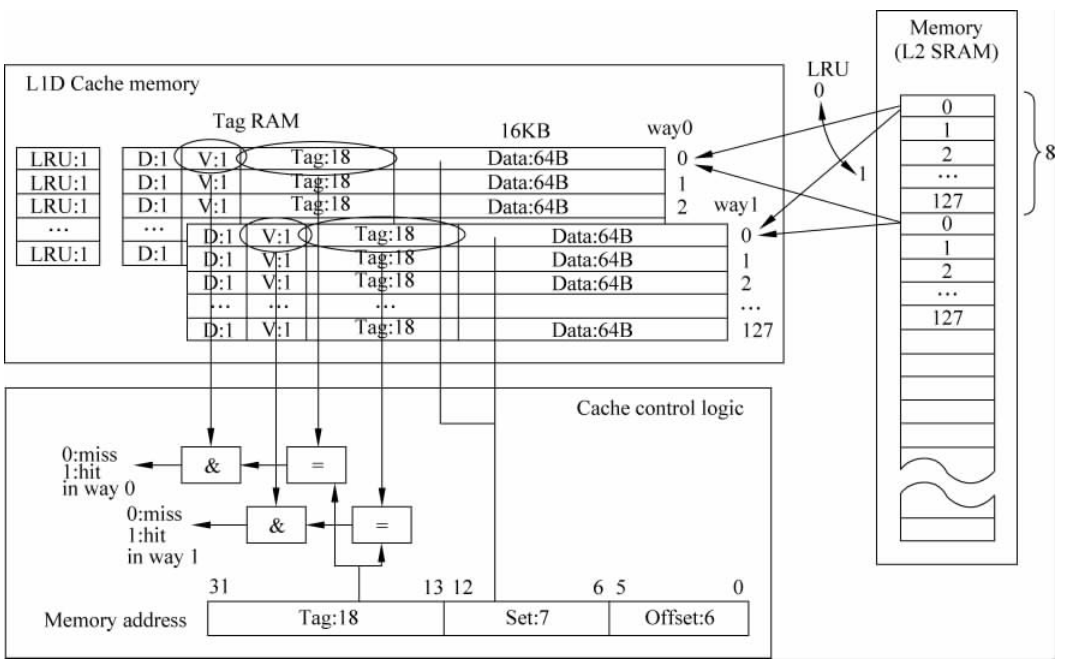

图 3.3 组相联 Cache 架构

1. 读缺失

如果两路都为读缺失，数据首先从存储器被取出。LRU（Least Recently Used）位决定 Cache 行帧被分配在哪一路中。每个 Set 有一个 LRU 位，可以被认为是一个开关。如果 LRU 位是 0，行帧在路 0 被分配；如果 LRU 位是 1，行帧在路 1 被分配。任何时候只要存在一个到该行帧的访问，LRU 的状态位就被改变。当一路被访问，LRU 位总是切换到相反的路，为的是保护最近使用的行帧不被驱逐。

基于位置原理，最近最少使用原则（LRU）被用来在同一 Set 里选择一个行帧作为被驱逐的行，用于保存新的 Cache 数据。

2. 写缺失

L1D 是一个读分配的 Cache，意味着在读缺失时一个行帧被分配到 Cache。在一个写缺失时，数据通过一个写缓冲被写到更低级存储器，不会因此而产生新的 L1D Cache 关系。写缓冲有 4 个条目（entry），在 C66x 器件中每个 entry 是 128 位宽。

3. 读命中

如果在路 0 有一个读命中，该行帧的数据在路 0 被访问；如果在路 1 有一个读命中，该行帧的数据在路 1 被访问。

4. 写命中

在一个写命中活动中，数据被写到 Cache，但是不是立即传递到更低的存储器。这种类型的 Cache 被称为写回 write-back Cache，因为数据被一个内核的写访问修改并且在之后被写回到存储器。为了写回被修改的数据，哪一行被核写回必须清楚。为了实现这个目的，每条 Cache Line 具有一个 Dirty 位和它相关。最初，Dirty 位是 0。只要核写到一个被 Cache 的行，相应的 Dirty 位被设置。因为读缺失冲突，当 Dirty 的行需要被驱逐，它会被写回到存储器。如果那一行没有被修改(Clean Line)，它的内容被丢弃。例如，假设行在 Set 0，路 0 被内核写，LRU 位指示在下一个缺失时路 0 将会被替换；如果内核当前产生一个到存储器位置映射到 Set 0 的地址的读访问，当前的 Dirty 行首先写回到存储器，随后新数据被存储到这个行帧。一个写回可能被程序发起，通过发送一个写回命令到 Cache 控制器。

3.4.4 二级 Cache

如果在存储器尺寸和访问时间上，Cache 和主存储器之间有较大差别，二级 Cache 被引进用于减少更多存储器访问数量。二级 Cache 基本操作方式与 1 级 Cache 相同；然而，2 级 Cache 在容量上更大。1 级和 2 级 Cache 相互作用如下：一个地址在 L1 缺失就传给 L2 处理；L2 使用相同的 Valid 位和 Tag 比较来决定被请求的地址是否在 L2 Cache。L1 命中直接从 L1 Cache 得到服务，并不需要牵涉 L2 Cache。

与 L1P 和 L1D 一样，L2 存储空间可以被分成一个可寻址的内部存储器(L2 SRAM)和一个 Cache (L2 Cache)部分。与 L1 Cache 只有读分配(read allocate)不一样，L2 Cache 是读分配和写分配(write allocate)的 Cache。L2 Cache 只被用来 Cache 缓存外部存储器地址，然而，L1P 和 L1D 被用于 Cache 缓存 L2 存储器和外部存储器地址。L2 Cache 特征概述如表 3.3 所示。

表 3.3 L2 Cache 特征

特　　征	C66x DSP	C64x DSP
组织方式	4 路组相联	4 路组相联
协议	读分配和写分配	读分配和写分配
	写回	写回
容量	32KB、64KB、128KB 或 256KB	32KB、64KB、128KB 或 256KB
行尺寸	128B	128B
替换策略	最近使用(LRU)	最近最少使用(LRU)
外部存储器容量	可配置	可配置

1. 读缺失和读命中

考虑一个内核读请求的场景，即访问可被 Cache 缓存的外部存储器地址，而 Cache 在 L1 缺失(可能是 L1P 或 L1D)。如果地址也在 L2 Cache 缺失，相应的行会引入 L2 Cache。LRU 位决定了哪路行帧被分配到其中。如果行帧包含 Dirty 数据，在新行被取出前，首先会写回到外部存储器(如果这一行的数据也包含在 L1D，在 L2 Cache Line 被发送给外部存

储器前，首先会写回到 L2。为保持 Cache 一致性，这个操作是需要的）。最新分配的一行形成一个 L1 Line 并包含请求的地址，然后传送给 L1。L1 在其 Cache 存储器中存储该行，并最后发送请求的数据到内核。如果在 L1 中新行替换一个 Dirty 行，它的内容首先写回到 L2。如果地址是一个 L2 命中，相应的行直接从 L2 传到 L1 Cache。

2. 写缺失和写命中

如果一个核写请求到一个外部存储器地址在 L1D 中缺失，它将被通过写缓冲传送给 L2。如果对于这个地址 L2 检测到一个缺失，相应的 L2 Cache Line 被从外部存储器取出，被用内核写操作修改并被存入分配的行帧中。LRU 位决定哪路行帧用于分配给新数据。如果行帧包含 Dirty 数据，它会在新行取出前首先被写回到外部存储器。注意新行没有存储进 L1D，因为它只是一个 read-allocate Cache。如果地址是一个 L2 命中，相应的 L2 Cache Line 直接更新为核写的数据。

3. 外部存储地址 Cache 能力

L2 SRAM 地址总是 Cache 缓存进 L1P 和 L1D，然而，默认状态下，外部存储地址在 L1D 和 L2 Cache 中，被分配为不可 Cache 缓存的。因此，Cache 能力必须首先被用户明确使能。注意 L1P Cache 是不被配置影响的，并且总是 Cache 缓存外部存储器地址。如果地址是不可 Cache 缓存的，任何存储器访问（数据访问或程序取）无须分配行到 L1D 或 L2 Cache。

3.5 L1P Cache

C66x 内核中 L1P 与 L1D 上电后默认全为 Cache，与 L1D Cache 不同的是 L1P Cache 为直接映射 Cache。本节描述 L1P Cache 的相关知识。

3.5.1 L1P 存储器和 Cache

L1P 存储器和 Cache 的目的就是最大化程序执行效率。L1P Cache 的可配置性为系统设计提供了灵活性。

L1P Cache 的特点为：L1P Cache 可配置成 0KB、4KB、8KB 和 32KB，存储器保护可配置，Cache 块和全局一致性操作可配置。

L1P 存储器支持最大 128KB 的 RAM 空间（具体参见器件配置情况）。L1P 存储器不能被同一个核内的 L1D、L1P 和 L2 Cache 缓存。

L1P 只能被 EDMA 和 IDMA 写，不能被 DSP 存储写入。L1P 可以被 EDMA、IDMA 和 DSP 访问读取。

L1P 存储器最大的等待状态为 3 周期，等待周期不能被软件配置，这是由具体器件决定的。L1P 存储器等待状态通常为 0 个周期。

为了在一个较高的时钟频率取程序代码并维持一个较大的系统空间，L1P Cache 是很有必要的，并可以把部分或全部的 L1P 都作为 Cache。从 L1P 存储器地址映射的最顶端开始，采用自顶向下的顺序，L1P 把存储器转换为 Cache。最高地址的 L1P 存储器首先被 Cache 缓存。

用户可以通过寄存器控制 L1P Cache 的操作。表 3.4 列出了这些寄存器概要。

表 3.4 L1P Cache 寄存器概要

地　址	缩　略　词	寄存器描述
0184 0020h	L1PCFG	L1 程序配置寄存器
0184 0024h	L1PCC	L1 程序 Cache 控制寄存器
0184 4020h	L1PIBAR	L1 程序无效基址寄存器
0184 4024h	L1PIWC	L1 程序无效计数(字)寄存器
0184 5028h	L1PINV	L1 程序无效寄存器

3.5.2 L1P Cache 结构

L1P Cache 是直接映射的 Cache,意味着系统中每个物理存储位置都在 Cache 中有一个可能归属的位置。当 DSP 想取一段代码,DSP 首先要检查请求的地址是否在 L1P Cache 中。为了实现这个功能,DSP 提供的 32 位地址被分割成三段(Tag、Set 和 Offset),如图 3.2 所示。

Offset 占 5b,是因为 L1P 行尺寸为 32 字节。Cache 控制逻辑忽略地址的 0～4 位信息。如果被 Cache 缓存,Set 域指示被 Cache 缓存的数据位于 L1P Cache 中行的地址。Set 域的宽度主要取决于 L1P 被配置为 Cache 的大小。对于任何已经在这些地址被 Cache 缓存的数据,L1P 用 Set 来查找,并检查 Tag,如 Valid 标志位指示在 Tag 中地址是否代表 Cache 中的一个有效的地址。

如果 L1P 读缺失,请求被发给 L2 控制器,数据从系统中的位置中被取出。L1P 缺失可能会,也可能不会直接导致 DSP 阻塞(Stall)。L1P 通常情况下不能被 DSP 写。

替换规则:在所有 Cache 配置中,L1P Cache 都是直接映射的。这意味着每个系统存储位置在 L1P Cache 中有且只有一个对应位置。由于 L1P 直接映射,其替换策略为:每个新的 Cache Line 替换之前被 Cache 的数据。

L1P Cache 结构允许在运行中更改 L1P Cache 的大小,以实现 L1P 模式改变操作。具体操作为向 L1PCFG 寄存器中 L1PMODE 域写入与 L1P Cache 大小相对应的模式设置,对照表如表 3.5 所示。

表 3.5 L1 Cache 配置大小

L1PCFG 中 L1PMODE 设置	L1P Cache 大小	L1PCFG 中 L1PMODE 设置	L1P Cache 大小
000b	0KB	100b	32KB
001b	4KB	101b	最大的 Cache 对应 32KB
010b	8KB	110b	
011b	16KB	111b	最大的 Cache 对应 32KB

L1P Cache 模式的范围取决于实际的 L1P 存储器的大小。例如,如果 L1P 存储器的真实大小为 16KB,那么 L1P Cache 最大为 16KB(L1PMODE 为 111b 对应最大的 Cache)。

当程序发起一个 Cache 模式转换,L1P Cache 使当前内容无效。这样可以保证在更改 Cache Tag 的说明时没有错误的命中发生。

为保证正确的 Cache 缓存行为，使数据无效是必要的，但对于防止由于重新分配 L1P RAM 成为 Cache 而导致的数据丢失，还是不够的。为了安全转换 L1P 模式，应用程序必须遵循以下步骤，如表 3.6 所示。

表 3.6 L1P Cache 模式切换准则

由 此 切 换	到	程序必须执行以下步骤
没有或具有较少 L1P Cache	更多的 L1P Cache	(1) DMA、IDMA 或拷贝任何影响范围之外的 L1P RAM 数据(如果不需要保存，就不用 DMA) (2) 把想写入的模式写入 L1PCFG 寄存器的 L1PMODE 域 (3) 读回 L1PCFG，这样使 DSP 暂停直到模式转换结束
具有较多的 L1P Cache	没有或具有较少 L1P Cache	(1) 把想写入的模式写入 L1PCFG 寄存器的 L1PMODE 域 (2) 读回 L1PCFG，这样使 DSP 暂停直到模式转换结束

3.5.3 L1P 冻结模式

对于应用层面，L1P Cache 直接支持一种冻结模式。这种模式下允许应用避免 DSP 的数据访问破坏 Cache 中的程序代码。这种特征在中断上下文时十分有用。在冻结模式，L1P 仍然支持命中访问，读命中从 Cache 返回数据。但是，在该模式，既不允许对新的读缺失分配 Cache Line，也不允许现存的 Cache Line 内容被标志为 invalid。

L1PCC 寄存器的 OPER 域控制 L1P 是正常工作还是被冻结。DSP 可以向 L1PCC 的 OPER 域写 001b 使 L1P 进入冻结模式，向 L1PCC 的 OPER 区域写 000b 使 L1P 进入正常状态。

L1P 冻结模式 Cache 可以通过 CSL 函数被控制：

```
Cache_freezeL1p();
Cache_unfreezeL1p();
```

3.5.4 程序启动的一致性操作

1. 全局一致性操作

在主要的事件如任务切换、L1P 模式改变、存储器保护设置改变等活动中，全局一致性操作使 L1P 与系统同步。全局一致性操作可以在软件控制中使 L1P Cache 全局无效，通过向 L1PINV 寄存器中的 1 位写入 1 发起全局无效操作。

2. 块一致性操作

块一致性操作与全局一致性操作相似，但块一致性只应用于定义好的程序块。L1PIBAR 和 L1PIWC 分别定义块的基址和字数(32 位)。

3.6 L1D Cache

如上一节所述，C66x 内核中 L1P 与 L1D 上电后默认全为 Cache，不同的是 L1D Cache 为两路组相联 Cache。本节描述 L1D Cache 的相关知识。

3.6.1 L1D 存储器和 Cache

L1D 存储器和 Cache 的目的就是最大化数据处理性能。L1D 存储器和 Cache 可配置性为系统设计提供了灵活性。

L1D 存储器和 Cache 具有以下特征：

L1D Cache 可配置成 0KB、4KB、8KB 和 32KB，具备存储器保护功能，可以进行 Cache 块和全局一致性操作。

L1D 存储器不能被同一个核内 L1D、L1P 和 L2 Cache 缓存。L1D 被初始化成全是 Cache。C66x L1D 存储器和 Cache 结构允许转换一部分或全部 L1D 为一个 read-allocate、writeback、两路组相联 Cache。

3.6.2 L1D Cache 结构

L1D Cache 是一个两路组相联 Cache，意味着系统中每个物理存储器位置在 Cache 中具有两个可能的映射位置。当 DSP 试图访问一片数据，L1D Cache 必须检查请求的地址是否保留在 L1D Cache 任一路中。

为此，DSP 提供的 32 位地址被分割成 6 个域，如图 3.4 所示。

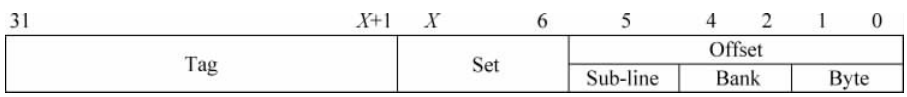

图 3.4 L1D 数据访问地址结构

6 位的 Offset 说明实际上 L1D 行尺寸为 64 字节。Cache 控制逻辑忽略地址的 0～5 位（Byte、Bank 和 Sub-line 域）。0～5 位决定哪个 Bank 及 Bank 中哪个字节被访问，因而它们与 Cache 的 Tag 比较逻辑是无关的。Set 域指示数据如果被 Cache 缓存，将会被保存的 L1D Cache Line 地址。Set 域的宽度取决于用户配置 L1D Cache 的大小，如表 3.5 所示。与检查 Valid 位一样，使用 Set 域来查找并对任何已经从那个地址被 Cache 缓存的数据检查每一路的 Tag 域，Valid 指示 Tag 中的地址是否实际代表一个保存在 Cache 中的有效地址。

Tag 域是地址的高位部分，确定数据元素的真实物理位置。Cache 功能检查 L1D Cache 中两路保存的 Tag。如果一个 Tag 匹配，在读操作时，相应的 Valid 位如果被设置，那么就是一个命中，数据缓存直接从 L1D Cache 中返回数据，否则，读为缺失。DSP 也可以写数据到 L1D。当 DSP 执行存储（store）操作，L1D 执行与读操作类似的 Tag 比较操作。如果发现为一个有效的匹配，写操作就为一个命中，数据被直接写到 L1D Cache 位置。否则，写操作为一个缺失，并且数据在 L1D 写缓冲排队，这个缓冲被用于在写缺失时减少 DSP 阻塞周期。由于 DSP 不等待数据从写操作返回，因而不会在 L2 访问时阻塞。

L1D Cache 配置决定 Set 尺寸和 Tag 域，如表 3.7 所示。

表 3.7 L1D Cache 配置大小

L1DCFG 中 L1DMODE 设置	L1D Cache 大小	'X' Bit 位置	描 述
000b	0KB	N/A	L1D 全为 RAM
001b	4KB	10	32 L1D Cache line
010b	8KB	11	64 L1D Cache line
011b	16KB	12	128 L1D Cache line
100b	32KB	13	256 L1D Cache line
101b	保留,映射为 32KB		
110b			
111b	最大的 Cache 对应 32KB		

3.6.3 L1D 冻结模式

对于应用的操作,L1D Cache 直接支持冻结模式。这个模式允许实时应用在不同代码段执行时(如中断处理程序)限制从 L1D 被驱逐的数据数量。L1D 冻结模式只影响 L1D Cache。L1D RAM 不会被冻结模式影响。

在冻结模式,L1D Cache 正常服务于读命中和写命中,稍微不同的是 LRU 位不被修改。读命中从 Cache 返回数据,写命中更新 Cache Line 中被 Cache 缓存的数据并且根据需要标记为 Dirty; LRU 位没有被更新。在冻结模式,当读缺失时,L1D Cache 不再分配新 Cache Line,也不会驱逐任何已经存在的 Cache 内容。

写缺失在 L1D 写缓冲中被正常排队。在冻结模式,L1D Cache 仍然正常响应从 L2 发起的 Cache 一致性命令(snoop-read、snoop-write),与任何程序发起的 Cache 控制一样(writeback、invalidate、writeback-invalidate 和模式切换)。L1D 的冻结模式对 L2 是否分配 Cache Line 没有影响。同样地,L2 的冻结模式对 L1D 是否分配 Cache Line 没有影响。L1DCC 寄存器中的 OPER 域控制 L1D 冻结模式。通过写 1 到 OPER 域,DSP 把 L1D 置为冻结模式; 通过写 0 到 L1DCC 寄存器 OPER 域,DSP 恢复 L1D 为正常操作状态。

L1DCC 寄存器 POPER 域保存 OPER 域以前的值。L1DCC 寄存器 OPER 域的值拷贝到 L1DCC 寄存器 POPER 域。这缓解了在被写之前读 L1DCC 寄存器的读开销周期(为了保存 OPER 以前的值)。

当用户需要执行一个写操作到 L1DCC 寄存器,执行以下操作:

(1) L1DCC 寄存器中 OPER 域的内容拷贝到 POPER 域;

(2) POPER 域失去之前的值;

(3) OPER 域根据 DSP 写 L1DCC 寄存器 bit 0 的值更新,因而,写 L1DCC 寄存器只修改了 L1DCC 的 OPER 域。

为了确保 L1PCC 寄存器更新了,软件必须在执行一个写操作到 L1PCC 寄存器之后紧跟一个读 L1PCC 寄存器的操作,这确保请求的模式生效。程序不能用一个写操作直接修改 POPER 域。

3.6.4 程序发起的 Cache 一致性操作

C66x L1D Cache 支持程序发起的 Cache 一致性操作,这些操作或操作在块地址,或操

作在整个 L1D Cache。

以下 Cache 一致性操作被支持。

(1) 失效(Invalidate)：有效的 Cache Line 被设置成无效，受影响的 Cache Line 被废弃。

(2) 写回(Writeback)：所有 Dirty 的 Cache Line 写回到更低级别的存储器。

(3) 写回-失效(Writeback-Invalidate)：写回操作之后紧接着是失效操作，只有 Dirty 的 Cache Line 被写回到更低级别的存储器，但是所有行被失效。

3.7 L2 Cache

通常，L2 被配置为全是 SRAM，可以用于做数据缓冲。不过，L2 也可以配置为 Cache，与 L1P 和 L1D Cache 不同的是，L2 Cache 为 read allocate 和 write allocate 的 4 路组相联 Cache。

3.7.1 L2 存储器和 Cache

在一个使用 C66x 内核的器件中，L2 存储器和 Cache 提供灵活的配置方式，为系统设计提供了灵活性。

L2 存储器和 Cache 具有以下特征：

L2 Cache 可配置成 32KB、64KB、128KB、256KB、512KB 或 1MB(最大值由器件配置决定)，具备存储器保护功能，支持 Cache 块和全局一致性操作。

C66x 内核默认配置映射所有 L2 存储器全为 RAM。L2 存储器控制器支持容量为 32KB、64KB、128KB、256KB、512KB 或 1MB 的 Cache，为 4 路组相联 Cache。

L2 Cache 通过一些寄存器来被控制，如表 3.8 所示。

表 3.8 L2 Cache 寄存器概要

地　址	缩　略　词	寄存器描述
0184 0000h	L2CFG	L2 配置寄存器
0184 4000h	L2WBAR	L2 Writeback 基址寄存器
0184 4004h	L2WWC	L2 Writeback 字(word)数寄存器
0184 4010h	L2WIBAR	L2 Writeback-Invalidate 基址寄存器
0184 4014h	L2WIWC	L2 Writeback-Invalidate 字(word)数寄存器
0184 4018h	L2IBAR	L2 Invalidate 基址寄存器
0184 401Ch	L2IWC	L2 Invalidate 字(word)数寄存器
0184 5000h	L2WB	L2 Writeback 寄存器
0184 5004h	L2WBINV	L2 Writeback-Invalidate 寄存器
0184 5008h	L2INV	L2 Invalidate 寄存器
	MARn	存储器属性寄存器

3.7.2 L2 Cache 结构

L2 Cache 是一个读和写分配(read allocate 和 write allocate)的 4 路组相联 Cache。为

了跟踪 L2 Cache Line 的状态，L2 Cache 控制器包含一个 4 路 Tag RAM。L2 Tag 中的地址结构是一个 Cache 和 RAM 分割的函数，由 L2CFG 寄存器 L2MODE 域控制。

L2 数据访问地址结构如图 3.5 所示。

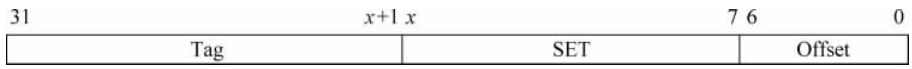

图 3.5 L2 数据访问地址结构

7 位的 Offset 说明 L2 行尺寸实际上为 128 字节。Cache 控制逻辑忽略地址的这部分域。数据如果被 Cache 缓存，Set 域指示数据将会存进每路中 L2 Cache Line 的地址。Set 域的宽度取决于用户配置 L2 为 Cache 的大小，如表 3.9 所示。

L2 控制器用 Set 域查找，并检查任何已经被 Cache 缓存的数据每一路的 Tag 域。同时控制器检查 Valid 位，以确认 Cache Line 的内容对 Tag 比较是否有效。

表 3.9 L2 Cache 配置大小

L2CFG 中 L2MODE 设置	L2 Cache 大小	'*x*' Bit 位置	描 述
000b	0KB	N/A	L2 全为 RAM
001b	32KB	12	64 L2 Cache Line
010b	64KB	13	128 L2 Cache Line
011b	128KB	14	256 L2 Cache Line
011b	256KB	15	512 L2 Cache Line
100b	512KB	16	1024 L2 Cache Line
110b	1024KB	17	2048 L2 Cache Line
111b	最大的 Cache 对应 1024KB		

注意：总的来说，一个大的 L2MODE 值指定一个大的 Cache 尺寸，最大为器件有效的 L2 存储器容量。

Tag 域是指示 Cache Line 的物理位置地址的高位部分。Cache 对一个给定的地址与存储的所有 4 路的 Tag 域进行比较。如果任何一个 Tag 匹配成功并且 Cache 数据是 Valid，那么访问就为命中，数据元素直接从 L2 Cache 读或直接写到 L2 Cache 位置。否则，就是一个缺失。当 L2 从系统存储器位置取完整的行时，请求者保持阻塞。

L2 Cache 的替换机制为典型的(LRU)机制。

3.7.3 L2 冻结模式

L2 Cache 提供一个冻结模式。L2 Cache 的内容在这种模式被冻结(也就是说，在正常操作时不会更新)。L2 冻结模式允许一个实时应用在不同代码段期间(如中断服务程序)限制从 L2 驱逐数据的数量。用 L2CFG 寄存器 L2CC 域来设置这种模式。冻结模式只影响 L2 Cache 的操作。同样地，L1 的冻结模式不影响 L2 Cache。

在冻结模式，L2 Cache 正常响应读和写命中。L2 直接发送读和写缺失到外部存储器，就像 L2 Cache 不存在。当冻结时，L2 不分配新的 Cache Line。在冻结模式，只可能由程序发起的 Cache 一致性操作从 L2 驱逐 Cache Line。

3.7.4 程序发起的 Cache 一致性操作

L2 存储器架构支持多种一致性操作，分成两种基本类型：到一个指定地址范围的块操作和操作一个或更多 Cache 的全部内容的全局操作。下面列出了存储器支持的 Cache 一致性操作。

(1) 失效(Invalidate)：有效的 Cache Line 被设置成无效，受影响的 Cache Line 内容被废弃。

(2) 写回(Writeback)：Valid 和 Dirty 的 Cache Line 内容被写到更低级别存储器中。

(3) 写回失效(Writeback-invalidate)：写回操作紧跟一个失效，只有受影响的 Cache Line 内容被写回到更低级别的存储器，但是所有行都失效了。

1. 全局一致性操作

全局一致性操作在整个 L2 Cache 上执行。一些全局一致性操作也会影响 L1 Cache。表 3.10 列出了所有 L2 全局一致性命令和它们在每个 Cache 上执行的操作

表 3.10 全局一致性操作

Cache 操作	使用的寄存器	L1P 影响	L1D 影响	L2 影响
L2 写回	L2WB	没影响	所有更新的数据写回 L2 或外部存储器，但 L1D 中 Cache 关系保持有效	所有更新的数据写回外部存储器，但 L2 中 Cache 关系保持有效
L2 写回并失效	L2WBINV	所有在 L1P 的行失效	所有更新的数据写回 L2 或外部存储器，所有 L1D 中的行无效	所有更新的数据写回外部存储器，所有 L2 中的行无效
L2 失效	L2INV	所有在 L1P 的行失效	所有 L1D 中的行无效，更新的数据被废弃	所有 L2 中的行无效，更新的数据被废弃

程序发起全局 Cache 一致性操作通过写 1 到每个 L2WB、L2WBINV 和 L2INV 寄存器中相应的寄存器位。操作完成后，程序可以查询相应的寄存器位以决定什么时候命令结束。

以下给出如何使用 L2WBINV 寄存器的示例。

```
L2WBINV = 1; /* Write back and Invalidate anything held in cache. */
/* ---------------------------------------------------------------- */
/* OPTIONAL: Spin waiting for operation to complete. */
/* ---------------------------------------------------------------- */
while ((L2WBINV & 1) != 0)
;
```

对于这些命令的完成，硬件实现查询，不需要软件参与。然而，当全局命令正在进行时，硬件可能使程序阻塞。不考虑 L2 冻结的状态，全局一致性操作正常工作。而且，全局一致性操作不会改变 L2 冻结的状态。

2. 块一致性操作

块一致性操作具有与全局一致性类似的功能，但是块一致性操作只能应用于一个限定的数据块。块通过相关寄存器中的基地址和字长度来确定。表 3.11 为块 Cache 操作。

表 3.11 块 Cache 操作

Cache 操作	使用的寄存器	L1P 影响	L1D 影响	L2 影响
L2 写回	L2WBAR L2WWC	没影响	所有更新的数据写回 L2 或外部存储器，但 L1D 中保持有效	所有更新的数据写回外部存储器，但 L2 中 Cache 关系保持有效
L2 写回并失效	L2WIBAR L2WIWC	所有范围内的 L1P 行失效	所有更新的数据写回 L2 或外部存储器，所有范围内的行在 L1D Cache 中失效	所有更新的数据写回外部存储器，所有范围内的行在 L2 中失效
L2 失效	L2IBAR L2IWC	所有范围内的 L1P 行失效	所有地址范围内的行在 L1D 中失效，更新的数据被废弃	所有范围内的行在 L2 中的 Cache 关系失效，更新的数据被废弃

程序通过写一个字的地址到基地址寄存器发起块 Cache 操作，然后写一个字的计数到计数寄存器(写 1 到 WC 表示 4 个字节)。如有必要，C66x 内核需严格实施以下准则：

(1) 每次只有一个程序发起的一致性操作正在进行中。

(2) 当另一个块或全局 Cache 一致性操作在进行中，写到 L2XXBAR 或 L2XXWC 导致阻塞。

L2XXBAR/L2XXWC 用于建立块 Cache 操作机制，允许用户指定范围，最小颗粒度为字。然而，存储器系统操作粒度以 Cache Line 为单位，因而，所有覆盖范围的行被执行。

对于 C66x DSP，推荐程序等待块一致性操作完成后再继续。为了执行一个块一致性操作，执行以下命令：

(1) 写起始地址到 L2XXBAR 寄存器。

首先，写一个非 0 值到 L2XXBAR 寄存器设置了下一个 Cache 一致性操作的基地址。

(2) 写字计数到 L2XXWC 寄存器。

写一个非 0 值到 L2XXWC 发起这个操作。在一个 Cache 操作之后或正在操作中，程序必须依赖 L2XXBAR 的内容。最好的方式是：程序应该总是先写一个新值到 L2XXBAR，然后再写 L2XXWC。

(3) 通过以下途径之一等待完成：

①执行一个 MFENCE 指令(推荐)；②查询 L2XXWC 寄存器直到字计数域读为 0。

MFENCE 指令对于 C66x DSP 是新的，它使 DSP 阻塞直到所有未完成的存储器操作完成。

当一个块 Cache 操作正在进行中，读 L2XXWC 返回一个非 0 的值，当它完成返回为 0。无论 L2 是否为冻结状态，块一致性操作正确地工作。

3.7.5 Cache 能力控制

在一些应用中，访问一些指定的地址可能需要从它们物理位置读(如 FPGA 中的状态寄存器、多核共享的控制信号等)。L2 控制器提供寄存器组，用来控制特定范围内的存储器是否可以 Cache 缓存、是否一个或更多请求者实际被允许访问这些范围。这些寄存器指的

是 MAR(存储器属性寄存器)寄存器。

注意：使用 C 语言中 volatile 关键字不能保护一个变量不被 Cache 缓存。

如果一个应用使用一个周期性被外部硬件更新的存储器位置，为了在 C 代码中保护这个操作，需要遵循以下两步：

(1) 使用 volatile 关键字阻止代码产生工具不正确的优化变量。

(2) 用户必须设置覆盖范围包含该变量的 MAR 寄存器，以阻止 Cache 缓存功能。

1. MAR 功能

每个 MAR 寄存器为 2b：许可拷贝(Permit Copies，PC)和外部可预取(Prefetchable Externally，PFX)。每个 MAR 寄存器的 PC 位控制 Cache 功能是否可以保持被影响地址范围的一份拷贝。如果 PC = 1，影响的地址范围可以被 Cache 缓存；如果 PC = 0，影响的地址范围不可以被 Cache 缓存。每个 MAR 寄存器的 PFX 位用来向 XMC 表达一个给定地址范围是否可以预取。如果 PFX = 1，影响的地址范围是可以预取的。如果 PFX = 0，影响的地址范围是不可以预取的。

2. 特殊 MAR 寄存器

MAR0～MAR15 代表 C66x 内核中保留的地址范围，并按如下处置：

(1) MAR0 是一个只读寄存器。MAR0 的 PC 总是被读为 1。

(2) MAR1～MAR11 与内部和外部配置地址空间相对应。因而，这些寄存器是只读的，并且 PC 域总是被读为 0。

(3) MAR12～MAR15 与 MSMC 存储器对应。这些是只读寄存器，PC 总是读为 1。这使得 MSMC 存储器通过其最初地址范围访问总是在 L1D 内可被 Cache 缓存。

MAR 寄存器定义如图 3.6 所示，存储器属性寄存器(MARn)域描述如表 3.12 所示。

图 3.6　MAR 寄存器结构

说明：R 表示只读；W 表示只写；-n 表示复位后的值；SRW 表示只被管理者读写。

表 3.12　存储器属性寄存器(MARn)域描述

位	域	值	描　　述
31～4	Reserved	0	Reserved
3	PFX	0	存储器范围不可以预取
		1	存储器范围可以预取
2～1	Reserved	0	Reserved
0	PC	0	存储器范围不可以 Cache 缓存
		1	存储器范围可以 Cache 缓存

存储器属性寄存器 MAR 地址对照表见附录 B，其中列出了各 MAR 寄存器地址、描述和定义属性的地址范围。MAR 寄存器只能被管理代码修改。

3.8 使用 Cache

如何正确地使用 Cache 是提升内核综合处理能力的一个重要因素。本节描述如何正确配置 Cache。

3.8.1 配置 L1 Cache

器件加载后的状态取决于特定的 C66x 器件，器件可能加载成全为 Cache、全为 SRAM 或为两者混合。对于 C6678，L1P 加载后全为 Cache。

在程序代码中执行相应的(CSL)命令(Cache_L1psetSize()、Cache_L1dsetSize())可以改变 L1P 和 L1D Cache 尺寸。

此外，在连接命令文件(Linker Command File)中，如果存储器被用作 SRAM 则其必须被指定。因为 Cache 不能被连接器用作代码或数据放置，所有连接命令文件中的段必须连接到 SRAM 或外部存储器。

3.8.2 配置 L2 Cache

在加载时，L2 Cache 被禁用，所有 L2 被配置为 SRAM(可寻址的内部空间)。如果 SYS/BIOS 被使用，L2 Cache 功能被自动使能；否则，在程序代码中 L2 Cache 可以通过使用相应的(CSL)命令 Cache_L2setSize()使能。

此外，在连接命令文件中，如果存储器被用作 SRAM，则其必须被指定。因为 Cache 不能通过连接器被用作代码或数据位置，所有连接命令文件的段必须连接到 SRAM 或外部存储器。

对于 L1D 和 L2，用户可以通过控制外部存储器以决定地址是否被 Cache 缓存或不被 Cache 缓存。每个 16MB 外部存储器地址空间由一个 MAR 寄存器控制(MAR 寄存器值为 0 表示不被 Cache 缓存，值为 1 表示可被 Cache 缓存)。例如，为使能外部存储器 Cache 范围 8000 0000h～80FF FFFFh，可以使用 CSL 函数 CHE_enableCaching(Cache_MAR128)设置 MAR128 为 1(PC=1)。外部存储器空间 MAR 位被设置后，通过核的新地址访问会被 Cache 缓存。如果它被设为不被 Cache 缓存，访问数据会简单地从外部存储器发送给核，而不会被存在 L1D 或 L2 Cache。

注意：程序取 L1P 总是被 Cache 缓存的，无论 MAR 的设置如何。

在加载时，外部存储地址空间的 Cache 功能是被禁用的。

以下描述假设 L2 存储器容量为 2048KB，并且 L1P 及 L1D 都是 Cache。对于具有不同的 L2 尺寸的 C66x 器件，具体参数需查相应的器件数据手册。

连接命令文件配置 1792KB SRAM 和 256KB Cache，如示例 3.1 所示。

请求的 CSL 命令顺序使能外部存储器位置的 Cache 并使能 L2 Cache，如示例 3.2 所示。通过设置相应的 MAR 位，第一个命令允许 Cache 缓存第一个外部存储空间的 16MB。最后，L2 Cache 尺寸被设置为 256KB。

图 3.7 显示对于有 2048KB L2 存储器的 C66x 器件，所有 Cache 配置的可能情况。在其他 C66x 器件中配置略有不同。

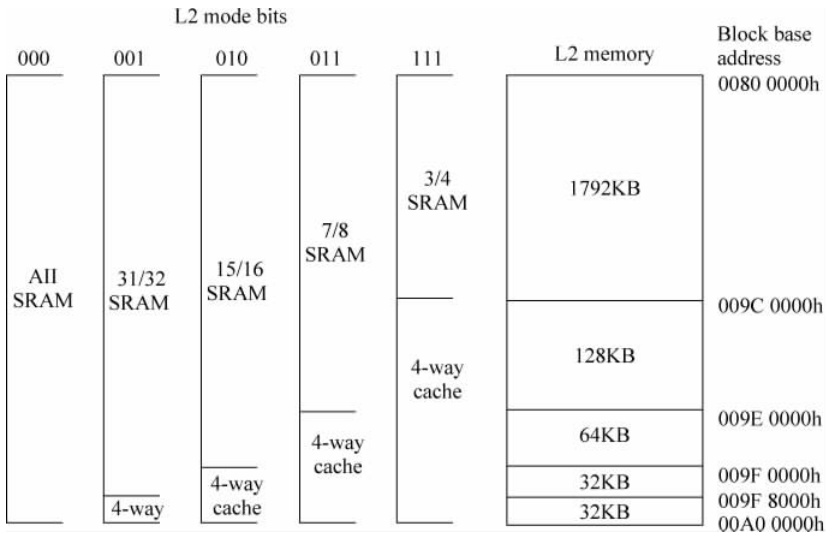

图 3.7　L2 存储器配置 Cache 情况

注意：当 L2 Cache 尺寸增大，高位存储器地址被占用。

L2 Cache 尺寸设置如示例 3.1 和示例 3.2 所示。

注意：不要在 MEMORY 指示中定义将被用作或加载起来作为 Cache 的存储器。对于连接器放置代码或数据的位置，存储器是无效的。如果要使用 L1D SRAM、L1P SRAM，首先通过减少 Cache 尺寸使 RAM 是可用的。数据或代码必须连接到 L2 SRAM 或外部存储器，然后在运行中被拷贝到 L1。

【示例 3.1】 C66x 连接命令文件。

```
MEMORY
{
    L2SRAM: origin = 00800000h length = 001C0000h
    CE0: origin = 80000000h length = 01000000h
}
SECTIONS
{
    …
    .external > CE0
}
End of 示例
```

【示例 3.2】 C66x 使能 Cache 的 CSL 命令顺序。

```
#include <csl.h>
#include <csl_Cache.h>
…
Cache_enableCaching(Cache_CE00);
Cache_setL2Size(Cache_256KCache);
End of 示例
```

3.9 数据一致性

通常，如果多个设备(如内核或外围设备)共享可被 Cache 缓存的存储器区域，Cache 和存储器可以变得不一致。了解数据一致性产生的机理，有助于正确使用 Cache。本节介绍数据一致性的相关知识。

考虑如图 3.8 所示的系统，以下描述步骤(1)到(3)与图对应。假设内核访问一个存储器位置，这导致其后该存储器相应的行被分配在 Cache 中(1)。随后，外围设备写数据到同一物理位置，这些数据打算被用于内核读取和处理(2)。然而，因为这个存储器的值保留在 Cache 中，存储器访问会命中 Cache，内核从 Cache 中读旧数据而不是从物理位置中读新数据(3)。如果核写数据到被 Cache 缓存的存储器位置，同时该数据要被外围设备读，相似的问题出现了，外围设备读的是物理位置的旧数据而不是 Cache 中的新数据。这些情况下，Cache 和存储器被称为不一致。

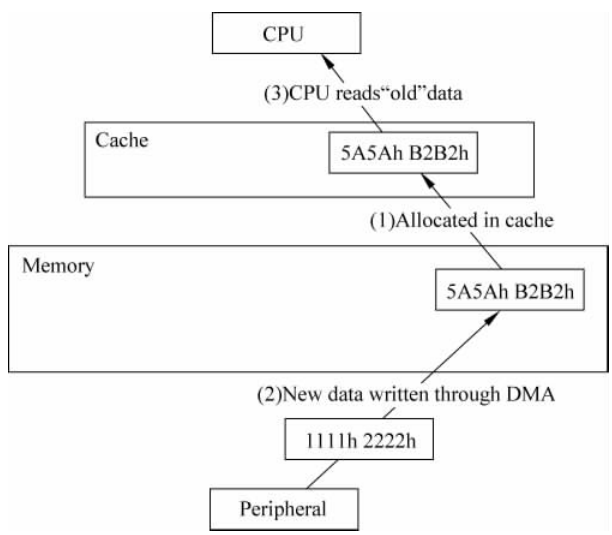

图 3.8 Cache 一致性问题

如果存在以下情况，需要考虑一致性问题：

(1) 多个请求者(CORE 数据通道、CORE 取指令通道、外围设备、DMA 控制器、其他外部实体)共享一个存储器区域用于数据交换；

(2) 这个存储器区域可以被至少一个设备 Cache 缓存；

(3) 该区域中的一个存储器位置已经被 Cache 缓存；

(4) 这个存储器位置被修改(被任何设备)。

因此，如果一个存储器位置被分享、被 Cache 缓存，并被修改，这就存在 Cache 一致性问题。

通过基于 snoop 命令的一种硬件 Cache 一致性协议，对于内核访问和 EDMA/IDMA 访问，C66x DSPs 自动保持 Cache 一致性。

在一个 DMA 读和写访问时，一致性机制被激活。当一个 DMA 读一个被 Cache 缓存

的 L2 存储器位置，数据直接从 L1D Cache 传到 DMA，不需要被 L2 SRAM 更新。在一个 DMA 写，数据传送到 L1D Cache 并在 L2 SRAM 中更新。

在以下情况中，需要靠编程人员来维护 Cache 一致性：

（1）DMA 或其他外部实体写数据或代码到外部存储器，然后被内核读；

（2）内核写数据到外部存储器，然后被 DMA 或其他外部实体读；

（3）DMA 写代码到 L2 SRAM，然后被内核执行（这种情况在 C621x/C671x 和 C64x DSPs 硬件协议中被支持，但在 C66x DSPs 中不被支持）；

（4）内核写代码到 L2 SRAM 或外部存储器，然后被该内核执行。

为了这个目的，Cache 控制器提供多种命令允许手动保持 Cache 一致性。

3.9.1 Snoop 一致性协议

Cache 控制器使用基于 Snoop 的协议来维持 L1D Cache 和 L2 SRAM 之间的 DMA 访问一致性。通常，snooping 是由更低级存储器发起的一个 Cache 操作，用来检查请求的地址是否在更高级的存储器中被 Cache 缓存（valid），如果是，则相应的操作被触发。C66x Cache 控制器支持以下 snoop 命令：L1D Snoop-Read 和 L1D Snoop-Write。

1. DMA 访问 L2 SRAM 的 Cache 一致性协议

为了说明 snooping，假设一个外围设备通过 DMA 写数据到一个分配在 L2SRAM 的输入缓冲中。然后内核读取、处理、输出缓存该数据，数据最终通过 DMA 发送到其他外围设备。

一个 DMA 写的过程如图 3.9 所示，并按如下步骤执行。

（1）外围设备请求一个写访问到 L2 SRAM 的一行，该行映射到 L1D 的 set 0。

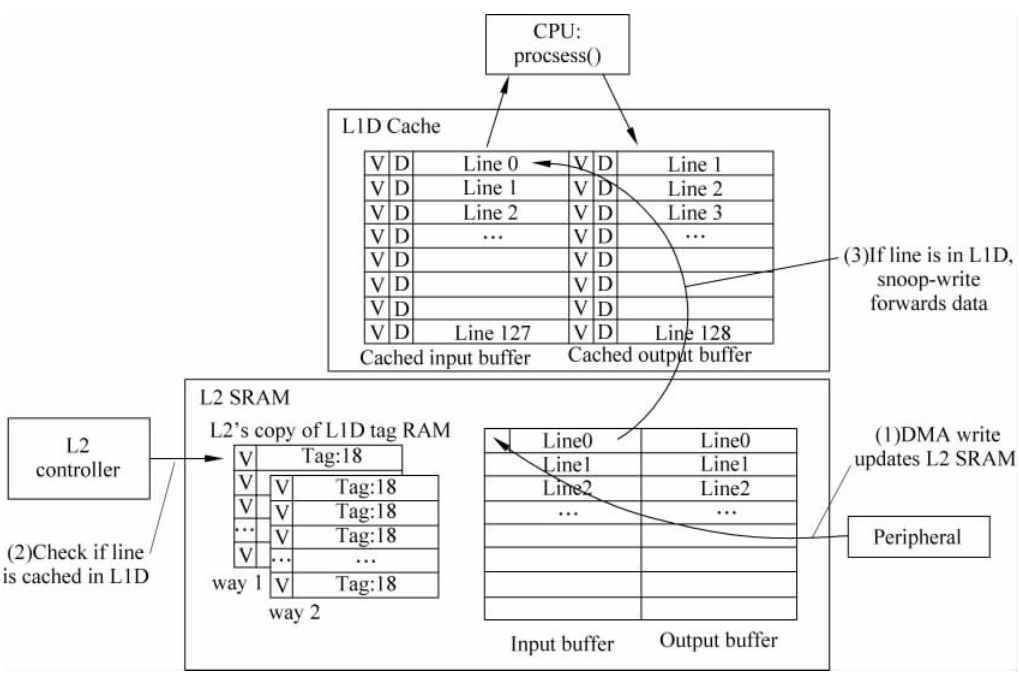

图 3.9 DMA 写访问 L2 一致性问题

(2) L2 Cache 控制器检查其 L1D tag RAM 的本地拷贝,并确定刚刚被请求的行是否被 L1D Cache 缓存(通过检查 Valid 位和 Tag 位)。如果该行没有被 L1D Cache 缓存,不需要进一步的行动,数据被写入存储器。

(3) 如果该行在 L1D 被 Cache 缓存,L2 控制器更新在 L2 SRAM 的数据,并通过执行一个 snoop-write 命令直接更新 L1D Cache。注意 Dirty 位不影响这个操作。

一个 DMA 读的过程如图 3.10 所示,并按如下步骤执行。

(1) 内核写结果到输出缓冲。假设输出缓冲被预先分配在 L1D。因为缓冲是被 Cache 缓存的,只有被 Cache 缓存的拷贝数据更新,而不是在 L2 SRAM 中的数据更新。

(2) 当外围设备执行一个 DMA 读请求到 L2 SRAM 中的存储器位置,控制器检查以决定包含请求存储器位置的行是否被 L1D Cache 缓存。在这个例子中,我们已经假设它是被 Cache 缓存的。然而,如果它没有被 Cache 缓存,不需要进一步的操作发生,且外围设备会完成读访问。

(3) 如果该行被 Cache 缓存,L2 控制器发送一个 snoop-read 命令给 L1D。snoop 首先检查以确定相应的行是否为 Dirty。如果不是,外围设备允许完成读访问。

(4) 如果 Dirty 位被设置,snoop-read 导致数据被直接传给 DMA,无须把它写到 L2 SRAM。这正是这个例子的情况,因为我们假设内核已经把数据写到输出缓冲了。

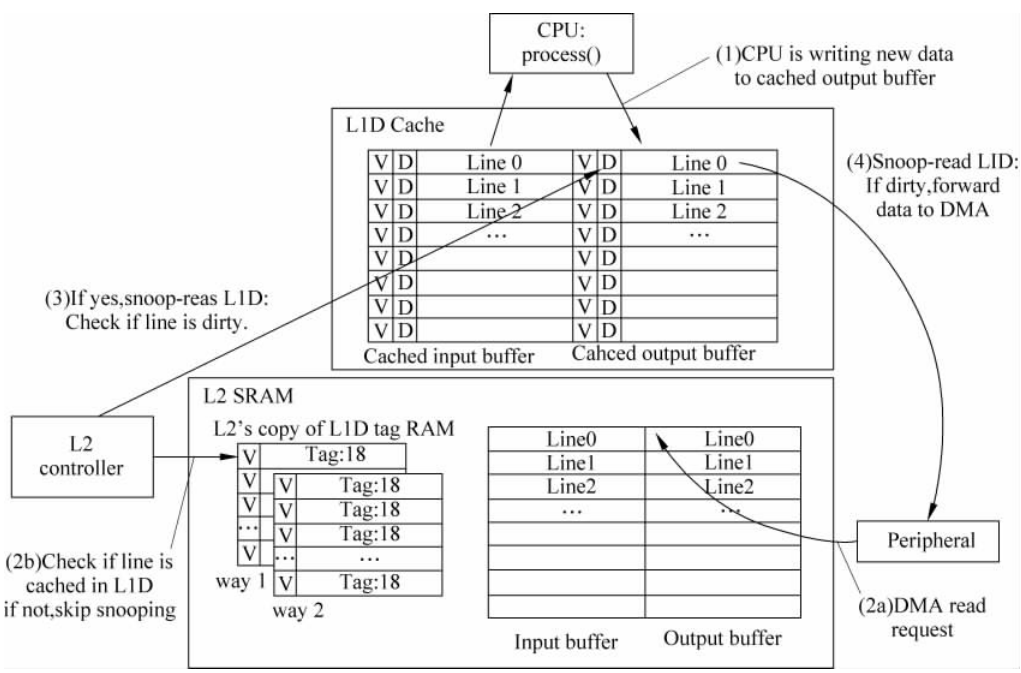

图 3.10 DMA 读访问 L2 一致性问题

2. L2 SRAM 双缓冲例子

假设数据从一个外围设备读入、处理并写到另一个外围设备,这是一个典型的信号处理应用场景,数据流如图 3.11 所示。当内核正在对一对缓冲(如 InBuffA 和 OutBuffA)的数据进行处理,外围设备使用另外一对缓冲(InBuffB 和 OutBuffB)正在写、读数据,这样 DMA

数据传输可以与核处理并行执行。示例中，假设 InBuffA 已经被外围设备填充，流程如下。

（1）当内核正在处理 InBuffA 中的数据，InBuffB 正在被填充。InBuffA 的行被分配到 L1D。数据正在被内核处理，并被通过写缓冲写到 OutBuffA（注意 L1D 只是 read-allocate 的）。

（2）当外围设备正在用新数据填充 InBuffA，第二个外围设备正从 OutBuffA 读，并且内核正在处理 InBuffB。对于 InBuffA，L2 Cache 控制器通过 snoop-writes 自动向前传输数据到 L1D。对于 OutBuffA，由于没有被 L1D Cache 缓存，没有必要进行 snoop 操作。

（3）缓冲又被交换，一直下去。

对每个 Cache 缺失，为了得到最高的回报（对被 Cache 缓存的数据而言），使 L2 SRAM 缓冲中放入多条 L1D Cache Line 可能是有益的。伪码如示例 3.3 所示，显示一个双缓冲如何设计实现。

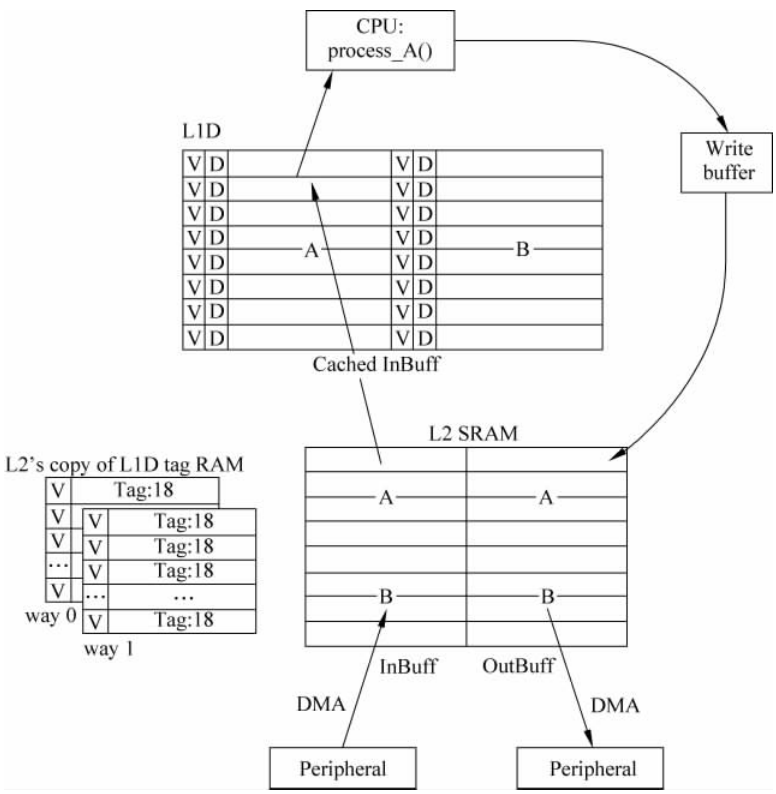

图 3.11　DMA 读、写访问 L2 一致性问题

【示例 3.3】 L2 SRAM DMA 双缓冲代码。

```
for (I = 0; i <(DATASIZE/BUFSIZE) - 2; i += 2)
{
    /* ---------------------------------------------------- */
    /* InBuffA -> OutBuffA Processing */
    /* ---------------------------------------------------- */
    <DMA_transfer(peripheral, InBuffB, BUFSIZE)>
    <DMA_transfer(OutBuffB, peripheral, BUFSIZE)>
    process(InBuffA, OutBuffA, BUFSIZE);
```

```
    /* ------------------------------------------------------ */
    /* InBuffB -> OutBuffB Processing */
    /* ------------------------------------------------------ */
    <DMA_transfer(peripheral, InBuffA, BUFSIZE)>
    <DMA_transfer(OutBuffA, peripheral, BUFSIZE)>
    process(InBuffB, OutBuffB, BUFSIZE);
}
End of 示例
```

3.9.2 在外部存储器和 Cache 之间维持一致性

考虑相同的双缓冲情景，但是缓冲位于外部存储器。因为 Cache 控制器在这种情况下不会自动维持一致性，程序员需要考虑一致性的问题。内核从外围设备读取数据并进行处理，之后通过 DMA 写数据到另外一个外围设备。但是现在数据传输通过 L2 Cache，如图 3.12 所示。假设传输已经发生了，InBuff 和 OutBuff 都被 Cache 缓存进 L2 Cache，并且

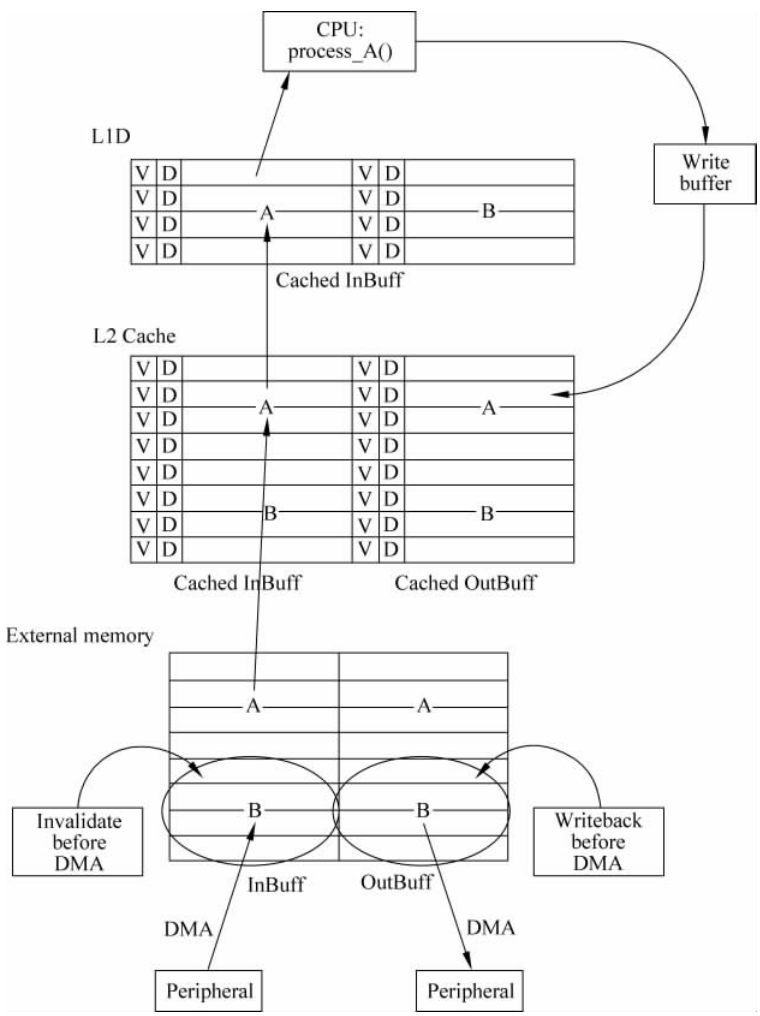

图 3.12 DMA 读、写访问外部存储器一致性问题

被 Cache 缓存进 L1D。更进一步假设内核已经完成对 InBuffB 的处理，结果填充了 OutBuffB，并正要开始处理 InBuffA 的行，即将开始引入新数据传输到 InBuffB，同时 OutBuffB 中的结果也要传到外围设备。为了维持一致性，在 DMA 传输开始前，所有映射外部存储器输入缓冲的 L1D 和 L2 Cache 的行需要被 invalidate。采用这个方法，当下次从外部存储器输入缓冲读数据时，内核会重新分配这些行。相似地，在 OutBuffB 被传输给外围设备前，数据首先必须从 L1D 和 L2 Cache 被写回到外部存储器。通过执行一个 writeback 操作，数据写回操作就被执行了。这是必要的，内核写数据只写到 OutBuffB 存储器位置的 Cache 缓存备份中，而数据可能还保存在 L1D 和 L2 Cache 中。

CSL(Chip Support Library)提供一组程序用于发起需要的 Cache 一致性操作。外部存储器中缓冲的起始地址和字节数需要被指定：

```
Cache_invL2(InBuffB, BUFSIZE, Cache_WAIT);
Cache_wbL2(OutBuffB, BUFSIZE, Cache_WAIT);
```

如果 Cache_WAIT 被使用，程序等待直到操作完成，这是推荐的做法。如果 Cache_NOWAIT 被使用，程序发起操作并立即返回。这允许内核继续执行程序，同时一致性操作在后台执行。

然而，需要小心，必须确保内核没有访问 Cache 控制器正在操作的地址。因为这可能导致非预期的结果。为了确保一致性操作执行完，程序 Cache_wait()可以在 DMA 传输被发起之前使用。

示例 3.4 中的伪码准确展示了 Cache 一致性调用和 DMA 传输必须发生的顺序。

【示例 3.4】 外部存储器 DMA 双缓冲代码。

```
for (i = 0; i <(DATASIZE/BUFSIZE) - 2; i += 2)
{
    /* ---------------------------------------------------------- */
    /* InBuffA -> OutBuffA Processing */
    /* ---------------------------------------------------------- */
    Cache_InvL2(InBuffB, BUFSIZE, Cache_WAIT);
    <DMA_transfer(peripheral, InBuffB, BUFSIZE)>
    Cache_wbL2(OutBuffB, BUFSIZE, Cache_WAIT);
    <DMA_transfer(OutBuffB, peripheral, BUFSIZE)>
    process(InBuffA, OutBuffA, BUFSIZE);
    /* ---------------------------------------------------------- */
    /* InBuffB -> OutBuffB Processing */
    /* ---------------------------------------------------------- */
    Cache_InvL2(InBuffA, BUFSIZE, Cache_WAIT);
    <DMA_transfer(peripheral, InBuffA, BUFSIZE)>
    Cache_wbL2(OutBuffA, BUFSIZE, Cache_WAIT);
    <DMA_transfer(OutBuffA, peripheral, BUFSIZE)>
    process(InBuffB, OutBuffB, BUFSIZE);
}
End of 示例
```

在一致性操作之外，所有 DMA 缓冲是 L2 Cache Line 对齐的，而且是 Cache Line 大小的整数倍。可以按照如下操作实现：

```
#pragma DATA_ALIGN(InBuffA, Cache_L2_LINESIZE)
#pragma DATA_ALIGN(InBuffB, Cache_L2_LINESIZE)
#pragma DATA_ALIGN(OutBuffA,Cache_L2_LINESIZE)
#pragma DATA_ALIGN(OutBuffB,Cache_L2_LINESIZE)
unsigned char InBuffA [N * Cache_L2_LINESIZE];
unsigned char OutBuffA[N * Cache_L2_LINESIZE];
unsigned char InBuffB [N * Cache_L2_LINESIZE];
unsigned char OutBuffB[N * Cache_L2_LINESIZE];
```

或者，CSL 宏 Cache_ROUND_TO_LINESIZE(Cache，element count，element size)可以被用来自动向上取整对齐队列到下一个 Cache Line 尺寸倍数的位置。第一个参数为 Cache 的类型，可以为 L1D、L1P 或 L2。

数组定义可以看起来如下：

```
unsigned char InBuffA [Cache_ROUND_TO_LINESIZE(L2, N, sizeof(unsigned char)];
unsigned char OutBuffA[Cache_ROUND_TO_LINESIZE(L2, N, sizeof(unsigned char)];
unsigned char InBuffB [Cache_ROUND_TO_LINESIZE(L2, N, sizeof(unsigned char)];
unsigned char OutBuffB[Cache_ROUND_TO_LINESIZE(L2, N, sizeof(unsigned char)];
```

3.9.3 对 L2 Cache 一致性操作使用指导

表 3.13 为 C66x 器件可用的 L2 Cache 一致性操作概览。注意这些操作总是操作在 L1P 和 L1D 上，即使 L2 Cache 被禁用。表 3.13 诠释如下。

(1) 首先，Cache 控制器操作在 L1P 和 L1D 上；

(2) 然后，一致性操作在 L2 Cache 上被执行。

表 3.13 L2 Cache 一致性概览

范围	一致性操作	CSL 命令	L2 Cache 上的操作	L1D Cache 上的操作	L1P Cache 上的操作
有效范围	Invalidate L2	Cache_invL2 (start address，byte count，wait)	所有在范围内的行 invalidate（任何 Dirty 数据被忽略）	所有在范围内的行 invalidate（任何 Dirty 数据被忽略）	所有在范围内的行 invalidate
	Writeback L2	Cache_wbL2 (start address，byte count，wait)	在范围内的 Dirty 行写回，所有行保持 valid	在范围内的 Dirty 行写回，所有行保持 valid	无
	Writeback Invalidate L2	Cache_wbInvL2 (start address，byte count，wait)	在范围内的 Dirty 行写回。所有范围内的行 invalidate	在范围内的 Dirty 行写回。所有范围内的行 invalidate	所有在范围内的行 invalidate
所有 L2 Cache	Writeback All L2	Cache_wbAllL2 (wait)	所有 L2 中的 Dirty 行写回。所有行保持有效	所有范围内的行 invalidate，所有 L1D 的 Dirty 行写回。所有行保持 valid，L1D snoop-invalidate	无
	Writeback Invalidate All L2	Cache_wbInvAllL2 (wait)	所有 L2 中的 Dirty 行被写回。所有 L2 的行 invalidate	所有在 L1D 中的 Dirty 行被写回。所有在 L1D 的行 invalidate	所有在 L1P 的行 invalidate

注意：一个行被 Cache 缓存进 L1P 或 L1D，未必被 Cache 缓存进 L2。一个行可能被从 L2 驱逐，不一定被从 L1P 或 L1D 驱逐。

重要提示：尽管起始地址和字节计数被指定，Cache 控制器总是操作在整个行上。因而，为了维持一致性，数组必须是：

(1) L2 Cache Line 倍数大小；

(2) 在 L2 Cache Line 分界处对齐。

一条 L2 Cache Line 长 128 字节。Cache 控制器操作在所有被指定地址范围涉及到的行上。注意最大字节数可以被指定为 4×65535 字节(在一些 C66x 器件上最大为 4×65408 字节，参考器件手册)，也就是说，一个 L2 Cache 操作可以操作最多 256KB。如果被操作的外部存储器缓存更大，多个 Cache 操作必须被执行。如果核与 DMA(或其他外部实体)分享一个可 Cache 缓存的外部存储空间区域，才需要用户发起 L2 Cache 一致性操作。也就是说，在以下场景需要一致性操作，当内核读被 DMA 写的数据或 DMA 读被内核写的数据时，最安全的规则就是，在任何 DMA 传输对外部存储器读或写之前，执行一个 Writeback-Invalidate All。然而，这种做法的缺点是，可能需要更多的 Cache Line 被操作，而导致更大的开销。一个精准的方法更加有效，首先，它只需要操作在那些真实包含共享缓冲的 Cache Line 上；其次，以下三种场景可以被区别对待，如表 3.14 所示。

表 3.14　L2 Cache 一致性操作

场　　景	需要的一致性操作
DMA/Other 读的数据被内核写	在 DMA/Other 开始读之前 Writeback L2
DMA/Other 写的数据(代码)将被内核读	在 DMA/Other 开始写之前 Invalidate L2
DMA/Other 修改被内核写的数据之后又将被内核读回	在 DMA/Other 开始写之前 Writeback-Invalidate L2

在场景 3，DMA 可能修改被核写的数据且数据被内核读回。例如，内核在一个外围设备写到缓冲前初始化存储器(如把它清 0)。在 DMA 开始之前，被核写的数据需要被传到外部存储器并且缓冲必须被 Invalidate。

3.9.4　对 L1 Cache 一致性操作使用指导

表 3.15 和表 3.16 显示 C66x 器件可用的 L1 Cache 一致性操作概览。

表 3.15　L1D Cache 一致性操作

范围	一致性操作	CSL 命令	L1D Cache 上的操作
有效范围	Invalidate L1D	Cache_invL1d (start address, byte count, wait)	所有范围内的行 invalidate (任何 Dirty 数据被忽略)
	Writeback L1D	Cache_wbL1d (start address, byte count, wait)	范围内的 Dirty 行被写回。所有行保持 valid
	Writeback Invalidate L1D	Cache_wbInvL1d (start address, byte count, wait)	范围内的 Dirty 行写回。所有范围内的行 invalidate
所有 L1D Cache	Writeback All L1D	Cache_wbAllL1d (wait)	所有在 L1D 的 Dirty 行被写回。所有行保持 valid
	Writeback Invalidate All L1D	Cache_wbInvAllL1d (wait)	所有在 L1D 的 Dirty 行被写回。所有行 invalidate

表 3.16 列出了 L1P Cache 一致性操作。

表 3.16 L1P Cache 一致性操作

范围	一致性操作	CSL 命令	L1D Cache 上的操作
有效范围	Invalidate L1P	Cache_invL1p(start address, byte count, wait)	所有范围内的行 invalidate
所有 L1P Cache	Invalidate All L1P	Cache_wbInvAllL1p (wait)	所有 L1P 的行 invalidate

注意：尽管一个起始地址和一个字节计数被指定，Cache 控制器操作总是在整个行上。因而，为维护一致性的目的，数组必须是：

(1) L1D Cache Line 大小的倍数；

(2) 对齐 L1D Cache Line 分界线。

一个 L1D Cache Line 是 64 字节。Cache 控制器操作在所有指定地址范围涉及到的行上。注意最大字节数可以被指定为 4×65535。表 3.17 中列出了 Cache 一致性操作必须被执行的场景。

表 3.17 需要 L1 一致性操作的场景

场 景	需要的一致性操作
DMA/Other 写代码到 L2 SRAM 将要被内核执行	在内核开始执行前 Invalidate L1P
内核修改在 L2 SRAM 或外部存储器的代码，该代码将被内核执行	在内核开始执行前 Invalidate L1P 和 Writeback-Invalidate L1D

3.10 片上 Debug 支持

C66x DSPs 提供片上 Debug 功能用于 Debug Cache 一致性问题。C66x 存储器系统允许仿真直接访问个别 Cache 并报告 Cache 状态信息(Valid、Dirty、LRU 位)。

用户可以通过 CCS IDE 中 Memory 窗口的显示功能，获得 Debug 能力。如果怀疑出现 Cache 一致性问题，可以遵循以下步骤：首先，确保排除任何不可预测的、影响内核访问一致性的操作，这需要先排除除了 Cache 一致性以外的其他原因。其次，确保缓冲对齐 L2 Cache Line 边界以减少虚假地址(False Addresses)。为了达到这个目的，存储器窗口提供可视的 Cache Line 边界标记，帮助用户很容易确定是否对齐。

下一步，确保正确使用 Cache 一致性操作，如下：

(1) 在完成 invalidate 一致性操作之后暂停内核执行，但是必须在第一次 DMA 写访问之前。

(2) 核实此刻缓冲中没有行是 Dirty 的。为了检查该信息，用户可以使能存储器分析功能(通过属性窗口，任何 Dirty 行会被显示成醒目的字体格式)。

(3) 继续内核执行。

(4) 在第一次内核读之前再次暂停内核。

(5) 确定缓冲仍然是 invalidate 并且包含期望的新数据。如果存在问题或数据被 Cache 缓存，用户可以使用 Cache 旁路复选框去观察在外部存储器中的数据内容。

3.11 在运行中改变 Cache 配置

3.11.1 禁用外部存储器 Cache 功能

在 Cache 功能被使能后，通常是不需要禁用外部存储器 Cache 功能的。然而，如果确实需要，就必须考虑。如果 MAR 中 PC 位被从 1 变到 0，已经被 Cache 缓存的外部存储器地址保持 Cache 关系，并且访问那些地址仍为命中。如果外部存储器地址在 L2 缺失，MAR 位是唯一的参考(这包括 L2 是全 SRAM 的情况，因为没有 L2 Cache，这也可以被解释为一个 L2 缺失)。如果在各个外部存储器地址空间中所有地址被设置成不可 Cache 缓存，首先需要被 write back 和 invalidate。

3.11.2 在运行中改变 Cache 尺寸

在运行中改变 Cache 尺寸可能对一些应用是有必要的，例如有的任务把 L2 作为 SRAM，将程序、一些全局变量等都放在 L2 SRAM 更有利；有的程序把 L2 大部分作为 Cache 更有利。

当 Cache 尺寸被改变时，需要按照一定的步骤执行以保证正确性。改变 L2 Cache 尺寸的具体步骤如表 3.18 所示，该步骤适用于 L1P 和 L1D Cache。

表 3.18 为 L1P、L1D 和 L2 改变 Cache 尺寸的步骤

切换到	执　　行
更多 Cache (更少 SRAM)	(1) 用 DMA 或 copy 方式将需要的代码/数据移出将被转换为 Cache 的 SRAM (2) 等待步骤(1)完成 (3) 改变 Cache 大小，使用 Cache_setL1pSize()、Cache_setL1dSize()或 Cache_setL2Size()
更少 Cache (更多 SRAM)	(1) 减小 Cache 大小，使用 Cache_setL1pSize()、Cache_setL1dSize()或 Cache_setL2Size() (2) DMA 或拷贝回任何需要的代码/数据 (3) 等待步骤(2)完成

3.12 优化 Cache 性能

3.12.1 Cache 性能特征

Cache 性能多半依赖于重用 Cache Line。访问没有被 Cache 缓存的存储器中的一个行会导致内核出现阻塞周期。只要这个行被保留在 Cache 中，随后对该行的访问不会导致任何阻塞。因而，该行在被从 Cache 驱逐前越经常被使用，阻塞周期越少。因此，优化一个应用的 Cache 性能的一个重要目标是最大化 Cache Line 重用这可以通过恰当的代码和数据的存储器布局，以及改变内核存储器访问顺序来实现。

为了执行这些优化，用户必须熟悉 Cache 存储器架构，熟悉在 Cache 存储器中独有的特征，如行大小、关联性、容量、替换机制、读/写分配、缺失流水和写缓冲。

3.12.2 阻塞情况

在C66x器件上最常见的阻塞情况如下。

(1) 交叉路径阻塞(Cross Path Stall)

当一个指令试图通过一个交叉路径去读一个在之前周期被更新的寄存器,一个阻塞周期被引入。编译器在任何可能的时候试图自动避免这些阻塞。

(2) L1D读和写命中

内核访问在L1D SRAM或Cache命中通常不会导致阻塞,除非与其他请求者存在访问冲突。访问优先级被带宽管理设置控制。

(3) L1D Cache写命中

内核写那些在L1D Cache命中的区域通常不会导致阻塞。然而,在高速率情况下,一个写命中流使之前干净的(Clean)Cache Line变成Dirty,可以导致阻塞周期。原因是一个Tag更新缓冲,其排队缓冲clean-to-dirty转换到L1D Tag RAM的L2拷贝(这也称为影子tag RAM,被用于snoop Cache一致性协议)。

(4) L1D Bank冲突

L1D存储器被组织成8×32字节块。并行访问都命中L1D的同一个bank,导致1个周期阻塞。

(5) L1D读缺失

由于L2 SRAM、L2 Cache或外部存储器的行分配,阻塞周期被引入。L1D读缺失可以被以下情况加长。

① L2 Cache读缺失:数据必须先从外部存储器取出,阻塞周期数取决于特定的器件和外部存储器类型。

② L2访问/bank冲突:L2每次只可以服务一个请求,访问优先级别由带宽管理设置控制。L2请求者包括L1P(Line Fills)、L1D(Line Fills、Write Buffer、Tag Update Buffer、Victim Buffer)、IDMA或EDMA及Cache一致性操作。

③ L1D Write Buffer Flush:如果写缓冲包含数据并且一个读缺失发生,在L1D读缺失被服务之前,写缓冲首先完全排干。为保持写之后紧跟一个读操作的适当的顺序,这是需要的。通过L2访问/bank冲突和L2 Cache写缺失(the Write Buffer Data Misses L2 Cache),写缓冲排干可以被拉长。

④ L1D Victim缓冲写回:如果Victim缓冲包含数据并且一个读缺失发生,在L1D读缺失被服务之前,内容首先被写回L2。对于写后紧随一个读,保持正确的顺序是必要的。写回可以被L2访问/bank冲突加长。

连续的、并行的缺失会部分重叠,假如没有任何以上阻塞加长的状况并且两个并行、连续的缺失不是对同一Set。

(6) L1D写缓冲满

如果一个L1D写缺失发生,并且写缓冲满,阻塞发生直到一个条目有效。写缓冲排干可以被以下情况加长。

① L2 Cache读缺失:数据必须首先被从外部存储器取出。阻塞周期数量取决于特定

器件和外部存储器类型。

② L2 访问/bank 冲突：每次 L2 只可以服务一个请求。访问的优先级由带宽管理设置管理。

③ L1P 读命中：在 L1P SRAM 或 Cache 命中，内核访问通常不会导致阻塞，除非一个访问与其他请求者冲突或者到 L1P ROM 的访问具有等待状态。访问优先级由带宽管理设置控制。L1P 请求者包含核程序访问、IDMA、EDMA 和 Cache 一致性操作。

(7) L1P 读缺失

对于来自 L2 SRAM、L2 Cache 和外部存储器的行分配，阻塞周期被引入。L1P 读缺失阻塞可以被以下情况加长。

① L2 Cache 读缺失：首先数据必须从外部存储器取出。阻塞周期数取决于特定器件和外部存储器。

② L2 访问/块冲突：每次 L2 只可以服务一个请求。访问优先级别由带宽管理设置控制。L2 请求者包含 L1P (Line Fills)、L1D (Line Fills、Write Buffer、Tag Update Buffer、Victim Buffer)、IDMA 或 EDMA 和 Cache 一致性操作。

连续的缺失会部分交叠，假如没有任何一个以上阻塞加长情况发生。

图 3.13 显示了 C66x 存储器结构，其中会有所有重要特征的详细描述。

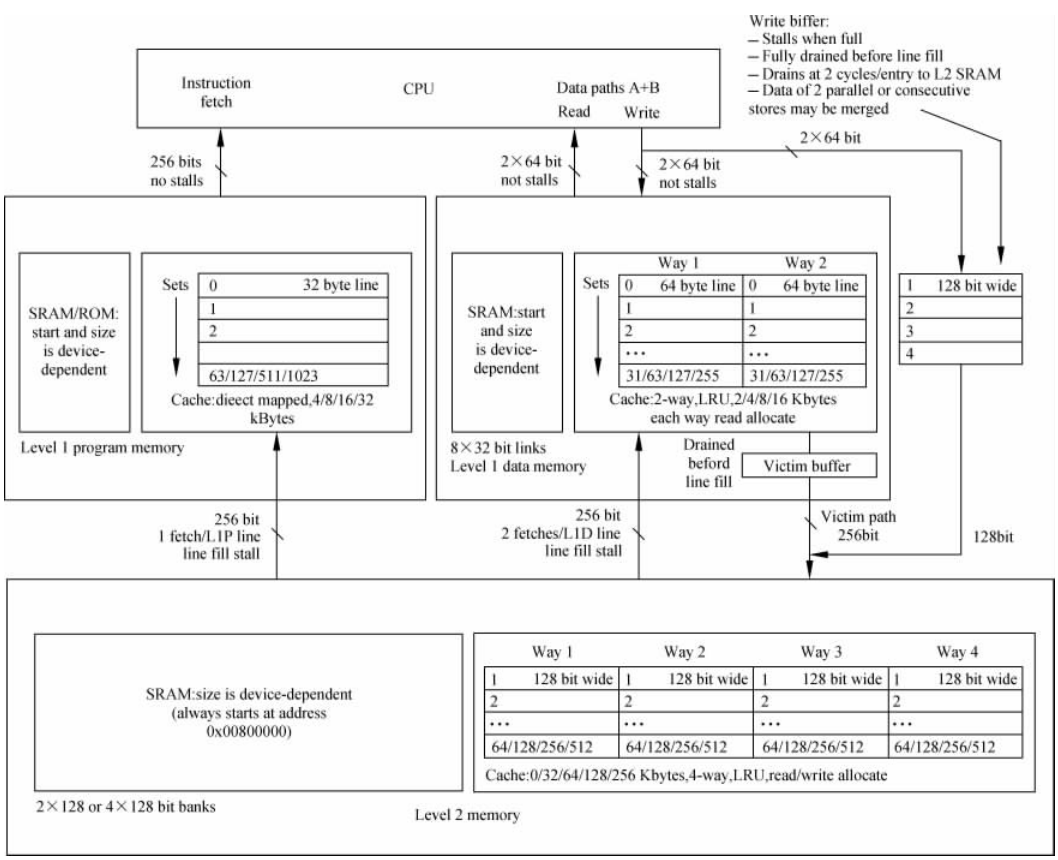

图 3.13　C66x Cache 存储器结构

3.12.3 优化技术概览

优化技术聚焦于 L1 Cache 的效率。因为 L1 特征(容量、关联性、行大小)比 L2 Cache 特征更严格,优化 L1 几乎肯定意味着 L2 Cache 也被高效地使用,而只优化 L2 Cache,则没有很多好处。对于应用中的通用部分,具有很大不可预期的存储器访问,推荐使用 L2 Cache。对时间严苛的信号处理算法,L1 和 L2 SRAM 必须被使用。使用 EDMA 或 IDMA,数据可以被直接流入 L1 SRAM;或使用 EDMA,流入 L2 SRAM。然后,存储器访问可以针对 L1 Cache 优化。有两个重要途径来减少 Cache 间接成本。

1. 减少 Cache 缺失数(在 L1P、L1D 和 L2 Cache)

减少 Cache 缺失数可以通过以下途径获得:

(1) 最大化 Cache 重用。

访问在一条 Cache Line 的所有存储器位置。

在一条 Cache Line 中相同存储器位置必须被重用,越经常越好。或者相同数据被重读或新数据被写到已经被 Cached 的位置,这样随后的读操作会命中。

(2) 只要 Cache Line 还在重用,避免驱逐。

如果数据被分配进存储器,当它被访问时,对应 Cache 路数没有被超出,驱逐可以被阻止(如果比 Cache 可用路数更多的行被映射到同一 Set,路数就被超出)。

如果这不可能,通过在时间上分开访问那些导致更多驱逐的地址,驱逐可以被延长。

同样,可以让行在一个被控制的方式被驱逐,依赖 LRU 替换机制,那样只有行不再被使用时才被驱逐。

2. 减少每个缺失阻塞数

通过充分利用缺失流水实现。

对于优化 Cache 性能,采用 top-down 方式是一个好的策略。在应用级开始,再到过程级,并且如有必要可以考虑算法级优化。应用级优化方法倾向于直接应用,典型地对全局性能提升具有更高的影响。如果必须,使用更低级别优化方法,可以执行微调。

3.12.4 应用级优化

在应用级和系统级,为了实现更佳的 Cache 性能,以下建议是非常重要的。

1. 分配数据流到外部存储器或 L1/L2 SRAM

对于 DSP 内核与一个外围设备或协处理器进行数据流交互的场景,使用 DMA 传送数据流,建议在 L1 或 L2 SRAM 中分配数据缓冲。数据流在 L1 或 L2 存储器分配缓冲,具有以下优点。

(1) L1 和 L2 SRAM 更接近内核,因而,可以减少延迟。如果缓冲被分配在外部存储器,数据会首先通过 DMA 从外围设备写到外部存储器,之后被 L2 Cache 缓存,最后在到达核之前还需要被 L1D Cache 缓存。

(2) 对于通过 Cache 控制器到 L2 SRAM 的数据访问,Cache 一致性自动维护。如果缓冲被分配在外部存储器,用户必须手动执行 L2 Cache 一致性操作,小心地维持数据一致性。在有些情况下,由于存储器容量限制,缓冲可能必须分配到外部存储器。

(3) 由于一致性操作,不会产生额外的延迟。延时可以被认为是添加到用于处理缓冲数据需要的时间。在一个典型的双缓冲机制中,选择缓冲的尺寸时就必须考虑。

对于快速原型应用,应用 DMA 双缓冲机制被认为消耗太长时间,应该尽量避开,分配所有代码和数据在外部存储器,使用 L2 作为全部 Cache 可能会是一个恰当的方法。

一旦正确的应用功能被验证,存储器管理瓶颈和关键的算法可以被确定和被优化。

2. 使用 L1 SRAM

C66x 器件提供 L1D 和 L1P SRAM,可能被用作代码和数据,对 Cache 损失敏感,如:

(1) 性能关键的代码和数据;

(2) 代码和数据被很多算法共享;

(3) 代码和数据被经常访问;

(4) 函数具有大的代码尺寸或大的数据结构;

(5) 数据结构因不规律的访问会使 Cache 更低效;

(6) 流缓冲(如 L2 很小的器件,最好配置为 Cache)。

由于 L1 SRAM 大小是有限的,因此需要很小心地决定什么代码和数据应该分配在 L1 SRAM。分配大量的 L1 SRAM 可能需要减少 L1 Cache 尺寸,对于代码和数据在 L2 和外部存储器的情况,这意味着更低的性能。

L1 SRAM 尺寸可以保持更小,如果代码和数据可以根据需要被复制到 L1 SRAM,使用代码、数据覆盖。IDMA 可被用来快速地从 L2 SRAM 寻找代码或数据。如果代码、数据要从外部存储器寻找,EDMA 必须被使用。然而,非常频繁的寻找可能会导致比用缓冲更高的代价。因而在 SRAM 和 Cache 尺寸之间,需要一个权衡。

3. 区分信号处理和常规处理代码

在一个应用中,区分信号处理类型和常规处理类型可能非常有益。常规处理通常包含控制流和条件分支,这些代码没有呈现多少并行性,而且执行依赖很多条件,处理的过程通常是不可预测的。

也就是说,数据存储器访问大多数是随机的,访问程序存储器是线性的,具有很多分支,这使得优化更加困难。因而,在 L2 SRAM 足够保留整个应用的代码和数据的情况下,推荐分配常规代码和相关的数据到外部空间并且允许 L2 Cache 去处理存储器访问。这使得对性能严苛的信号处理代码,有更多的 L2 存储空间可用。对于不可预测的常规代码类型,L2 Cache 应当被设置成尽可能大。Cache 可以被配置在 32～256KB。

DSP 代码和数据可能从被分配到 L2 SRAM 或 L1 SRAM 中获益。分配进 L2 SRAM 减少间接 Cache 损失,并给用户更多对存储器访问的控制,因为只有 L1 Cache 被包含,其行为更加容易去分析。在内核访问数据的方式上,允许用户修改一些算法或改变数据结构,以提供更加 Cache 友好的存储器访问形式。分配进 L1 SRAM 消除任何全部 Cache,并且除了 bank 冲突,不需要存储器优化。

3.12.5 过程级优化

随着数据和函数被分配进存储器的方式以及函数被调用的方式的改变,过程级优化需要被考虑。那些基于线性的存储器模型实现的算法(例如 FIR 滤波等),不需要对单个

算法进行优化。只有当通过算法访问的数据结构优化产生更高效的 Cache 使用时，算法才需要被优化。在多数情况下，过程级优化是有效的。除了一些算法（如 FFT），为了利用 Cache，其算法结构必须被修改。一个 Cache 优化的 FFT 在 C66x DSP 库（DSPLIB）中提供。

过程级优化的目标是减小 Cache 缺失的数量和一个缺失相关的阻塞周期。通过减少被 Cache 缓存的存储器数量和重用已经被 Cache 缓存的行，可以减少缺失的数量。可以通过避免驱逐和写入到预分配的行实现重用。通过利用缺失流水，一个缺失的阻塞周期可以被减少。

我们可以区分以下三种不同读缺失场景：

（1）所有工作集的数据、代码都纳入到 Cache（按照定义没有容量缺失），但是冲突缺失发生。通过在存储器中连续性分配代码或数据，冲突缺失可以被减少。

（2）数据集比 Cache 大、连续分配，并且没有重用。冲突缺失发生，但是没有容量缺失（因为数据没有被重用）。冲突缺失可以被减少，例如通过交叉 Cache Set。

（3）数据集比 Cache 更大，容量缺失（因为一些数据被重用）并且冲突缺失发生。通过将数据的 Set 分开并且每次只处理一个 Set，冲突和容量缺失可以被减少。这个方法指的是分割数据组，并且一次处理一个组。

采用链式（chain）处理，除了 chain 中的第一个算法为强制性缺失，在链式处理中一个算法的结果成为下一个算法的输入，这样减少 Cache 缺失的机会。

以下介绍一些应用场景中，过程级 Cache 优化设计的经验。

1. 通过选择相应的数据类型减少存储器带宽需求

数据类型选择需要确保存储器效率。例如，如果数据长度最大是 16bit，应该被声明为 short 类型而不是 integer。这减半了数组对存储器的需求，也减少了强制缺失因子为 2。典型地，接受新数据类型只需要算法中的一个小改变。因为更小的数据容器可以允许 SIMD 优化被编译器执行，算法可以执行得更快。

2. 链式处理

链式处理通常的流程是一个算法的结果作为下一个算法的输入。如果算法操作与链式处理流程不匹配（也就是说，结果放置的数组和输入不同），会产生较大的 Cache 缺失损失。例如，输入数组被分配进 L1D，但是输出通过写缓冲被传送到下一个更低存储器级（L2 或外部存储器）。然后，当下一个算法读该数据时，又承受缺失的代价。相反，如果第一个算法的输出被写到 L1D，然后数据可以从 Cache 中直接被重用，则无须引起 Cache 阻塞。对链式处理，有很多可能的配置。链式处理概念示意如图 3.14 所示。

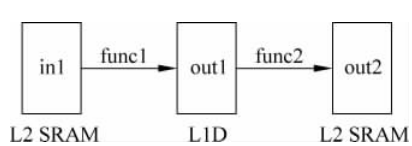

图 3.14 链式处理示意图

3. 避免 L1P 冲突缺失

在这个读缺失场景，所有工作集代码适合 Cache（定义没有容量缺失），但是冲突缺失发生。存储器地址映射到相同 Set 并且没有包含在同一 Cache Line 会互相驱逐。

编译和连接不会考虑 Cache 冲突，在执行过程中不适当的存储器布局可能导致冲突缺失。通常，这可以通过在存储器中连续分配在一些本地时间窗内访问的代码解决。

考虑示例 3.5 的代码：假设 function_1 和 function_2 已经被连接器放置，并且它们在 L1P 中交叠，如图 3.15 所示。当 function_1 第一次被调用，它被分配进 L1P 导致 3 个缺失(1)。一个紧接着调用的 function_2 导致其代码被分配在 L1P，导致 5 个缺失(2)。这也会驱逐 function_1 的部分代码(Cache Line 3 和 4)，因为这些行在 L1P 交叠(3)。当在下一个循环，function_1 又被调用，这些行必须被调进 L1P，却又将被 function_2 驱逐。因而，对于所有随后的循环，每个函数调用导致两个缺失，每个循环总共 4 个 L1P 缺失。这些类型缺失被称为冲突缺失。通过分配两个函数代码到不冲突的 Set，这些冲突可以被完全避免。最直接的方式是在存储器连续放置两个程序代码。

注意：也可以移动 function_2 到任何与 function_1 没有 Set 冲突的位置，这也可以阻止驱逐。然而，第一个方法有优势，用户不必担心绝对地址位置，只需简单改变函数在存储器中的顺序。

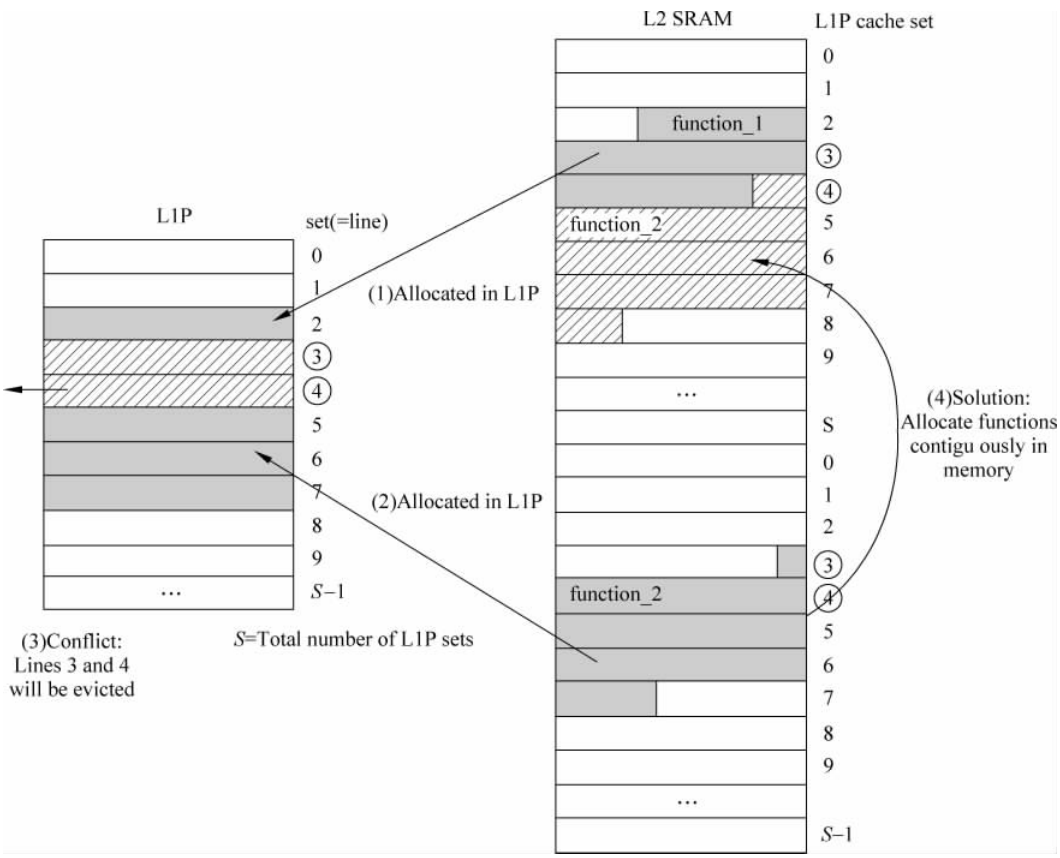

图 3.15　避免 L1P 驱逐

【示例 3.5】 L1P 冲突。

```
for (i = 0; i < N; i++)
{
    function_1();
    function_2();
}
End of 示例
```

注意：采用代码产生工具5.0(CCS 3.0)或以后版本，为了强制一个指定的连接顺序，GROUP编译指示必须被使用。

有以下两种方法用于在存储器连续分配函数。

(1) 使用编译器选项-mo

使用编译器选项-mo，放置每个C和线性汇编函数到自身的独立段中，汇编函数必须用.sect指令放置到段中。

在这个例子中，段名是.text:_function_1和.text:_function_2。现在，连接命令文件可以被指定为：

```
…
SECTIONS
{
    .cinit > L2SRAM
    .GROUP > L2SRAM
    {
        .text:_function_1
        .text:_function_2
        .text
    }
    …
}
```

连接器会按照GROUP声明中指定的顺序连接所有段。在这个例子中，function_1代码接着的是function_2，然后接着的是分配在该段的其他函数，在源代码中不需要被改变。然而，小心使用-mo编译器优化选项，可能导致总的代码量增大，因为任何包含代码的段会在32字节分界线对齐。

注意，连接器只能放置完整段，而不是分配存在于相同段的每个函数。如果预编译的库或对象文件多个函数在一个段或没有用-mo编译，没有办法重新分配每个函数到不同的段，除非重新编译库。`

(2) 使用SECTION编译指示

为了避免使用-mo的缺点，通过使用SECTION编译指示，只有那些需要连续放置的函数可以被分配到单独段中。

```
#pragma
CODE_SECTION 在函数定义前:
#pragma CODE_SECTION(function_1,".funct1")
#pragma CODE_SECTION(function_2,".funct2")
void function_1(){…}
void function_2(){…}
```

连接命令文件可以被指定如下：

```
…
SECTIONS
{
    .cinit > L2SRAM
    .GROUP > L2SRAM
```

```
    {
        .funct1 .funct2
        .text
    }
    …
}
```

在同一个循环中或者在一些时段中反复被调用的函数，可以考虑被重新安排。

如果 Cache 容量不足以保持一个循环的所有函数，如果为了获得代码重用、不被驱逐，循环必须被分割。这可能提高对临时缓冲的存储器需求来保持输出数据。假设合并的 function_1 和 function_2 的代码尺寸比 L1P 尺寸更大。在示例 3.6，代码循环被分割，这样两个函数可以从 L1P 重复地被执行，从而显著地减少缺失。然而，临时缓冲 tmp[ ]必须保持从每次调用 function_1 的所有中间结果，导致了一定的数据依赖性。

【示例 3.6】 代码分割从 L1P 执行。

```
for (i = 0; i < N; i++)
{
    function_1(in[i], tmp[i]);       //代码分割可以在 L1P 执行
}
for (i = 0; i < N; i++)
{
    function_2(tmp[i], out[i]);      //代码分割可以在 L1P 执行
}
End of 示例
```

4. 避免 L1D 冲突缺失

在这个读缺失场景，所有工作集的数据与 Cache 相适应(定义为没有容量缺失)，但是冲突缺失发生。下面首先说明 L1D 冲突缺失如何被产生，描述如何通过在存储器中连续分配数据减少冲突缺失。

在一个直接映射的 Cache(如 L1P)，如果这些地址不在同一 Cache Line 上，它们会被相互驱逐。然而，在一个 2 路组相联的 L1D Cache，两个冲突行可以被保存在 Cache 中，无须被驱逐。只有当又有第三个存储器位置被分配并映射到相同 Set，之前分配的行的其中一个必须被驱逐(会被驱逐的行由最近使用规则 LRU 决定)。

编译器和连接器不考虑 Cache 冲突。在执行期间，不适当的存储器布局可能导致冲突缺失。通过改变存储器数组布局，可以最大限度地避免被驱逐。通常，对于同一时间窗的数据访问，通过在存储器中连续分配可以避免驱逐。然而，在代码和数据之间不同的是：L1D 是一个 2 路组相联 Cache，L1P 是直接映射的。这意味着在 L1D，两个数据队列可以映射到相同 Set 且在同一时间还驻留在 L1D。以下例子说明了 L1D Cache 的相关性。

假设每个数组是 1/4 总 L1D 容量，这样所有 4 个数组可以装进 L1D。然而，假如我们不考虑存储器布局和声明数组，如下：

```
short in1 [N];
short other1 [N];
short in2 [N];
short other2 [N];
```

```
short w1 [N];
short other3 [N];
short w2 [N];
```

数组 other1、other2 和 other3 在相同应用中被其他程序使用。假设数组以它们被声明的顺序被连续分配在段.data 中，因为在 L1D 中每一路是总容量的一半，所有存储器位置到相同 Set 是一路大小的间隔。在这个例子中，in1、in2、w1 和 w2 都映射到 L1D 中相同的 Set，如图 3.16(a)所示(图中 S 为 L1D set 的总数目)。注意，这只是许多可能的配置中的一种。确切地配置取决于第一个数组起始地址 in1 和 LRU 位置的状态(这决定了行被分配到那一路)。然而，就 Cache 性能而言，所有配置是相同的。

如图 3.16 所示为点乘例子中数组映射到 L1D set 的情况。

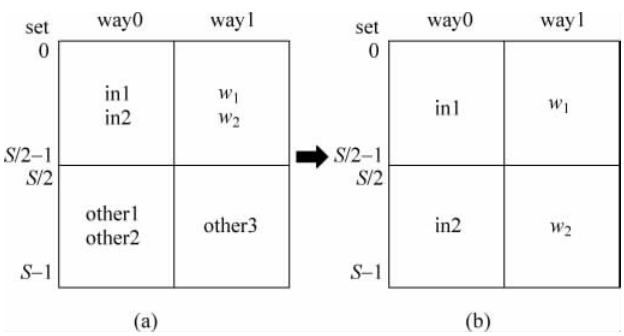

图 3.16 点乘例子中数组映射到 L1D set 的情况

为了减少读缺失，我们可以在存储器中分配连续数组，如下所示：

```
short in1 [N];
short in2 [N];
short w1 [N];
short w2 [N];
short other1 [N];
short other2 [N];
short other3 [N];
```

把程序使用的数组放在一块定义，现在所有数组(in1、in2、w1 和 w2)可以装进 L1D，如图 3.16(b)所示。注意，由于连接器存储器分配规则，不能总是确信数组的连续定义被分配在相同的段(如 const 数组会被替换在.const 段，而不在.data 段)。因而，数组必须被分配到用户定义段中。

此外，数组对齐一条 Cache Line 边界用于减少额外的缺失。注意，可能有必要在不同的存储器 bank 对齐数组，以避免 bank 冲突，例如：

```
#pragma DATA_MEM_BANK(in1, 0)
#pragma DATA_MEM_BANK(in2, 0)
#pragma DATA_MEM_BANK(w1, 2)
#pragma DATA_MEM_BANK(w2, 2)
```

利用缺失流水可以更多减少 Cache 缺失阻塞。在 L1D 中用户预分配所有数组 in1、in2、w1 和 w2。

因为所有数组在存储器中被连续地分配，调用一个 touch 程序就足够：

```
touch(in1, 4 * N * sizeof(short));
r1 = dotprod(in1, w1, N);
r2 = dotprod(in2, w2, N);
r3 = dotprod(in1, w2, N);
r4 = dotprod(in2, w1, N);
```

5. 避免 L1D 崩溃(Thrashing)

在这个读缺失场景，数据 Set 比 Cache 大，是连续分配的，但是数据没有被重用。冲突缺失发生，但是没有容量缺失(因为数据没有被重用)。本节描述如何减少冲突缺失，例如通过交叉分配 Cache Set。如果多于两个读缺失发生在相同 Set，在所有数据访问前驱逐一个行，导致 L1D 崩溃。假设所有数据在存储器中连续地分配，如果被访问总数据集比 L1D 容量更大，这种情况可能发生。通过在存储器连续分配数据集并填充数组强制形成一个交叉映射到 Cache Set，这些冲突缺失可以被完全减少。

例如，如果被分配在存储器中的三个数组 w[ ]、x[ ]和 h[ ]都被分配到相同 Set，L1D 崩溃发生。可以看到任何时候一个数组元素试图被读，都不包含在 L1D。考虑第一个循环迭代，所有三个数组被访问并导致到相同组的三个读缺失。通过在存储器中连续分配数据组并填充数组来强制交叉映射到 Cache Set，冲突缺失可以被完全消除，如图 3.17 所示。

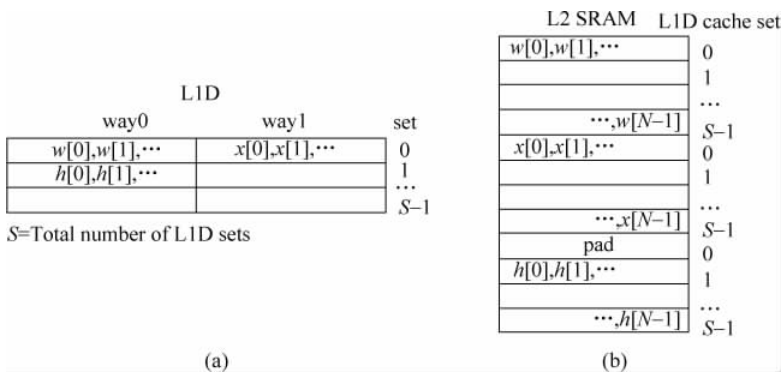

图 3.17　存储器分配添加 pad

例如：

```
#pragma DATA_SECTION(w, ".mydata")
#pragma DATA_SECTION(x, ".mydata")
#pragma DATA_SECTION(pad,".mydata")
#pragma DATA_SECTION(h, ".mydata")
#pragma DATA_ALIGN (w, Cache_L1D_LINESIZE)
short w [N]; short x [N];
char pad [Cache_L1D_LINESIZE];
short h [N];
```

连接命令文件可以被指定：

```
...
SECTIONS
{
    GROUP > L2SRAM
    {
        .mydata:w
        .mydata:x
        .mydata:pad
        .mydata:h
    }
    ...
}
```

6. 避免容量缺失

在这个读缺失场景，数据被重用，但是数据集比 Cache 大，导致容量和冲突缺失。通过分割数据集并每次处理一个子块，这些缺失可以被消除（每个子块比 Cache 小）。这个方法被指为分块（block）或平铺（tiling）。例如用一个参考向量和 4 个不同输入向，点乘程序被调用 4 次：

```
short in1[N];
short in2[N];
short in3[N];
short in4[N];
short w [N];
r1 = dotprod(in1, w, N);
r2 = dotprod(in2, w, N);
r3 = dotprod(in3, w, N);
r4 = dotprod(in4, w, N);
```

假设每个数组是 L1D 容量的两倍。对于首次调用 in1[ ]和 w[ ]，预期为强制缺失。对于剩下的调用，对于 in2[ ]、in3[ ]和 in4[ ]，预期为强制缺失；但是会更愿意从 Cache 重用 w[ ]。然而，在每次调用后，因为容量不够，w[ ]起始已经被 w[ ]的结束替代。对于 w[ ]，后续调用后又遭受缺失。

如果处理完 in1[ ]的 1/4 并开始处理 in2[ ]，可以重用刚刚分配进 Cache 的 w[ ]元素。同样的，在计算完另一个 $N/4$ 输出，跳过去处理 in3[ ]和最后到 in4[ ]。在那之后，开始计算对于 in1[ ]第二个 $N/4$ 输出，…。这个结构代码会看起来像如下这样：

```
for (i = 0; i < 4; i++)
{
    o = i * N/4; dotprod(in1 + o, w + o, N/4);
    dotprod(in2 + o, w + o, N/4);
    dotprod(in3 + o, w + o, N/4);
    dotprod(in4 + o, w + o, N/4);
}
```

上述点乘例子中的存储器分配如图 3.18 所示。

通过利用缺失流水，我们可以进一步减少读缺失的数目。一旦在循环开始，touch 循环被用来分配 w[ ]。然后在每个点乘调用前，需要的数组被分配：

```
for (i = 0; i < 4; i++)
{
  o = i * N/4;
  touch(w + o, N/4 * sizeof(short));
  touch(in1 + o, N/4 * sizeof(short));
  dotprod(in1 + o, w + o, N/4);
  touch(w + o, N/4 * sizeof(short));
  touch(in2 + o, N/4 * sizeof(short));
  dotprod(in2 + o, w + o, N/4);
  touch(w + o, N/4 * sizeof(short));
  touch(in3 + o, N/4 * sizeof(short));
  dotprod(in3 + o, w + o, N/4);
  touch(w + o, N/4 * sizeof(short));
  touch(in4 + o, N/4 * sizeof(short));
  dotprod(in4 + o, w + o, N/4);
}
```

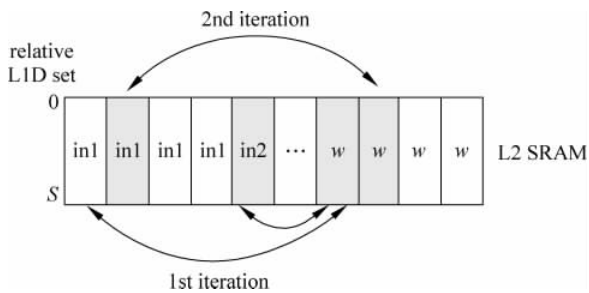

图 3.18　点乘例子存储器分配图

重要提醒：只要两条在相同 Set 的行总是以相同顺序被访问，LRU 策略自动保留命中的行(在这个例子中为 $w[]$)。如果访问的顺序改变，这个 LRU 行为就不能保证，这可以通过在分配下一个 in[]前重新 touch w[ ]实现。

这样强制 $w[\ ]$成为 MRU 并被保护以免被驱逐。外部 touch 不会消耗很多周期，因为没有 Cache 缺失发生，也就是说需要(行数/2+16)周期。

在这个例子中，数组 $w[\ ]$和 in[ ]必须和不同的存储器 bank 对齐以免 bank 冲突。

```
#pragma DATA_SECTION(in1, ".mydata")
#pragma DATA_SECTION(in2, ".mydata")
#pragma DATA_SECTION(in3, ".mydata")
#pragma DATA_SECTION(in4, ".mydata")
#pragma DATA_SECTION(w,".mydata") /* this implies #pragma DATA_MEM_BANK(w, 0) */
#pragma DATA_ALIGN(w, Cache_L1D_LINESIZE) short w [N]; /* avoid bank conflicts */
#pragma DATA_MEM_BANK(in1, 2)
short in1[N];
short in2[N];
short in3[N];
short in4[N];
```

7. 避免写缓冲相关的阻塞

L1D 写缓冲可以导致另外的阻塞。通常，写缺失不会导致阻塞，因为写的数据直接通

过写缓冲传递给较低层存储器(L2 或外部存储器)。然而,写缓冲的深度被限制在 4 个条目。为了确保每个 128 位宽的条目效率更高,写缓冲把对顺序地址的连续写缺失合并到同一条目。如果写缓冲满了并且另外一个写缺失发生,内核阻塞直到缓冲中的一个条目可用。另外,在读缺失被处理前,一个读缺失会导致写缓冲被完全用光。确保正确的写之后读的顺序是有必要的(导致缺失的读操作可能访问还在写缓冲中的数据)。阻塞周期为:排干写缓冲的周期数加上正常的读缺失阻塞周期。

通过在 L1D Cache 中分配输出缓冲,写缓冲相关阻塞会很容易被避免。写会命中 L1D 而不是被传递给写缓冲。

3.12.6 C66x DSP Cache 一致性操作小结

在以上描述中,不存在硬件一致性协议,程序员负责保持 Cache 一致性。为了实现这个目的,C66x DSP 存储器控制器支持程序启动的 Cache 一致性操作。一致性操作包括以下几个。

Invalidate (INV):驱逐 Cache Line 并忽略数据;

Writeback (WB):写回数据,行保留在 Cache 中并被标记为 Clean;

Writeback-Invalidate (WBINV):写回数据并驱逐 Cache Line。

对于 L1P、L1D 以及 L2 Cache,这些操作是可用的。注意 L2 Cache 一致性操作总是首先操作在 L1P 和 L1D 上。

对于 C66x DSP 存储器系统,表 3.19 和表 3.20 列出了一致性矩阵。如果存在于 Cache 中的一个物理地址(L2 SRAM 或外部存储器)的拷贝在被一个源实体写访问时,一致性矩阵指示对于读访问数据如何使其被目的实体可见。

表 3.19 L2 SRAM Cache 一致性矩阵

源	目 的	在写访问时行的位置	
(写访问)	(读访问)	L1P Cache	L1D Cache
DMA	DMA	不需要操作,因为内在的一致性(L1P Cache 不会影响可见性)	L1D WB、INV 或 WBINV 以避免潜在地破坏新写的数据:在 DMA 写的时候,行必须不为 Dirty
	CORE 数据路径	不需要操作,因为内在的一致性(L1P Cache 不会影响可见性)	Snoop-write:写到 L2 SRAM 的数据直接传递给 L1D Cache
	CORE Fetch Path 内核取指令路径	L1P INV 用于可见性,在写之后核第一次取指令访问时,行必须被 invalid	L1D WB、INV 或 WBINV 以避免潜在的对新写的代码的破坏:在 DMA 写访问的时行必须不为 Dirty
CORE 数据路径	DMA	不需要操作,因为内在的一致性(L1P Cache 不会影响可见性)	Snoop-read:数据直接传递给 DMA,无须更新 L2 SRAM
	CORE 数据路径	不需要操作,因为内在的一致性(L1P Cache 不会影响可见性)	不需要操作,因为内在的一致性
	CORE Fetch Path 内核取指令路径	L1P INV 用于可见性:在写之后核第一次取指令访问时,行必须被 invalid	L1D WB 或 WBINV 用于可见性:在取指令访问发生前有新代码的 Dirty 行必须已经被写回

表 3.20 外部 SRAM Cache 一致性矩阵

源	目 的	在写访问时行的位置		
（写访问）	（读访问）	L1P Cache	L1D Cache	L2 Cache
DMA/Other	DMA/Other	不需要操作，因为内在的一致性（L1P Cache 不会影响可见性）	L1D WB、INV 或 WBINV 以避免潜在地破坏新写的数据：在 DMA/Other 写的时候，行必须不为 Dirty	L2 WB、INV 或 WBINV 以避免潜在地破坏新写的数据：在 DMA/Other 写的时候，行必须不为 Dirty
	CORE 数据路径	不需要操作，因为内在的一致性（L1P Cache 不会影响可见性）	L1D WB、INV 或 WBINV 以避免潜在地破坏新写的数据：在 DMA/Other 写的时候，行必须不为 Dirty L1D INV 或 WBINV 用于可见性：在写之后第一次核访问时，行必须为 invaild	L2 WB、INV 或 WBINV 以避免潜在地破坏新写的数据：在 DMA/Other 写的时候，行必须不为 Dirty L2 INV 或 WBINV 用于可见性：在写之后第一次核访问时，行必须为 invaild
	CORE Fetch Path 内核取指令路径	L1P INV 用于可见性：在写之后核第一次取指令访问时，行必须被 invalid	L1D WB、INV 或 WBINV 以避免潜在地破坏新写的代码：在 DMA/Other 写访问时，行必须不为 Dirty	L2 WB、INV 或 WBINV 以避免潜在地破坏新写的代码：在 DMA/Other 写访问时，行必须不为 Dirty L2 INV 或 WBINV 用于可见性：在写之后核第一次取指令访问时，行必须被 invalid
CORE 数据路径	DMA/Other	不需要操作，因为内在的一致性（L1P Cache 不会影响可见性）	L1D INV 或 WBINV 用于可见性：在 DMA/Other 读访问时，有新数据的 Dirty 行必须被写回	L2 WB 或 WBINV 用于可见性：在 DMA/Other 读访问发生时，有新数据的 Dirty 行已经被写回
	CORE 数据路径	不需要操作，因为内在的一致性（L1P Cache 不会影响可见性）	不需要操作，因为内在的一致性	不需要操作，因为内在的一致性
	CORE Fetch Path 内核取指令路径	L1P INV 用于可见性：在写之后核第一次取指令访问时，行必须被 invalid	L1D WB 或 WBINV 用于可见性：在内核取指令访问发生时，有新代码的 Dirty 行已经被写回。不需要操作，因为内在的一致性	不需要操作，因为内在的一致性

这可以通过多种方法获得，例如：

（1）使新数据到一个 Cache 或存储器，并对目标实体可见，具体操作包括 snoop-write、L1D WB/WBINV、L2 WB/WBINV；

(2) 使新数据直接到目的实体,具体操作为 snoop-read;

(3) 从 Cache 中移除 Cache 的拷贝,使得对于目的实体而言,存储器中保持新数据可见,具体操作为 L1P INV、L1D INV/WBINV、L2 INV/WBINV。

对于目的实体,使数据部分可见也是要确保数据没有被任何驱逐 Dirty 行破坏。如果由于某些原因写入的地址的 Cache 始终是 Dirty 状态,驱逐可以覆盖被其他实体写的数据。驱逐是一部分常规核存储器行为,并且不总是可预测的。在一致性矩阵中指出如何获得一致性。

最通用的场景是 DMA 写入、内核数据路径读(DMA-to-data)和内核数据路径写、之后 DMA 读(data-to-DMA)。对于 DMA 写入、内核取指令(DMA-to-fetch)情况的例子是代码覆盖,对于内核写、内核取指令(data-to-fetch)情况的例子是代码覆盖、拷贝加载代码(memcpy)和自我修改代码。DMA 写、DMA 读(DMA-to-DMA)是一个典型使用场景,例如,考虑数据被 DMA 写到一个外部地址空间,该空间被指定用于内核数据路径。如果地址被内核缓存进 Cache,而 DMA 写该地址时,首先任何通过潜在地驱逐 Dirty 行、破坏新数据的操作必须被避免;其次,由于数据在 L2 Cache“掩盖下”被写入,新写的数据对于内核必须可见(readable)。通过使行 Clean(通过 writeback 指令)或从 Cache 全部失效(通过 invalidate 指令),可以避免数据破坏。通过 invalidate 相应的地址可以获得数据可见性,从而一个内核读访问从外部存储器获得新数据,而不是 L2 中的旧数据。事实上,用户可能不需要像一致性矩阵指示的那样操作每个行。通过指定起始地址和长度,发起对一块地址的一致性操作。注意零散的内核访问可以降低一致性操作的效果。这里假定零散的内核访问不存在或被消除了。如果没有,那么一个零散访问可能潜在地重新分配或 Redirty 刚刚一致性操作的一个行,甚至是在 DMA 或其他访问期间。这个结果是不可预期的。为了确保一致性矩阵中的需求,提供如下重要的实践经验:

(1) 在最后一次写操作之后,发起块一致性操作,一致性操作需要在第一次访问该块前完成。

(2) 可见性的需求:“在写之后第一次核访问(读数据/取指令:read/fetch)时,行必须为 invaild”。如果没有虚假地址,并且在第一次写之前块一致性操作完成,也可以确保可见性。

(3) 避免数据破坏的需要:“在 DMA/Other 写的时候,行必须不为 Dirty”。在第一次 DMA/Other 的写访问前,如果块一致性操作完成,可以确保避免数据被破坏,但是多个主设备之间必须没有虚假地址依赖关系。

(4) 通过使用 invalidate 操作(没有 writeback),避免数据破坏(虚假地址必须被消除)。

一些可以简化的一致性操作如下:

(1) 必须假定一个地址被保留在所有 Cache,因为每个地址在哪里被保存通常是不被知道的。因而,对于一个给定的 source-destination 场景,多种一致性操作必须被执行。然而实际上,在外部存储器场合发起一个 L2 一致性操作是足够的。因为任何 L2 Cache 一致性操作隐含首先操作在 L1D 和 L1P。例外的情况是 data-to-fetch 路径场景,对于 L1D 和 L1P 不同的一致性操作需要被执行(**注意**:与外部存储地址一样,L2 SRAM 也必须执行相同的操作)。

(2) 如果可以确定 DMA/Other 不会写 Cache 中 Dirty 的行,在 DMA/Other 访问时

write back 或 Invalidate 相应的行是不需要的。

(3) 为了可见性以及避免数据破坏,在第一个写 DMA/Other 访问之前完成一个 INV 或 WBINV 操作,需要执行的两个一致性操作可以合成一个。

注意:该操作只在没有虚假地址时起作用。

图 3.19～图 3.23 显示了在每个场景用户发起的 Cache 一致性操作的正确时序。

图 3.19 所示的场景为外部存储器:DMA 写、CORE 读(数据)。

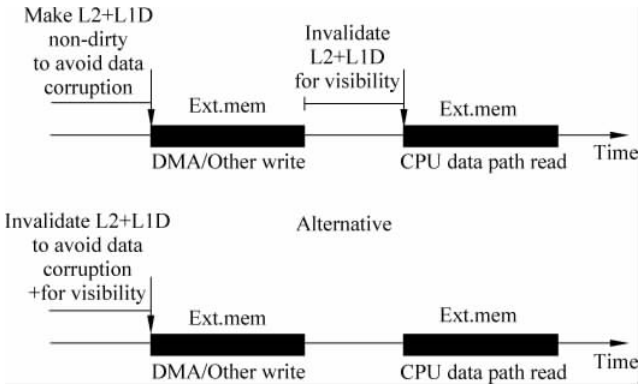

图 3.19 外部存储器:DMA 写、CORE 读(数据)

如图 3.20 所示为外部存储器:DMA 写、CORE 取指令(代码)场景。

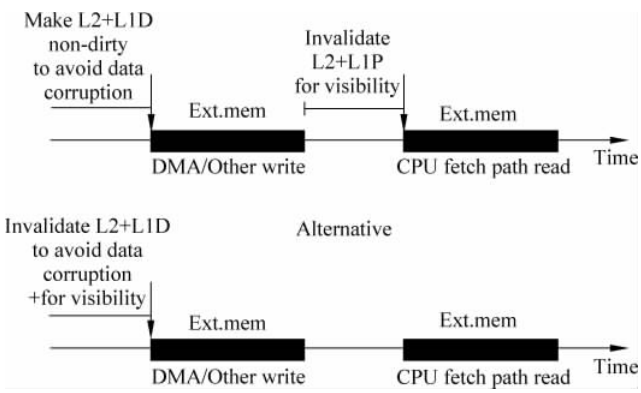

图 3.20 外部存储器:DMA 写、CORE 取指令(代码)

图 3.21 所示的场景为外部存储器:CORE 写、DMA 读(数据)。

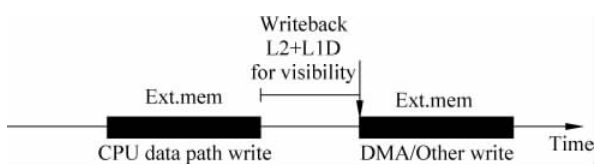

图 3.21 外部存储器:CORE 写、DMA 读(数据)

图 3.22 所示的场景为 L2 SRAM、外部存储器:CORE 写(数据)、CORE 取指令(代码)。

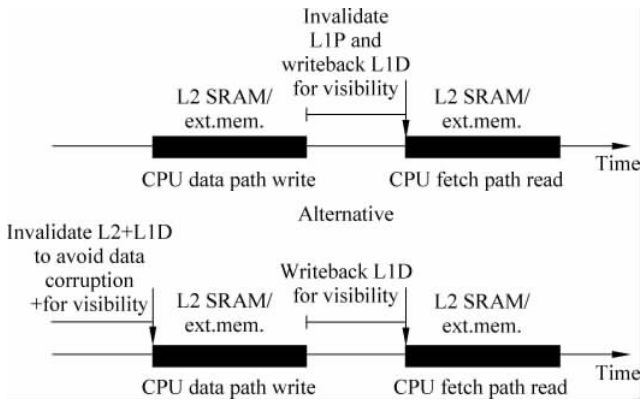

图 3.22 SRAM、外部存储器：CORE 写(数据)、CORE 取指令(代码)

图 3.23 所示的场景为 L2 SRAM：DMA 写、CORE 取指令(代码)。

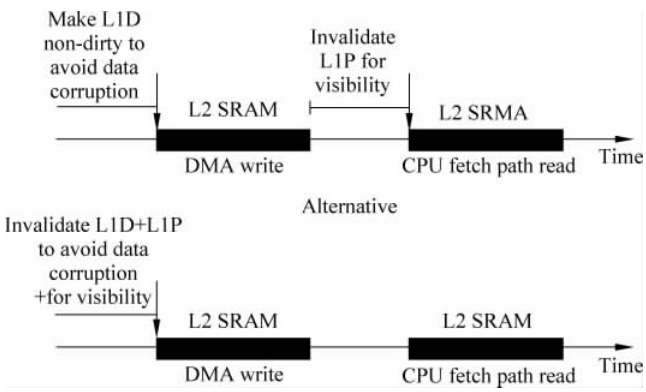

图 3.23 L2 SRAM：DMA 写、CORE 取指令(代码)

3.13 设计建议

3.13.1 消除虚假地址

在一致性矩阵中，假定每个行只包含打算操作的地址，没有打算操作的地址被认为是虚假地址(False Addresses)。如果它们存在，那么：

(1) 为内核数据可见性准备的一致性操作的影响可以被破坏。

在一致性矩阵中，注明的条件如果满足，可以确保数据的可见性，例如“在写之后的第一次读，取指令访问时行必须被 invalid”。然而，如果在虚假地址所在行已经 invalid 之后，内核因虚假地址关系而又访问了该行，这一行可能因为虚假地址读缺失导致又被 Cache 缓存，这样在写操作之前数据又为 invallid。

(2) 为消除潜在的对被 DMA/Other 新写入的数据的破坏准备的 Cache 一致性操作的影响可能被毁坏。

这种情况在一致性矩阵中标注为：“在 DMA/Other 些访问时，行必须不为 Dirty”。然而，在行已经被置成 Clean 或失效(通过 WB、INV 或 WBINV)后，如果内核写到 Cache 中的

虚假地址，可能该行又被弄成 Dirty。

(3) 如果这些虚假地址最近被核写，但是还没有写回(Writeback)到物理存储地址，使用 L1D INV 或 L2 INV 会导致数据丢失。

使用 WBINV 代替 INV 会避免这种数据破坏。因为控制内核访问到虚假地址非常困难，强烈推荐消除虚假地址。通过以 L2 Cache Line 边界大小对齐一个外部缓冲起始地址，并使其长度是 L2 Cache Line 大小(128 字节)的整数倍，就可以消除虚假地址。对于 L2 SRAM 地址，L1D Cache Line 大小(64 字节)可能被使用。对于内核数据路径相对于取指令(fetch)路径一致性情况，L1P Cache Line 尺寸(32 字节)可能被使用(不考虑 L2 SRAM 或外部存储器地址)。

3.13.2 数据一致性问题

在多核设计中，如果由于 Cache 功能经常导致数据一致性问题，可以根据需要按照表 3.19 和表 3.20 要求进行数据一致性操作。除了 Cache 会导致一致性问题外，预取也可能导致数据一致性问题，可以参见 2.3.6 节的方法进行预取数据一致性处理。