TMS320C66x
DSP Cache

User Guide

Literature Number: SPRUGY8
November 2010

Release History

Release	Date	Chapter/Topic	Description/Comments
1.0	November 2010	All	Initial Release

Contents

List of Tables

List of Figures

List of Examples

Preface

About This Manual

This user guide describes how the cache-based memory system of the C66x DSP can be efficiently used in DSP applications. The internal memory architecture of these devices is organized in a two-level hierarchy consisting of a dedicated program memory (L1P) and a dedicated data memory (L1D) on the first level. Accesses by the core to the these first level memories can complete without core pipeline stalls.

Notational Conventions

This document uses the following conventions:

- Commands and keywords are in **boldface** font.
- Arguments for which you supply values are in *italic* font.
- Terminal sessions and information the system displays are in `screen font`.
- Information you must enter is in **`boldface screen font`**.
- Elements in square brackets ([ ]) are optional.

Notes use the following conventions:

Note—Means reader take note. Notes contain helpful suggestions or references to material not covered in the publication.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

CAUTION—Indicates the possibility of service interruption if precautions are not taken.

WARNING—Indicates the possibility of damage to equipment if precautions are not taken.

Related Documentation from Texas Instruments

C66x CorePac User Guide	SPRUGW0
C66x CPU and Instruction Set Reference Guide	SPRUGH7

Trademarks

Code Composer Studio, TMS320C66x, and C66x are trademarks of Texas Instruments Incorporated.

All other brand names and trademarks mentioned in this document are the property of Texas Instruments Incorporated or their respective owners, as applicable.

Chapter 1

Introduction

This chapter describes the basic operation of memory caches and the operation of the TMS320C66x digital signal processor (DSP) two-level cache architecture.

1.1 Purpose of this User Guide

This user guide describes how the cache-based memory system of the C66x DSP can be efficiently used in DSP applications. The internal memory architecture of these devices is organized in a two-level hierarchy consisting of a dedicated program memory (L1P) and a dedicated data memory (L1D) on the first level. Accesses by the core to the these first level memories can complete without core pipeline stalls. Both L1P and L1D can be configured into SRAM and cache. If the data requested by the core is not contained in cache, it is fetched from the next lower memory level, L2 or external memory. A detailed technical description of the C66x memory architecture is given in *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x.

The following topics are covered in this user guide:

- The necessity of caches in high-performance DSPs (Chapter 1)
- General introduction into cache-based architectures (Chapter 1)
- Configuring and using the cache on C66x devices (Chapter 2)
- Maintaining cache coherence between different requestors (Chapter 2 and Appendix B)
- Linking code and data for increased cache efficiency (Chapter 3)
- Code-optimization techniques for increased cache efficiency (Chapter 3)

1.2 Chip Support Library (CSL)

This user guide makes references to the Chip Support Library (CSL). The CSL provides APIs for easy control of cache, DMA, and peripheral functions of a device. The CSL for your device either comes with the Code Composer Studio integrated development environment (IDE) or may be downloaded from www.ti.com. Note that cache APIs are also available through BIOS (version 5.21 or higher).

1.3 Cache Terms and Definitions

Table 1-1 lists the terms used throughout this document that relate to the operation of the C66x DSP two-level cache.

Table 1-1 Cache Terms and Definitions (Part 1 of 4)

Term	Definition
Allocation	The process of finding a location in the cache to store newly cached data. This process can include *evicting* data that is presently in the cache to make room for the new data.
Associativity	The number of *line frames* in each set. This is specified as the number of ways in the cache.
Capacity miss	A cache *miss* that occurs because the cache does not have sufficient room to hold the entire *working* set for a program. Compare with *compulsory miss* and *conflict miss*.
Clean	A cache *line* that is *valid* and that has not been written to by upper levels of memory or the core. The opposite state for a clean cache line is *dirty*.
Coherence	Informally, a memory system is coherent if any read of a data item returns the most recently written value of that data item. This includes accesses by the core and the DMA.
Compulsory miss	Sometimes referred to as a *first-reference miss*. A compulsory miss is a cache *miss* that must occur because the data has had no prior opportunity to be allocated in the cache. Typically, compulsory misses for particular pieces of data occur on the first access of that data. However, some cases can be considered compulsory even if they are not the first reference to the data. Such cases include repeated write misses on the same location in a cache that does not *write allocate*, and cache misses to noncacheable locations. Compare with *capacity miss* and *conflict miss*.
Conflict miss	A cache *miss* that occurs due to the limited *associativity* of a cache, rather than due to capacity constraints. A *fully-associative cache* is able to allocate a newly cached *line* of data anywhere in the cache. Most caches have much more limited associativity (see *set-associative cache*), and so are restricted in where they may place data. This results in additional cache misses that a more flexible cache would not experience.
Direct-mapped cache	A direct-mapped cache maps each address in the *lower-level memory* to a single location in the cache. Multiple locations may map to the same location in the cache. This is in contrast to a multi-way *set-associative cache*, which selects a place for the data from a set of locations in the cache. A direct-mapped cache can be considered a single-way set-associative cache.
Dirty	In a *writeback cache*, writes that reach a given level in the memory hierarchy may update that level, but not the levels below it. Therefore, when a cache *line* is *valid* and contains updates that have not been sent to the next lower level, that line is said to be dirty. The opposite state for a dirty cache line is *clean*.
DMA	Direct Memory Access. Typically, a DMA operation copies a block of memory from one range of addresses to another, or transfers data between a peripheral and memory. From a cache *coherence* standpoint, DMA accesses can be considered accesses by a parallel processor.
Eviction	The process of removing a *line* from the cache to make room for newly cached data. Eviction can also occur under user control by requesting a *writeback-invalidate* for an address or range of addresses from the cache. The evicted line is referred to as the *victim*. When a victim line is *dirty* (that is, it contains updated data), the data must be written out to the next level memory to maintain *coherency*.
Execute packet	A block of instructions that begin execution in parallel in a single cycle. An execute packet may contain between 1 and 8 instructions.
Fetch packet	A block of 8 instructions that are fetched in a single cycle. One fetch packet may contain multiple *execute packets*, and thus may be consumed over multiple cycles.
First-reference miss	A cache *miss* that occurs on the first reference to a piece of data. First-reference misses are a form of *compulsory miss*.
Fully-associative cache	A cache that allows any memory address to be stored at any location within the cache. Such caches are very flexible, but usually not practical to build in hardware. They contrast sharply with *direct-mapped caches* and *set-associative caches*, both of which have much more restrictive allocation policies. Conceptually, fully-associative caches are useful for distinguishing between *conflict misses* and *capacity misses* when analyzing the performance of a direct-mapped or set-associative cache. In terms of set-associative caches, a fully-associative cache is equivalent to a set-associative cache that has as many *ways* as it does *line frames*, and that has only one *set*.
Higher-level memory	In a hierarchical memory system, higher-level memories are memories that are closer to the core. The highest level in the memory hierarchy is usually the Level 1 caches. The memories at this level exist directly next to the core. Higher-level memories typically act as caches for data from *lower-level memory*.
Hit	A cache hit occurs when the data for a requested memory location is present in the cache. The opposite of a hit is a *miss*. A cache hit minimizes stalling, since the data can be fetched from the cache much faster than from the source memory. The determination of hit versus miss is made on each level of the memory hierarchy separately-a miss in one level may hit in a lower level.

Table 1-1 Cache Terms and Definitions (Part 2 of 4)

Term	Definition
Invalidate	The process of marking *valid* cache *lines* as invalid in a particular cache. Alone, this action discards the contents of the affected cache lines, and does not write back any updated data. When combined with a *writeback*, this effectively updates the next lower level of memory that holds the data, while completely removing the cached data from the given level of memory. Invalidates combined with writebacks are referred to as *writeback-invalidates*, and are commonly used for retaining *coherence* between caches.
Least Recently Used (LRU) allocation	For set-associative and *fully-associative caches*, least-recently used *allocation* refers to the method used to choose among *line frames* in a set when allocating space in the cache. When all of the *line frames* in the *set* that the address maps to contain *valid data*, the line frame in the set that was read or written the least recently (furthest back in time) is selected to hold the newly cached data. The selected line frame is then *evicted* to make room for the new data.
Line	A cache line is the smallest block of data that the cache operates on. The cache line is typically much larger than the size of data accesses from the core or the next higher level of memory. For instance, although the core may request single bytes from memory, on a read *miss* the cache reads an entire line's worth of data to satisfy the request.
Line frame	A location in a cache that holds cached data (one *line*), an associated *tag* address, and status information for the line. The status information can include whether the line is *valid*, *dirty*, and the current state of that line's *LRU*.
Line size	The size of a single cache *line*, in bytes.
Load through	When a core request misses both the first-level and second-level caches, the data is fetched from the external memory and stored to both the first-level and second-level cache simultaneously. A cache that stores data and sends that data to the upper-level cache at the same time is a load-through cache. Using a load-through cache reduces the stall time compared to a cache that first stores the data in a lower level and then sends it to the higher-level cache as a second step.
Long-distance access	Accesses made by the core to a noncacheable memory. Long-distance accesses are used when accessing external memory that is not marked as cacheable.
Lower-level memory	In a hierarchical memory system, lower-level memories are memories that are further from the core. In a C66x DSP system, the lowest level in the hierarchy includes the system memory below L2 and any memory-mapped peripherals.
LRU	Least Recently Used. See *least recently used allocation* for a description of the LRU replacement policy. When used alone, LRU usually refers to the status information that the cache maintains for identifying the least-recently used *line* in a set. For example, consider the phrase "accessing a cache line updates the LRU for that line."
Memory ordering	Defines what order the effects of memory operations are made visible in memory. (This is sometimes referred to as consistency.) Strong memory ordering at a given level in the memory hierarchy indicates it is not possible to observe the effects of memory accesses in that level of memory in an order different than program order. Relaxed memory ordering allows the memory hierarchy to make the effects of memory operations visible in a different order. Note that strong ordering does not require that the memory system execute memory operations in program order, only that it makes their effects visible to other requestors in an order consistent with program order.
Miss	A cache miss occurs when the data for a requested memory location is not in the cache. A miss may stall the requestor while the *line frame* is allocated and data is fetched from the next lower level of memory. In some cases, such as a core write miss from L1D, it is not strictly necessary to stall the core. Cache misses are often divided into three categories: *compulsory misses*, *conflict misses*, and *capacity misses*.
Miss pipelining	The process of servicing a single cache miss is pipelined over several cycles. By pipelining the miss, it is possible to overlap the processing of several misses, should many occur back-to-back. The net result is that much of the overhead for the subsequent misses is hidden, and the incremental stall penalty for the additional misses is much smaller than that for a single miss taken in isolation.
Read allocate	A read-allocate cache only allocates space in the cache on a read *miss*. A write miss does not cause an allocation to occur unless the cache is also a *write-allocate cache*. For caches that do not write allocate, the write data would be passed on to the next lower-level cache.
Set	A collection of *line frames* in a cache that a single address can potentially reside. A *direct-mapped cache* contains one line frame per set, and an N-way *set-associative cache* contains N line frames per set. A *fully-associative cache* has only one set that contains all of the line frames in the cache.
Set-associative cache	A set-associative cache contains multiple *line frames* that each *lower-level memory* location can be held in. When allocating room for a new *line* of data, the selection is made based on the *allocation* policy for the cache. The C66x devices employ a *least recently used allocation* policy for its set-associative caches.
Snoop	A method by which a *lower-level memory* queries a *higher-level memory* to determine if the higher-level memory contains data for a given address. The primary purpose of snoops is to retain coherency.

Table 1-1 Cache Terms and Definitions (Part 3 of 4)

Term	Definition
Tag	A storage element containing the most-significant bits of the address stored in a particular *line*. Tag addresses are stored in special tag memories that are not directly visible to the core. The cache queries the tag memories on each access to determine if the access is a *hit* or a *miss*.
Thrash	An algorithm is said to thrash the cache when its access pattern causes the performance of the cache to suffer dramatically. Thrashing can occur for multiple reasons. One possible situation is that the algorithm is accessing too much data or program code in a short time frame with little or no reuse. That is, its *working set* is too large, and thus the algorithm is causing a significant number of *capacity misses*. Another situation is that the algorithm is repeatedly accessing a small group of different addresses that all map to the same set in the cache, thus causing an artificially high number of *conflict misses*.
Touch	A memory operation on a given address is said to touch that address. Touch can also refer to reading array elements or other ranges of memory addresses for the sole purpose of allocating them in a particular level of the cache. A core-centric loop used for touching a range of memory in order to allocate it into the cache is often referred to as a touch loop. Touching an array is a form of software-controlled prefetch for data.
Valid	When a cache *line* holds data that has been fetched from the next level memory, that *line frame* is valid. The invalid state occurs when the line frame holds no data, either because nothing has been cached yet, or because previously cached data has been *invalidated* for whatever reason (*coherence* protocol, program request, etc.). The valid state makes no implications as to whether the data has been modified since it was fetched from the *lower-level memory*; rather, this is indicated by the *dirty* or *clean* state of the line.
Victim	When space is allocated in a set for a new *line*, and all of the *line frames* in the set that the address maps to contain valid data, the cache controller must select one of the *valid* lines to evict in order to make room for the new data. Typically, the *least-recently used* (LRU) line is selected. The line that is *evicted* is known as the victim line. If the victim line is dirty, its contents are written to the next lower level of memory using a *victim writeback*.
Victim Buffer	A special buffer that holds *victims* until they are *written* back. Victim *lines* are moved to the victim buffer to make room in the cache for incoming data.
Victim Writeback	When a *dirty* line is *evicted* (that is, a line with updated data is evicted), the updated data is written to the lower levels of memory. This process is referred to as a victim writeback.
Way	In a *set-associative cache*, each *set* in the cache contains multiple *line frames*. The number of line frames in each set is referred to as the number of ways in the cache. The collection of corresponding line frames across all sets in the cache is called a way in the cache. For instance, a 4-way set-associative cache has 4 ways, and each set in the cache has 4 line frames associated with it, one associated with each of the 4 ways. As a result, any given cacheable address in the memory map has 4 possible locations it can map to in a 4-way set-associative cache.
Working set	The working set for a program or algorithm is the total set of data and program code that is referenced within a particular period of time. It is often useful to consider the working set on an algorithm-by-algorithm basis when analyzing upper levels of memory, and on a whole-program basis when analyzing lower levels of memory.
Write allocate	A write-allocate cache allocates space in the cache when a write miss occurs. Space is allocated according to the cache's *allocation* policy (*LRU*, for example), and the data for the *line* is read into the cache from the next lower level of memory. Once the data is present in the cache, the write is processed. For a *writeback cache*, only the current level of memory is updated-the write data is not immediately passed to the next level of memory.
Writeback	The process of writing updated data from a *valid* but *dirty* cache *line* to a *lower-level memory*. After the writeback occurs, the cache line is considered *clean*. Unless paired with an *invalidate* (as in *writeback-invalidate*), the line remains valid after a writeback.
Writeback cache	A writeback cache will only modify its own data on a write *hit*. It will not immediately send the update to the next lower-level of memory. The data will be written back at some future point, such as when the cache *line* is *evicted*, or when the lower-level memory *snoops* the address from the *higher-level memory*. It is also possible to directly initiate a *writeback* for a range of addresses using cache control registers. A write hit to a writeback cache causes the corresponding line to be marked as *dirty*-that is, the line contains updates that have yet to be sent to the lower levels of memory.

Table 1-1 Cache Terms and Definitions (Part 4 of 4)

Term	Definition
Writeback-invalidate	A writeback operation followed by an invalidation. See *writeback* and *invalidate*. On the C66x devices, a writeback-invalidate on a group of cache *lines* only writes out data for *dirty* cache lines, but invalidates the contents of all of the affected cache lines.
Write merging	Write merging combines multiple independent writes into a single, larger write. This improves the performance of the memory system by reducing the number of individual memory accesses it needs to process. For instance, on the C66x device, the L1D write buffer can merge multiple writes under some circumstances if they are to the same double-word address. In this example, the result is a larger effective write-buffer capacity and a lower bandwidth impact on L2.
Write-through cache	A write-through cache passes all writes to the *lower-level memory*. It never contains updated data that it has not passed on to the lower-level memory. As a result, cache *lines* can never be *dirty* in a write-through cache.

1.4 Cache Differences Between C64x and C66x DSP

Readers who are familiar with the TMS320C64x DSP cache architecture may want to take note of features that are new or have changed for C66x DSPs. The features described in this chapter are listed below. For a complete list of new and changed features, see Appendix A on page A-1.

Memory sizes and types:

- On C66x devices, each L1D and L1P implement SRAM additionally to cache. The size of cache is user-configurable and can be set to 4K, 8K, 16K, or 32K bytes. The amount of available SRAM is device dependent and specified in the device-specific data manual. On C64x devices, only cache with a fixed size of 16K bytes is implemented.
- On C66x devices, the maximum possible size of L2 is increased. See the data manual for the actual amount of available L2 memory. L2 cache size configurations are the same as on C64x devices.

Write buffer:

- The width of the write buffer on C66x devices is increased to 128 bits; on C64x devices, the width is 64 bits.

Cacheability:

- The cacheability settings of external memory addresses (through the MAR bits) only affect L1D and L2 caches on C66x devices; that is, program fetches to external memory addresses are always cached in L1P, regardless of the cacheability setting. This is not the case on C64x devices, where the settings affects all caches, L1P, L1D, and L2.
- The cacheability control of external memory addresses covers the entire external address space on C66x devices. In contrast, on C64x devices, only a subset of the address space is covered.

Snooping protocol:

- The snooping cache coherence protocol on C66x devices directly forwards data to L1D cache and the DMA. C64x devices invalid and writeback cache lines to maintain coherence. The C66x snooping mechanism is more efficient since it eliminates cache miss overhead caused by invalidates.
- The snoop coherence protocol on C66x devices does not maintain coherence between L1P cache and L2 SRAM, as is the case on C64x devices. This is the responsibility of the programmer.

Cache coherence operations:

- On C66x devices, the L2 cache coherence operations always operate on L1P and L1D even if L2 cache is disabled. This is not the case on C64x devices, which requires the explicit use of L1 coherence operations.
- C66x devices support a complete set of range and global L1D cache coherence operations. In contrast, C64x devices support only L1D range invalidate and writeback–invalidate.
- On cache size changes, C66x devices automatically writeback–invalidate cache before initializing it with the new size. In contrast, C64x devices required an explicit writeback–invalidate to be issued by the programmer (however, this is handled as part of the CSL function).

- On C66x devices, L2 cache is non inclusive of L1D and L1P. This means that a line eviction from L2 does not cause the corresponding lines in L1P and L1D to be evicted. However, this is the case on C64x devices. The advantage of noninclusivity is that line allocations in L2 due to program fetches do not evict data from L1D cache, and line allocations in L2 due to data accesses do not evict program code from L1P. This helps reduce the number of cache misses.

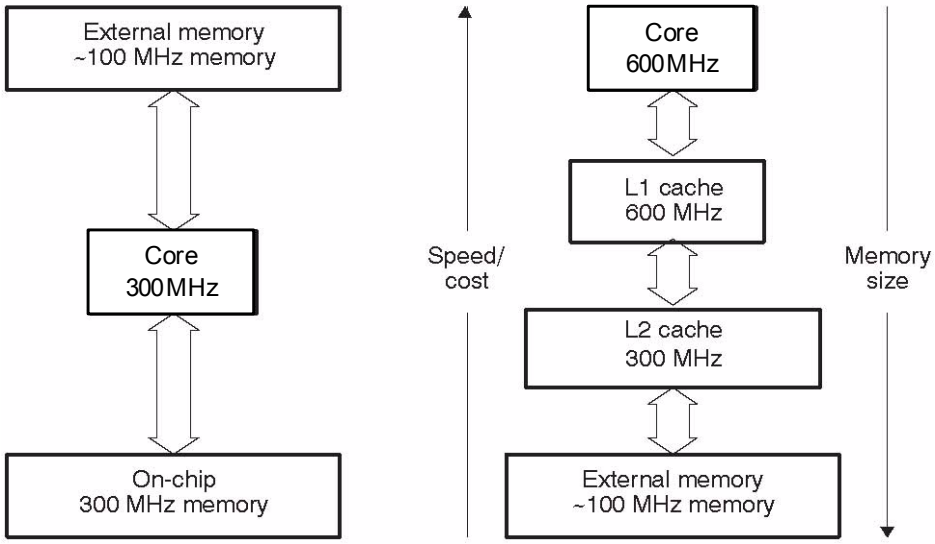
1.5 Why Use Cache?

From a DSP application perspective, a large amount of fast on-chip memory would be ideal. However, over the past years the performance of processors has improved at a much faster pace than that of memory. As a result, there is now a performance gap between core and memory speed. High-speed memory is available but consumes much more size and is more expensive compared with slower memory.

Consider the flat memory architecture shown on the left in Figure 1-1. Both core and internal memory are clocked at 300 MHz such that no memory stalls occur. However for accesses to the slower external memory, there will be core stalls. If the core clock was now increased to 600 MHz, the internal memory could only service core accesses every two core cycles and the core would stall for one cycle on every memory access. The penalty would be particularly large for highly optimized inner loops that may access memory on every cycle. In this case, the effective core processing speed would approach the slower memory speed. Unfortunately, today's available memory technology is not able to keep up with increasing processor speeds, and a same size internal memory running at the same core speed would be far too expensive.

The solution is to use a memory hierarchy, as shown on the right in Figure 1-1. A fast but small memory is placed close to the core that can be accessed without stalls. The next lower memory levels are increasingly larger but also slower the further away they are from the core. Addresses are mapped from a larger memory to a smaller but faster memory higher in the hierarchy. Typically, the higher–level memories are cache memories that are automatically managed by a cache controller. Through this type of architecture, the average memory access time will be closer to the access time of the fastest memory rather than to the access time of the slowest memory.

Figure 1-1 Flat Versus Hierarchical Memory Architecture



1.6 Principle of Locality

Caches reduce the average memory access time by exploiting the locality of memory accesses. The principle of locality assumes that if a memory location was referenced it is very likely that the same or a neighboring location will be referenced soon again. Referencing memory locations within some period of time is referred to as *temporal locality*. Referencing neighboring memory locations is referred to as *spatial locality*. A program typically reuses data from the same or adjacent memory locations within a small period of time. If the data is fetched from a slow memory into a fast cache memory and is accessed as often as possible before it is being replaced with another set of data, the benefits become apparent.

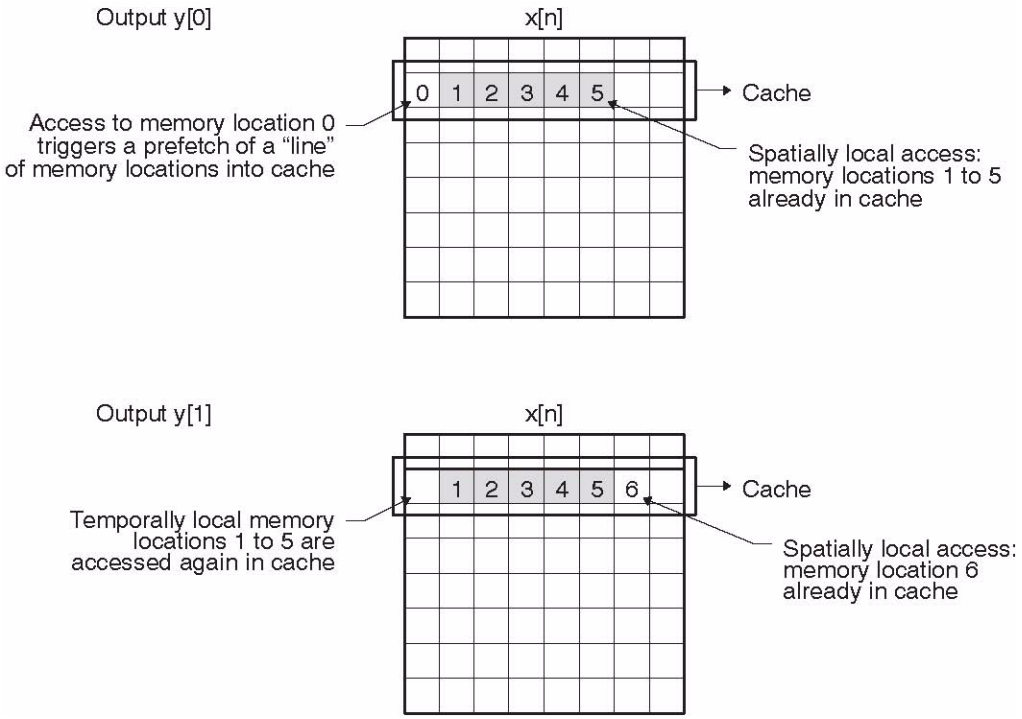
The following example illustrates the concept of spatial and temporal locality. Consider the memory access pattern of a 6-tap FIR filter. The required computations for the first two outputs y[0] and y[1] are: $y[0] = h[0] \times x[0] + h[1] \times x[1] + ... + h[5] \times x[5]$ $y[1] = h[0] \times x[1] + h[1] \times x[2] + ... + h[5] \times x[6]$

Consequently, to compute one output we have to read six data samples from an input data buffer x[ ]. Figure 1-2 shows the memory layout of this buffer and how its elements are accessed. When the first access is made to memory location 0, the cache controller fetches the data for the address accessed and also the data for a certain number of the following addresses into cache. This range of addresses is called a cache line. The motivation for this behavior is that accesses are assumed to be spatially local. This is true for the FIR filter, since the next five samples are required as well. Then all accesses will go to the fast cache instead of the slow lower-level memory.

Consider now the calculation of the next output, y[1]. The access pattern again is shown in Figure 1-2. Five of the samples are being reused from the previous computation and only one sample is new; but all of them are already held in cache and no core stalls occur. This access pattern exhibits high spatial and temporal locality: the same data that was used in the previous step is being used again for processing.

Cache builds on the fact that data accesses are spatially and temporally local. The number of accesses to a slower, lower-level memory are greatly reduced, and the majority of accesses can be serviced at core speed from the high-level cache memory.

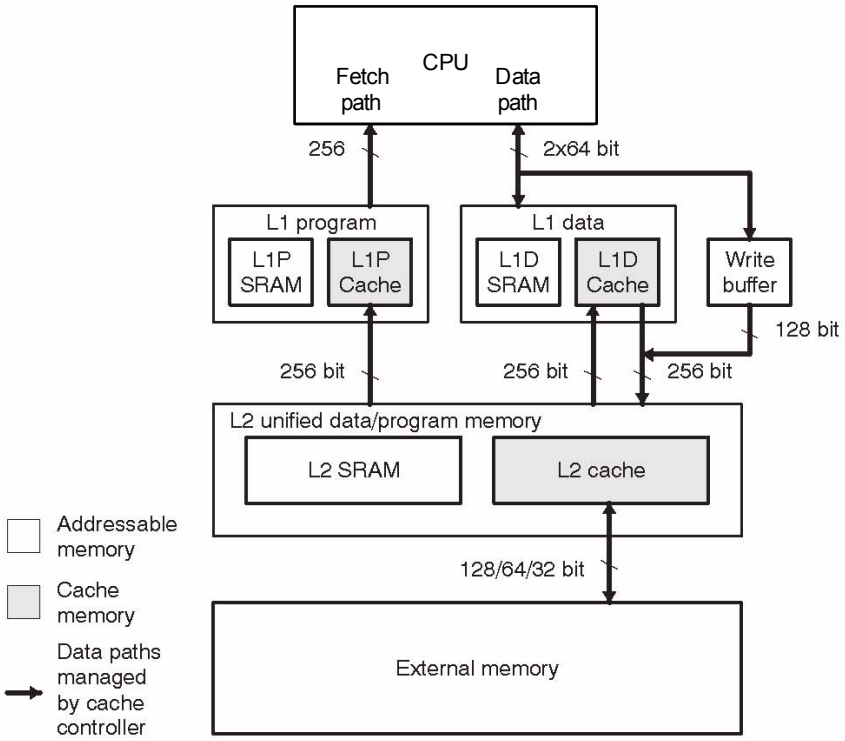
Figure 1-2 Access Pattern of a Six-Tap FIR Filter



1.7 Cache Memory Architecture Overview

The C66x DSP memory architecture consists of a two-level internal cache-based memory architecture plus external memory. Level 1 memory is split into program (L1P) and data (L1D). Both L1P and L1D can be configured into SRAM and cache with up to 32K bytes of cache. All caches and data paths shown in Figure 1-3 are automatically managed by the cache controller. Level 1 memory is accessed by the core without stalls. Level 2 memory is also configurable and can be split into L2 SRAM and cache with up to 256K bytes of cache. External memory can be several Megabytes large. The access time depends on the interface and the memory technology used.

Figure 1-3 C66x Cache Memory Architecture



1.8 Cache Basics

This section explains the different types of cache architectures and how they work. Generally, one can distinguish between direct-mapped caches and set-associative caches. The types of caches described use the C66x L1P (direct-mapped) and L1D (set-associative) as examples; however, the concept is similar for all cache-based computer architectures. This section focuses on the behavior of the cache system. Any performance considerations, including various stall conditions and associated stall cycles are described in Section 3.2 "Cache Performance Characteristics" on page 3-3.

1.8.1 Direct-Mapped Caches

The C66x program cache (L1P) will be used as an example to explain how a direct-mapped cache functions. Whenever the core accesses instructions in L2 SRAM or external memory, the instructions are brought into L1P cache. The characteristics of the C66x and the C64x L1P caches are summarized and compared in Table 1-2. The L1P miss stall characteristics are provided in Table 1-3

Table 1-2 L1P Cache Characteristics

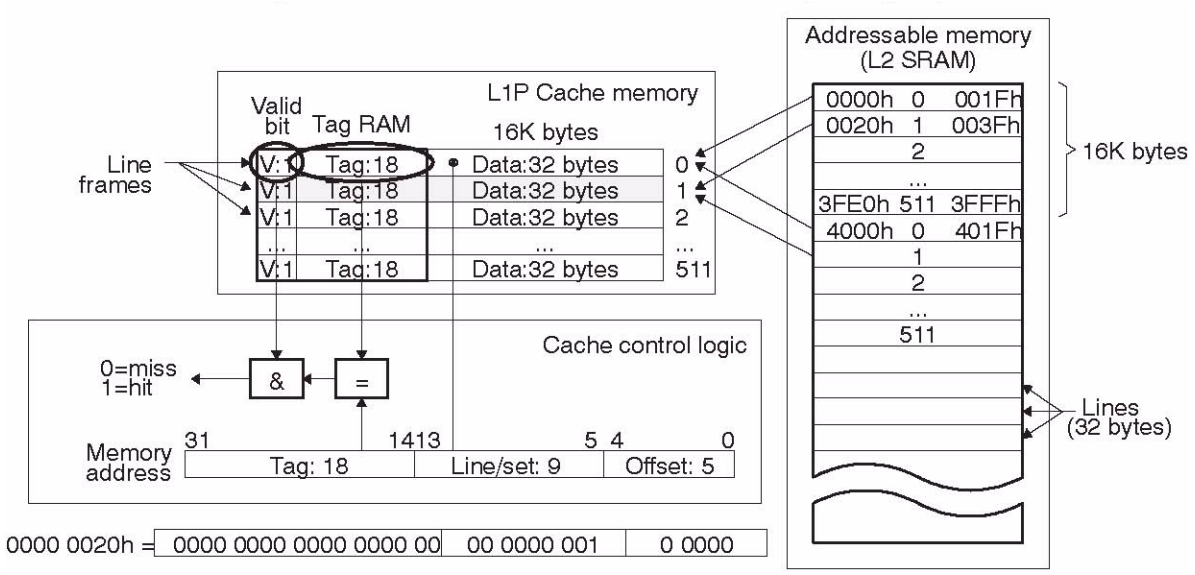
Characteristic	C66x DSP	C64x DSP
Organization	Direct-mapped	Direct-mapped
Protocol	Read Allocate	Read Allocate
core access time	1 cycle	1 cycle
Capacity	4K, 8K, 16K, or 32K bytes	16K bytes
Line size	32 bytes	32 bytes
External Memory Cacheability	Always cached	Configurable

Table 1-3 L1P Miss Stall Characteristics

L2 Type				
0 Wait-State, 2 × 128-bit Banks			1 Wait-State, 4 × 128-bit Banks	
Instructions per Execute Packet	L2 SRAM	L2 Cache	L2 SRAM	L2 Cache
1	0.000	0.000	0.000	0.000
2	0.001	0.497	0.167	0.499
3	0.501	1.247	0.751	1.249
4	0.997	1.997	1.329	1.999
5	1.499	2.747	1.915	2.749
6	2.001	3.497	2.501	3.499
7	2.497	4.247	3.079	4.249
8	2.999	4.997	3.665	4.999

Figure 1-4 shows the architecture of the C64+x L1P cache that consists of the cache memory and the cache control logic. Additionally, addressable memory (L2 SRAM or external memory) is shown. The cache memory size is 16K bytes in the example and consists of 512 32-byte lines. Each line frame always maps to the same fixed addresses in memory. For instance, as shown in Figure 1-4, addresses 0000h to 0019h are always cached in line frame 0 and addresses 3FE0h to 3FFFh are always cached in line frame 511. Since the capacity of the cache has been exhausted, addresses 4000h to 4019h map to line frame 0, and so forth. Note that one line contains exactly one instruction fetch packet.

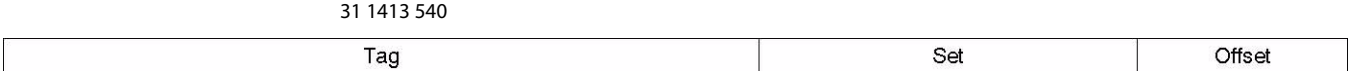
Figure 1-4 C66x L1P Cache Architecture (16K Bytes)



1.8.1.1 Read Misses

Consider a core program fetch access to address location 0020h. Assume that cache is completely invalidated, meaning that no line frame contains cached data. The valid state of a line frame is indicated by the valid (V) bit. A valid bit of 0 means that the corresponding cache line frame is invalid, that is, does not contain cached data. When the core makes a request to read address 0020h, the cache controller splits up the address into three portions as shown in Figure 1-5.

Figure 1-5 Memory Address from Cache Controller (For 16K Byte Cache Size)



The set portion (bits 13-5) indicates to which set the address maps to (in case of direct caches, a set is equivalent to a line frame). For the address 0020h, the set portion is 1. The controller then checks the tag (bits 31–14) and the valid bit. Since we assumed that the valid bit is 0, the controller registers a miss, that is the requested address is not contained in cache.

A miss also means that a line frame will be allocated for the line containing the requested address. Then the controller fetches the line (0020h-0039h) from memory and stores the data in line frame 1. The tag portion of the address is stored in the tag RAM and the valid bit is changed to 1 to indicate that the set now contains valid data. The fetched data is also forwarded to the core, and the access is complete. Why a tag portion of the address has to be stored becomes clear when address 0020h is accessed again. This is explained next.

1.8.1.2 Read Hits

The cache controller splits up the address into the three portions, as shown in Figure 1-5. The set portion determines the set, and the stored tag portion is now compared against the tag portion of the address requested. This comparison is necessary since multiple lines in memory are mapped to the same set. If we had

accessed address 4020h that also maps to the same set, the tag portions would be different and the access would have been a miss. If address 0020h is accessed, the tag comparison is true and the valid bit is 1; thus, the controller registers a hit and forwards the data in the cache line to the core. The access is complete.

1.8.2 Types of Cache Misses

Before set-associative caches are discussed, it is beneficial to acquire a better understanding of the properties of different types of cache misses. The ultimate purpose of a cache is to reduce the average memory access time. For each miss, there is a penalty for fetching a line of data from memory into cache. Therefore, the more often a cache line is reused the lower the impact of the initial penalty and the shorter the average memory access time becomes. The key is to reuse this line as much as possible before it is replaced with another line.

Replacing a line involves *eviction* of the line from cache and using the same line frame to store another line. If later the evicted line is accessed again, the access misses and the line has to be fetched again from slower memory. Therefore, it is important to avoid eviction of a line as long as it is still used.

1.8.2.1 Conflict and Capacity Misses

Evictions are caused by conflicts, that is, a memory location is accessed that maps to the same set as a memory location that was cached earlier. This type of miss is referred to as a *conflict miss*, a miss that occurred because the line was evicted due to a conflict before it was reused. It is further distinguished whether the conflict occurred because the capacity of the cache was exhausted or not. If the capacity was exhausted, all line frames in the cache were allocated when the miss occurred, then the miss is referred to as a *capacity miss*. Capacity misses occur if a data set that exceeds the cache capacity is reused. When the capacity is exhausted, new lines accessed start replacing lines from the beginning of the array.

Identifying the cause of a miss may help to choose the appropriate measure for avoiding the miss. Conflict misses mean that the data accessed fits into cache but lines get evicted due to conflicts. In this case, we may want to change the memory layout so that the data accessed is located at addresses in memory that do not conflict (map to the same set) in cache. Alternatively, from a hardware design, we can create sets that can hold two or more lines. Thus, two lines from memory that map to the same set can both be kept in cache without evicting one another. This is the idea of *set-associative* caches, described in Section 1.8.3.

In case of capacity misses, one may want to reduce the amount of data that is operated on at a time. Alternatively, from a hardware design, the capacity of the cache can be increased.

1.8.2.2 Compulsory Misses

A third category of misses are *compulsory misses* or first reference misses. They occur when the data is brought in cache for the first time. Unlike the other two misses, they cannot be avoided, hence, they are compulsory.

1.8.3 Set-Associative Caches

Set-associative caches have multiple *cache ways* to reduce the probability of conflict misses. The C66x L1D cache is a 2-way set-associative cache with 4K, 8K, 16K, or 32K bytes capacity and 64-byte lines. The characteristics of the L1D cache are summarized in Table 1-4. The L1D miss stall characteristics are provided in Table 1-5.

Table 1-4 L1D Cache Characteristics

Characteristic	C66x DSP	C64x DSP
Organization	2-way set-associative	2-way set-associative
Protocol	Read Allocate, Write–back	Read Allocate, Write–back
core access time	1 cycle	1 cycle
Capacity	4K, 8K, 16K, or 32K bytes	16K bytes
Line size	64 bytes	64 bytes
Replacement strategy	Least recently used (LRU)	Least recently used (LRU)
Write Buffer	4 x 128-bit entries	4 x 64-bit entries
External Memory Cacheability	Configurable	Configurable

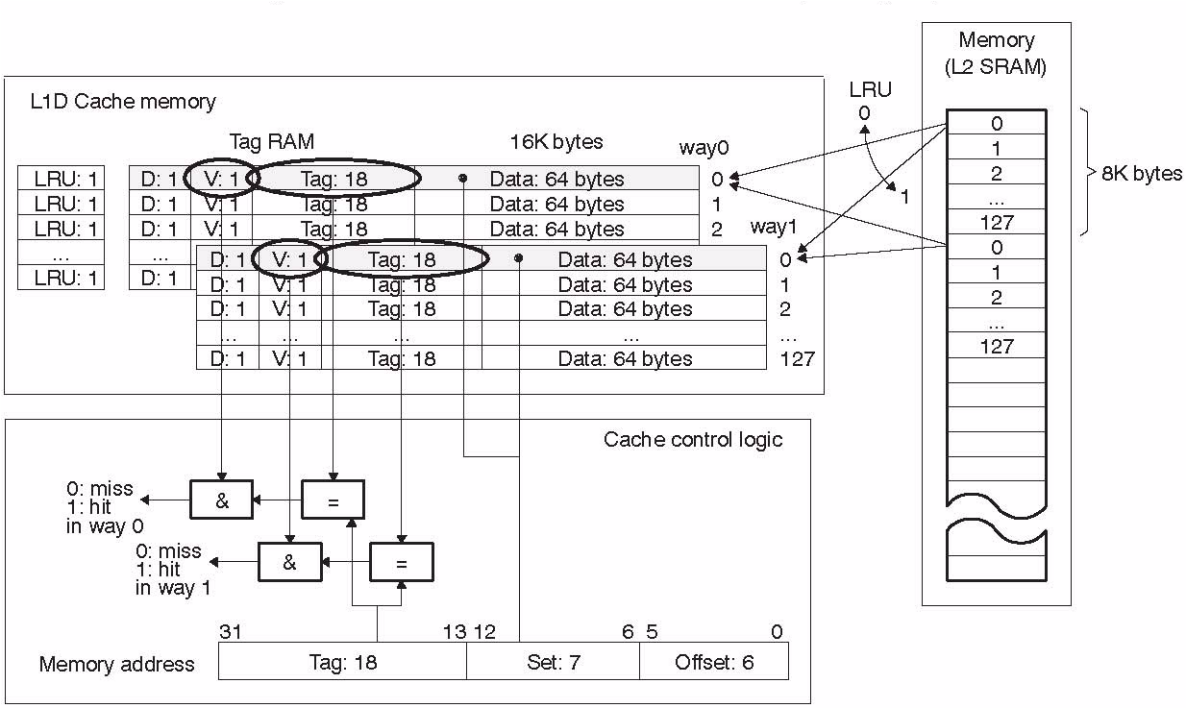
Table 1-5 L1D Miss Stall Characteristics

	L2 Type			
	0 Wait-State, 2 × 128-bit Banks		1 Wait-State, 4 × 128-bit Banks	
Parameter	L2 SRAM	L2 Cache	L2 SRAM	L2 Cache
Single Read Miss	10.5	12.5	12.5	14.5
2 Parallel Read Misses (pipelined)	10.5 + 4	12.5 + 8	12.5 + 4	14.5 + 8
M Consecutive Read Misses	10.5 + 3 × (M -1)	12.5 + 7 × (M -1)	12.5 + 3 × (M -1)	14.5 + 7 × (M -1)
(pipelined)				
M Consecutive Parallel Read	10.5 + 4 × (M/2 -1) +	12.5 + 8 × (M/2 -1) +	12.5 + 4 × (M -1)	14.5 + 8 × (M/2 -1) +
Misses (pipelined)	3 × M/2	7 × M/2		7 × M/2
Victim Buffer Flush on Read Miss	disrupts miss	disrupts miss	disrupts miss	disrupts miss
	pipelining plus	pipelining plus	pipelining plus	pipelining plus
	maximum 11 stalls	maximum 11 stalls	maximum 10 stalls	maximum 10 stalls
Write Buffer Drain Rate	2 cycles/entry	6 cycles/entry	2 cycles/entry	6 cycles/entry

Compared to a direct-mapped cache, each set of a 2-way set-associative cache consists of two line frames, one line frame in way 0 and another line frame in way 1. A line in memory still maps to one set, but now can be stored in either of the two line frames. In this sense, a direct-mapped cache can also be viewed as a 1-way cache.

The set-associative cache architecture is explained by examining how misses and hits are handled for the C66x L1D cache, shown in Figure 1-6. Hits and misses are determined similar as in a direct-mapped cache, except that two tag comparisons, one for each way, are necessary to determine which way the requested data is kept.

Figure 1-6 C66x LiD Cache Architecture (16K Bytes)



1.8.3.1 Read Misses

If both ways miss, the data first needs to be fetched from memory. The LRU bit determines in which cache way the line frame is allocated. An LRU bit exists for each set and can be thought of as a switch. If the LRU bit is 0, the line frame in way 0 is allocated; if the LRU bit is 1, the line frame in way 1 is allocated. The state of the LRU bit changes whenever an access is made to the line frame. When a way is accessed, the LRU bit always switches to the opposite way, as to protect the most-recently-used line frame from being evicted. Conversely, on a miss, the least-recently-used (LRU) line frame in a set is allocated to the new line evicting the current line. The reason behind this line replacement scheme is based on the principle of locality: if a memory location was accessed, then the same or a neighboring location will be accessed soon again. Note that the LRU bit is only consulted on a miss, but its status is updated every time a line frame is accessed regardless whether it was a hit or a miss, a read or a write.

1.8.3.2 Write Misses

L1D is a read–allocate cache, meaning that a line is allocated on a read miss only. On a write miss, the data is written to the lower-level memory through a write buffer, bypassing L1D cache (see Figure 1-3). The write buffer consists of 4 entries. On C66x devices, each entry is 128-bits wide.

1.8.3.3 Read Hits

If there is a read hit in way 0, the data of the line frame in way 0 is accessed; if there is a hit in way 1, the data of the line frame in way 1 is accessed.

1.8.3.4 Write Hits

On a write hit, the data is written to the cache, but is not immediately passed on to the lower level memory. This type of cache is referred to as *write–back* cache, since data that was modified by a core write access is written back to memory at a later time. To write back modified data, it must be known which line was written by the core. For this purpose, every cache line has a *dirty bit* (D) associated with it. Initially, the dirty bit is zero. As soon as the core writes to a cached line, the corresponding dirty bit is set. When the dirty line needs to be evicted due to a conflicting read miss, it will be written back to memory. If the line was not modified (*clean* line), its contents are discarded. For instance, assume the line in set 0, way 0 was written to by the core, and the LRU bit indicates that way 0 is to be replaced on the next miss. If the core now makes a read access to a memory location that maps to set 0, the current dirty line is first written back to memory, then the new data is stored in the line frame. A write–back may also be initiated by the program, by sending a writeback command to the cache controller. Scenarios where this is required include boot loading and self-modifying code.

1.8.4 Level 2 (L2) Cache

Until now, it was assumed that there is one level of cache memory between the core and the addressable main memory. If there is a larger difference in memory size and access time between the cache and main memory, a second level of cache is typically introduced to further reduce the number of accesses to memory. A level 2 (L2) cache basically operates in the same manner as a level 1 cache; however, level 2 cache are typically larger in capacity. Level 1 and level 2 caches interact as follows: an address misses in L1 and is passed on to L2 for handling; L2 employs the same valid bit and tag comparisons to determine if the requested address is present in L2 cache or not. L1 hits are directly serviced from the L1 caches and do not require involvement of L2 caches.

As L1P and L1D, the L2 memory space can also be split into an addressable internal memory (L2 SRAM) and a cache (L2 Cache) portion. Unlike L1 caches that are read–allocate only, L2 cache is a read and write allocate cache. L2 cache is used to cache external memory addresses only; whereas, L1P and L1D are used to cache both L2 SRAM and external memory addresses. L2 cache characteristics are summarized in Table 1-6.

Table 1-6 L2 Cache Characteristics

Characteristic	C66x DSP	C64x DSP
Organization	4-way set-associative	4-way set-associative
Protocol	Read and write allocate	Read and write allocate
	Writeback	Writeback
Capacity	32K, 64K, 128K, or 256K bytes	32K, 64K, 128K, or 256K bytes
Line size	128 bytes	128 bytes
Replacement strategy	Least recently used (LRU)	Least recently used (LRU)
External Memory Cacheability	Configurable	Configurable

1.8.4.1 Read Misses and Hits

Consider a core read request to a cacheable external memory address that misses in L1 cache (may be L1P or L1D). If the address also misses L2 cache, the corresponding line will be brought into L2 cache. The LRU bits determine the way in which the line frame is allocated. If the line frame contains dirty data, it will be first written back to external memory before the new line is fetched. (If data of this line is also contained in L1D, it will be first written back to L2 before the L2 line is sent to external memory. This is

required to maintain cache coherence, which is further explained in Section 2.4). The portion of the newly allocated line forming an L1 line and containing the requested address is then forwarded to L1. L1 stores the line in its cache memory and finally forwards the requested data to the core. Again, if the new line replaces a dirty line in L1, its contents are first written back to L2.

If the address was an L2 hit, the corresponding line is directly forwarded from L2 to L1 cache.

1.8.4.2 Write Misses and Hits

If a core write request to an external memory address misses L1D, it is passed on to L2 through the write buffer. If L2 detects a miss for this address, the corresponding L2 cache line is fetched from external memory, modified with the core write, and stored in the allocated line frame. The LRU bits determine the way in which the line frame is allocated. If the line frame contains dirty data, it will be first written back to external memory before the new line is fetched. Note that the line is not stored in L1D, since it is a read–allocate cache only.

If the address was an L2 hit, the corresponding L2 cache line frame is directly updated with the core write data.

1.8.5 Cacheability of External Memory Addresses

L2 SRAM address are always cached in L1P and L1D. However, external memory addresses by default are configured as noncacheable in L1D and L2 caches. Cacheability must first be explicitly enabled by the user. Note that L1P cache is not affected by this configuration and always caches external memory addresses. If an address is noncacheable, any memory access (data access or program fetch) is made without allocating the line in either L1D or L2 cache (see Section 2.1 and Section 2.2 for more information).

Chapter 2

Using Cache

This chapter describes how to enable and configure caches for C66x devices. It also describes the cache coherence protocol used by the cache controller and gives examples of common application scenarios.

Because multiple copies of the same memory location can exist simultaneously in a cache-based memory system, a protocol must be followed to ensure that requestors do not access an out-of-date copy of a memory location. This protocol is referred to as a cache coherence protocol.

CAUTION—In the following cases, it is the user's responsibility to maintain cache coherence. Failing to do so can cause the application to function incorrectly.

DMA or other external entity writes data or code to external memory that is then read by the CORE.

The CORE writes data to external memory that is then read by DMA or another external entity

DMA writes code to L2 SRAM that is then executed by the CORE (this case is supported by the hardware protocol on C621x/C671x and C64x DSPs, but is not supported on C66x DSPs).

CORE writes code to L2 SRAM or external memory that is then executed by the CORE.

2.1 Configuring L1 Caches

The configuration at boot time depends on the particular C66x device. The device may boot up as cache only, SRAM only, or a combination of each. See your device-specific data manual.

The L1P and L1D cache sizes can be changed in the program code by issuing the appropriate chip support library (CSL) commands:

```
CACHE_L1pSetSize();
CACHE_L1dSetSize();
```

Additionally, in the linker command file the memory to be used as SRAM has to be specified. Since caches cannot be used for code or data placement by the linker, all sections must be linked into SRAM or external memory.

2.2 Configuring L2 Cache

At boot time L2 cache is disabled and all of L2 is configured as SRAM (addressable internal memory). If DSP/BIOS is used, L2 cache is enabled automatically; otherwise, L2 cache can be enabled in the program code by issuing the appropriate chip support library (CSL) command: CACHE_L2SetSize();

Additionally, in the linker command file the memory to be used as SRAM has to be specified. Since cache cannot be used for code or data placement by the linker, all sections must be linked into SRAM or external memory.

2.3 Cacheability

For L1D and L2, you can control whether external memory addresses are cacheable or noncacheable. Each external memory address space of 16M bytes is controlled by a memory attribute register (MAR) bit (0 = noncacheable, 1 = cacheable). The memory attribute registers are documented in *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x. For instance, to enable caching for the external memory range from 8000 0000h to 80FF FFFFh, the CSL function CACHE_enableCaching(CACHE_MAR128) can be used. This sets MAR128 to 1. After the MAR bit is set for an external memory space, new addresses accessed by the CORE will be cached. If it was left noncacheable, the requested data would simply be forwarded from external memory to the CORE without being stored in L1D or L2 cache. Note that program fetches are always cached in L1P regardless of the MAR settings. At boot time, caching for external memory address space is disabled.

The following description assumes 2048K bytes of L2 memory and that L1P and L1D are all cache. For C66x devices with different L2 sizes, see the device-specific data manual. The linker command file for a configuration of 1792K SRAM and 256K-bytes cache is shown in Example 2-1.

The required CSL command sequence to enable caching of external memory locations and to enable L2 cache is shown in Example 2-2. The first command enables caching of the first 16 Mbytes in the external memory space by setting the appropriate MAR bit. Finally, L2 cache size is set to 256K bytes.

Figure 2-1 shows all possible cache configurations for C66x devices with 2048K bytes of L2 memory. Slightly different configurations may exist for other C66x devices, see your device-specific data manual.

Note that when the L2 cache size is increased, the memory is taken from the high memory addresses.

Other configurations are set by adjusting the cache size in Example 2-1 and Example 2-2.

Note—Do not define memory that is to be used or boots up as cache under the MEMORY directive. This memory is not valid for the linker to place code or data in. If L1D SRAM and/or L1P SRAM is to be used, it must first be made available by reducing the cache size. Data or code must be linked into L2 SRAM or external memory and then copied to L1 at run-time.

Example 2-1 C66x Linker Command File

```
MEMORY
{
    L2SRAM: origin = 00800000h length = 001C0000h
    CE0: origin = 80000000h length = 01000000h
}
SECTIONS
{
    .cinit > L2SRAM
    .text > L2SRAM
    .stack > L2SRAM
    .bss > L2SRAM
    .const > L2SRAM
    .data > L2SRAM
    .far > L2SRAM
    .switch > L2SRAM
    .sysmem > L2SRAM
```

```
    .tables > L2SRAM
    .cio > L2SRAM
    .external > CE0
}
```

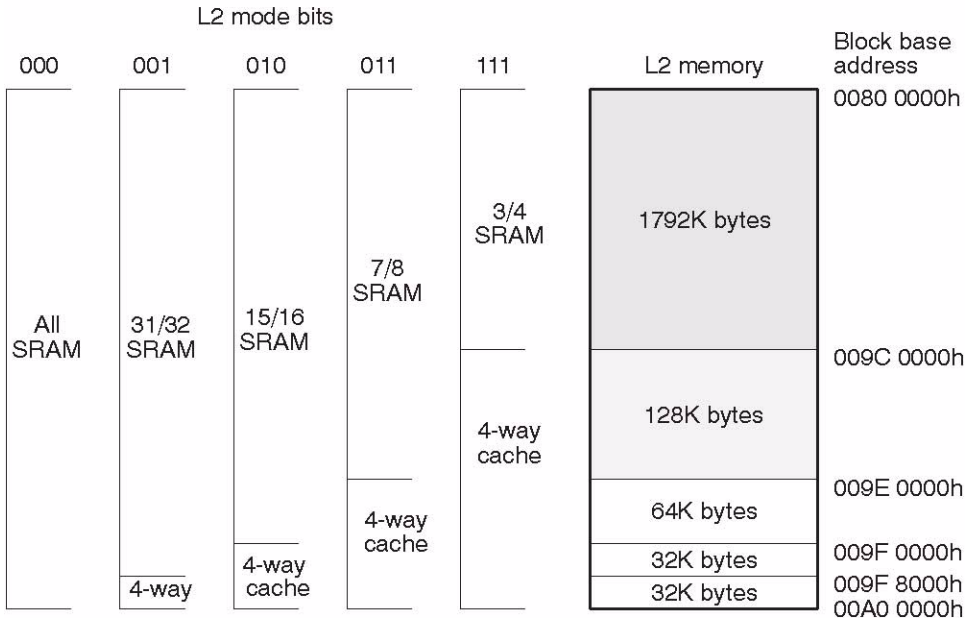
End of Example 2-1

Example 2-2 C66x CSL Command Sequence to Enable Caching

```
#include <csl.h>
#include <csl_cache.h>
...
CACHE_enableCaching(CACHE_CE00);
CACHE_setL2Size(CACHE_256KCACHE);
```

End of Example 2-2

Figure 2-1 C66x L2 Memory Configurations



2.4 Coherence

Generally, if multiple devices, such as the CORE or peripherals, share the same cacheable memory region, cache and memory can become incoherent. Consider the system shown in Figure 2-2. Suppose the CORE accesses a memory location that gets subsequently allocated in cache (1). Later, a peripheral is writes data to this same location that is meant to be read and processed by the CORE (2). However, since this memory location is kept in cache, the memory access hits in cache and the CORE reads the old data instead of the new data (3). A similar problem occurs if the CORE writes to a memory location that is cached, and the data is to be read by a peripheral. The data only gets updated in cache but not in memory from where the peripheral reads the data. The cache and the memory are said to be *incoherent*.

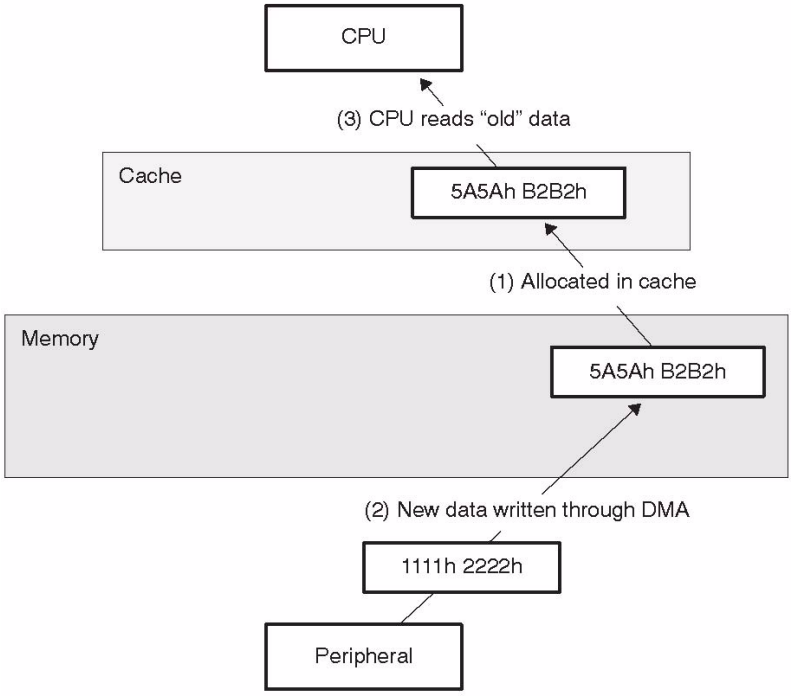
Coherence needs to be addressed if the following is true:

- Multiple requestors (CORE data path, CORE fetch path, peripherals, DMA controllers, other external entities) share a region of memory for the purpose of data exchange.
- This memory region is cacheable by at least one device.
- A memory location in this region has been cached.
- And this memory location is modified (by any device).

Consequently, if a memory location is shared, cached, and has been modified, there is a cache coherence problem.

C66x DSPs *automatically* maintain cache coherence for data accesses by the CORE and EDMA/IDMA through a hardware cache coherence protocol based on *snoop* commands. The coherence mechanism is activated on a DMA read and write access. When a DMA read of a cached L2 SRAM location occurs, the data is directly forwarded from L1D cache to the DMA without being updated in L2 SRAM. On a DMA write, the data is forwarded to L1D cache and is updated in L2 SRAM.

Figure 2-2 Cache Coherence Problem



In the following cases, it is your responsibility to maintain cache coherence:

- DMA or other external entity writes data or code to external memory that is then read by the CORE
- CORE writes data to external memory that is then read by DMA or another external entity
- DMA writes code to L2 SRAM that is then executed by the CORE (this case is supported by the hardware protocol on C621x/C671x and C64x DSPs, but is not supported on C66x DSPs)
- CORE writes code to L2 SRAM or external memory that is then executed by the CORE

For this purpose, the cache controller offers various commands that allow it to manually keep caches coherent.

This section explains how to maintain coherence by describing the cache coherence protocol and providing examples for common types of applications.

2.4.1 Snoop Coherence Protocol

Before describing programmer-initiated cache coherence operations, it is beneficial to first understand the snoop-based protocols that are used by the cache controller to maintain coherence between the L1D cache and L2 SRAM for DMA accesses. Generally, snooping is a cache operation initiated by a lower-level memory to check if the address requested is cached (valid) in the higher-level memory. If yes, the appropriate operation is triggered. The C66x cache controller supports the following snoop commands:

L1D Snoop–Read

L1D Snoop–Write

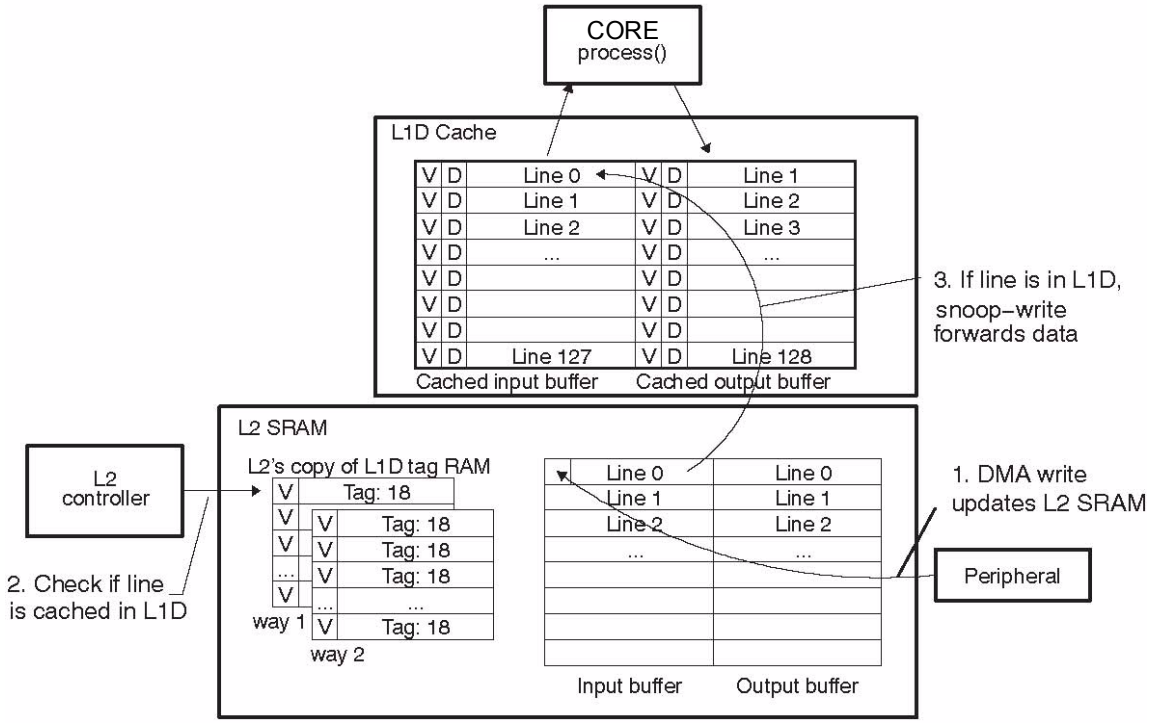
2.4.2 Cache Coherence Protocol for DMA Accesses to L2 SRAM

To illustrate snooping, assume a peripheral writes data through the DMA to an input buffer located in L2 SRAM. Then the CORE reads the data, processes it, and writes it to an output buffer. From there, the data is sent through the DMA to another peripheral.

The procedure for a DMA write is shown in Figure 2-3 and is:

1. The peripheral requests a write access to a line in L2 SRAM that maps to set 0 in L1D.
2. The L2 cache controller checks its local copy of the L1D tag RAM and determines if the line that was just requested is cached in L1D (by checking the valid bit and the tag). If the line is not cached in L1D, no further action needs to be taken and the data is written to memory.
3. If the line is cached in L1D, the L2 controller updates the data in L2 SRAM and directly updates L1D cache by issuing a *snoop–write* command. Note that the dirty bit is not affected by this operation.

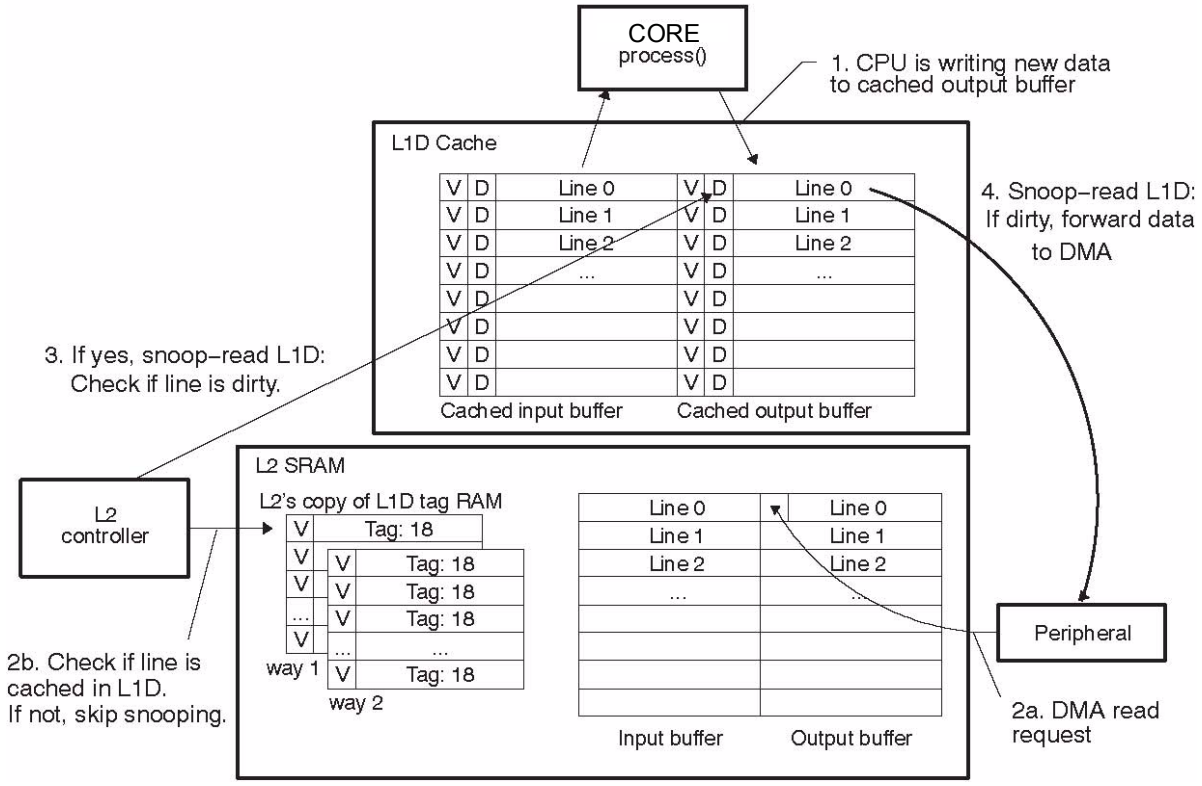
Figure 2-3 DMA Write to L2 SRAM



The procedure for a DMA read is shown in Figure 2-4 and is:

1. The CORE writes the result to the output buffer. Assume that the output buffer was preallocated in L1D. Since the buffer is cached, only the cached copy of the data is updated, but not the data in L2 SRAM.
2. When the peripheral issues a DMA read request to the memory location in L2 SRAM, the controller checks to determine if the line that contains the memory location requested is cached in L1D. In this example, we already assumed that it is cached. However, if it was not cached, no further action would be taken and the peripheral would complete the read access.
3. If the line is cached, the L2 controller sends a *snoop–read* command to L1D. The snoop first checks to determine if the corresponding line is dirty. If not, the peripheral is allowed to complete the read access.
4. If the dirty bit is set, the snoop–read causes the data to be forwarded directly to the DMA without writing it to L2 SRAM. This is the case in this example, since we assumed that the CORE has written to the output buffer.

Figure 2-4 DMA Read of L2 SRAM



2.4.2.1 L2 SRAM Double Buffering Example

Having described how coherence is maintained for a DMA write and read of L2 SRAM, a typical double buffering example is now presented. Assume data is read in from one peripheral, processed, and written out to another peripheral, a structure of a typical signal processing application. The data flow is shown in Figure 2-5. The idea is that while the CORE is processing data from one pair of buffers (for example, InBuffA and OutBuffA), the peripherals are writing/reading data using the other pair of buffers (InBuffB and OutBuffB) such that the DMA data transfer may occur in parallel with CORE processing.

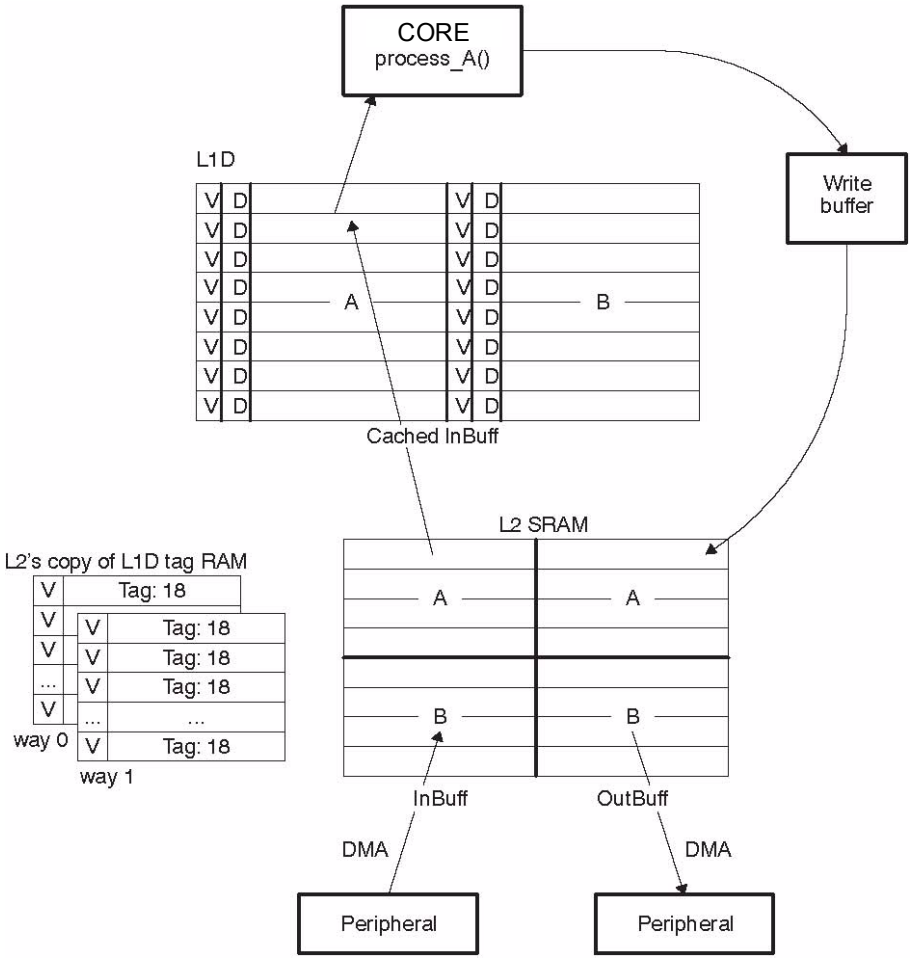
Assuming that InBuffA has been filled by the peripheral, the procedure is:

1. InBuffB is being filled while the CORE is processing data in InBuffA. The lines of InBuffA are allocated in L1D. Data is processed by the CORE and is written through the write buffer to OutBuff A (remember that L1D is read-allocate only).
2. When the peripheral is filling InBuffA with new data, the second peripheral is reading from OutBuffA and the CORE is processing InBuffB. For InBuffA, the L2 cache controller automatically takes care of forwarding the data to L1D through snoop–writes. For OutBuffA, since it is not cached in L1D, no snoops are necessary.
3. Buffers are then switched again, and so on.

It may be beneficial to make the buffers in L2 SRAM fit into a multiple of L1D cache lines, in order to get the highest return (in terms of cached data) for every cache miss.

The pseudo-code in Example 2-3 shows how a double buffering scheme could be realized.

Figure 2-5 Double Buffering in L2 SRAM



Example 2-3 Example 2-3. L2 SRAM DMA Double Buffering Code

```
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 Example 2-3

2.4.2.2 Maintaining Coherence Between External Memory and Cache

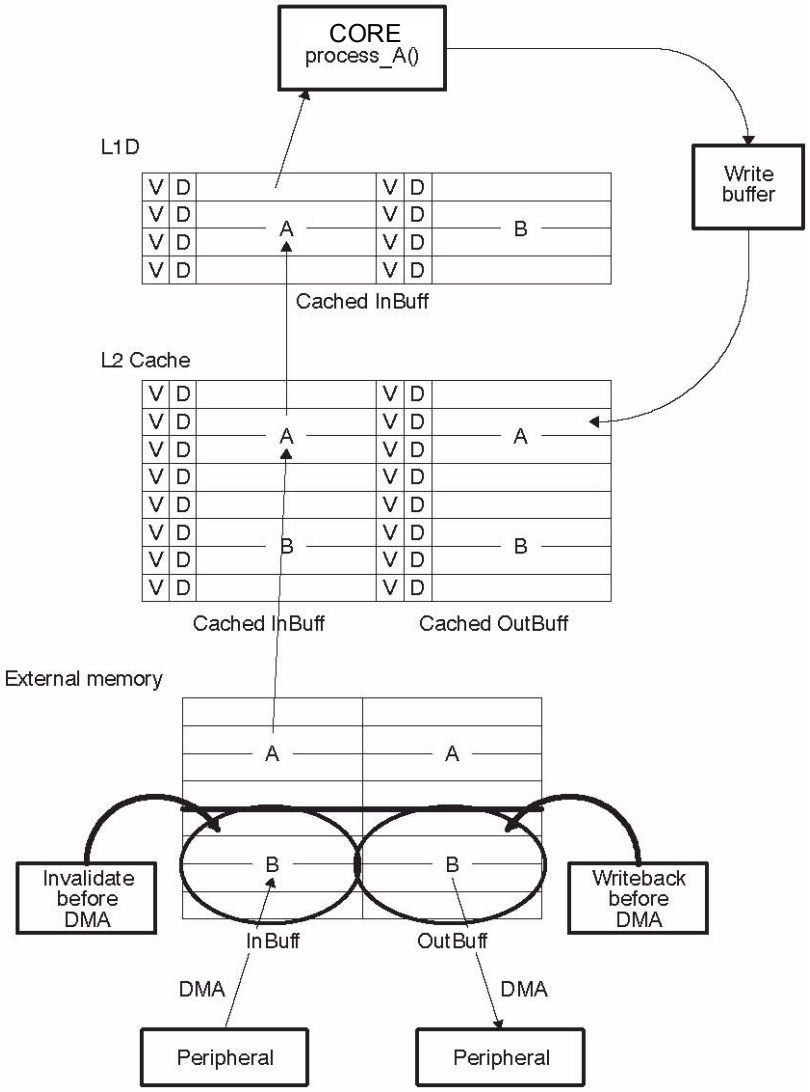
Now the same double buffering scenario is considered, but with the buffers located in external memory. Since the cache controller does not automatically maintain coherence in this case, it is the programmer's responsibility to maintain coherence. Again, the CORE reads in data from a peripheral, processes it, and writes it out to another peripheral via DMA. But now the data is additionally passed through L2 cache.

As shown in Figure 2-6, assume that transfers already have occurred, that both InBuff and OutBuff are cached in L2 cache, and that InBuff is cached in L1D. Further assume that the CORE has completed processing InBuffB, filled OutBuffB, and is now about to start processing InBuffA. The transfers that bring in new data into InBuffB and commit the data in OutBuffB to the peripheral are also about to begin.

To maintain coherence, all the lines in L1D and L2 cache that map to the external memory input buffer have to be *invalidated* before the DMA transfer starts. This way the CORE will reallocate these lines from external memory next time the input buffer is read.

Similarly, before OutBuffB is transferred to the peripheral, the data first has to be *written back* from L1D and L2 caches to external memory. This is done by issuing a *writeback* operation. Again, this is necessary since the CORE writes data only to the cached copies of the memory locations of OutBuffB that still may reside in L1D and L2 cache.

Figure 2-6 Double Buffering in External Memory



The chip support library (CSL) provides a set of routines that allow the required cache coherence operations to be initiated. The start address of the buffer in external memory and the number of bytes need to be specified:

```
CACHE_invL2(InBuffB, BUFSIZE, CACHE_WAIT);
CACHE_wbL2(OutBuffB, BUFSIZE, CACHE_WAIT);
```

If CACHE_WAIT is used, the routine waits until the operation has completed. This is the recommended mode of operation. If CACHE_NOWAIT is used, the routine initiates the operation and immediately returns. This allows the CORE to continue execution of the program while the coherence operation is performed in the background. However, care must be taken that the CORE is not accessing addresses that the cache controller is operating on since this may cause undesired results. The routine CACHE_wait() can then be used before the DMA transfer is initiated, to ensure completion of the coherence operation. More information on these cache coherence operations is in Section 2.4.3.

The pseudo-code in Example 2-4 shows exactly in which order the cache coherence calls and the DMA transfers should occur.

Example 2-4 External Memory DMA Double Buffering Code

```
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 Example 2-4

In addition to the coherence operations, it is important that all DMA buffers are aligned at an L2 cache line, and are an integral multiple of cache lines large. Further details on why this is required are given in Section 2.4.3. These requirements can be achieved as shown:

```
#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];
```

Alternatively, the CSL macro CACHE_ROUND_TO_LINESIZE(cache, element count, element size) can be used that automatically rounds array sizes up to the next multiple of a cache line size. The first parameter is the cache type, which can be L1D, L1P, or L2.

The array definitions would then look as:

```
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)];
```

2.4.3 Usage Guidelines for L2 Cache Coherence Operations

CAUTION—If the guidelines set out in the section are not followed, correct functioning of the application cannot be assured.

Table 2-1 shows an overview of available L2 cache coherence operations for C66x devices. Note that these operations always operate on L1P and L1D even if L2 cache is disabled. Table 2-1 has to be interpreted as follows:

1. First, the cache controller operates on L1P and L1D
2. then, the operation is performed on L2 cache

Note—A line cached in L1P or L1P is not necessarily cached in L2. A line may be evicted from L2 without being evicted from L1P or L1D.

Table 2-1 L2 Cache Coherence Operations

Scope	Coherence Operation	CSL Command	Operation on L2 Cache	Operation on L1D Cache	Operation on L1P Cache
Range	Invalidate L2	CACHE_invL2 (start address, byte count, wait)	All lines within range invalidated (any dirty data is discarded).	All lines within range invalidated (any dirty data is discarded).	All lines within range invalidated.
	Writeback L2	CACHE_wbL2 (start address, byte count, wait)	Dirty lines within range written back. All lines kept valid.	Dirty lines within range written back. All lines kept valid.	None
	Writeback– Invalidate L2	CACHE_wbInvL2 (start address, byte count, wait)	Dirty lines within range written back. All lines within range invalidated.	Dirty lines within range written back.All lines within range invalidated.	All lines within range invalidated.
All L2 Cache	Writeback All L2	CACHE_wbAllL2 (wait)	All dirty lines in L2 written back. All lines kept valid.	All lines within range invalidated All dirty lines in L1D written back. All lines kept validL1D snoop–invalidate.	None
	Writeback– Invalidate All L2	CACHE_wbInvAllL2 (wait)	All dirty lines in L2 written back. All lines in L2 invalidated.	All dirty lines in L1D written back. All lines in L1D invalidated.	All lines in L1P invalidated.

It is important to note that although a start address and a byte count is specified, the cache controller operates always on *whole lines*. Therefore, for the purpose of maintaining coherence, arrays must be:

- a multiple of L2 cache lines large
- aligned at an L2 cache line boundary

An L2 cache line is 128 bytes. The cache controller operates on all lines that are "touched" by the specified range of addresses. Note that the maximum byte count that can be specified is 4 × 65 535 bytes (on some C66x devices the maximum is 4 × 65 408 bytes, see your device-specific data manual), that is, one L2 cache operation can operate on at most 256K bytes. If the external memory buffer to be operated on is larger, multiple cache operations have to be issued.

The following guidelines should be followed for using cache coherence operations. Again, user-issued L2 cache coherence operations are only required if the CORE and DMA (or other external entity) share a cacheable region of external memory, that is, if the CORE reads data written by the DMA and conversely.

The safest rule would be to issue a Writeback–Invalidate All prior to any DMA transfer to or from external memory. However, the disadvantage of this is that possibly more cache lines are operated on than is required, causing a larger than necessary cycle overhead. A more targeted approach is more efficient. First, it is only required to operate on those cache lines in memory that actually contain the shared buffer. Second, it can be distinguished between the three scenarios shown in Table 2-2.

Table 2-2 Scenarios and Required L2 Coherence Operations on External Memory

Scenario	Coherence Operation Required
DMA/Other reads data written by the CORE	Writeback L2 *before* DMA/Other starts reading
DMA/Other writes data (code) that is to be read (executed) by the CORE	Invalidate L2 *before* DMA/Other starts writing
DMA/Other modifies data written by the CORE that data is to be read back by the CORE	Writeback–Invalidate L2 *before* DMA/Other starts writing

In scenario 3, the DMA may modify data that was written by the CORE and that data is then read back by the CORE. This is the case if the CORE initializes the memory (for example, clears it to zero) before a peripheral writes to the buffer. Before the DMA starts, the data written by the CORE needs to be committed to external memory and the buffer has to be invalidated.

For a more in-depth discussion of coherence requirements for the C66x DSP, see Appendix B.

2.4.4 Usage Guidelines for L1 Cache Coherence Operations

CAUTION—If the guidelines set out in the section are not followed, correct functioning of the application cannot be assured.

Table 2-3 and Table 2-4 show an overview of available L1 cache coherence operations for C66x devices.

Table 2-3 L1D Cache Coherence Operations

Scope	Coherence Operation	CSL Command	Operation on L1D Cache
Range	Invalidate L1D	CACHE_invL1d (start address, byte count, wait)	All lines within range invalidated (any dirty data is discarded).
	Writeback L1D	CACHE_wbL1d (start address, byte count, wait)	Dirty lines within range written back. All lines kept valid.
	Writeback–Invalidate L1D	CACHE_wbInvL1d (start address, byte count, wait)	Dirty lines within range written back. All lines within range invalidated.
All L1D Cache	Writeback All L1D	CACHE_wbAllL1d (wait)	All dirty lines in L1D written back. All lines kept valid.
	Writeback–Invalidate All L1D	CACHE_wbInvAllL1d (wait)	All dirty lines in L1D written back. All lines invalidated.

Table 2-4 L1P Cache Coherence Operations

Scope	Coherence Operation	CSL Command	Operation on L1P Cache
Range	Invalidate L1P	CACHE_invL1p	All lines within range invalidated.
		(start address, byte count, wait)	
All L1P Cache	Invalidate All L1P	CACHE_wbInvAllL1p	All lines in L1P invalidated.
		(wait)	

It is important to note that although a start address and a byte count is specified, the cache controller operates always on *whole lines*. Therefore, for the purpose of maintaining coherence, arrays must be:

- a multiple of L1D cache lines large
- aligned at an L1D cache line boundary

An L1D cache line is 64 bytes. The cache controller operates on all lines that are "touched" by the specified range of addresses. Note that the maximum byte count that can be specified is 4 × 65 535. Table 2-5 shows scenarios with the cache coherence operations to be followed.

Table 2-5 Scenarios and Required L1 Coherence Operations

Scenario	Coherence Operation Required
DMA/Other writes code to L2 SRAM that is to be executed by the CORE	Invalidate L1P before CORE starts executing
CORE modifies code in L2 SRAM or external memory that is to be executed by the CORE	Invalidate L1P and Writeback-Invalidate L1D before CORE starts executing

For a more in-depth discussion of coherence requirements for the C66x DSP, see Appendix B on page B-1.

2.5 On-Chip Debug Support

The C66x DSPs provide on-chip debug support for debugging cache coherence issues (on earlier versions of some C66x devices, full functionality may be only provided on simulator platforms). Specifically, the C66x memory system allows emulation direct access to individual caches and reports cache state information (valid, dirty, LRU bits). This capability is exposed through the Memory Window in Code Composer Studio IDE (version 3.2 or higher).

For example, if you suspect a coherence problem with DMA writing new data to a buffer in external memory because the CORE appears to read incorrect data, you could follow these steps. First ensure that you eliminated any unpredictable interaction of the CORE accesses with coherence operations to exclude other causes than cache incoherence (for example, source code errors such as stray CORE writes or reads).

Then ensure that the buffer is aligned on L2 cache line boundaries to eliminate false addresses. For this purpose, the Memory Window provides visual cache line boundary markers that help you to easily identify misalignments. Next verify the correct use of cache coherence operations:

1. Halt the CORE execution after completion of the invalidate coherence operation but before the first DMA write access.
2. Verify that no line in the buffer is dirty. To check this, enable the Memory Analysis function (through the property window). Any dirty lines will then be displayed in a bold font style.
3. Continue CORE execution.
4. Halt the CORE again before the first CORE read.
5. Verify that the buffer is (still) invalidated and contains the expected new data. If there is a problem and data happens to be cached, you can use the cache bypass check boxes to inspect data contents in external memory.

The diagrams in Appendix B on page B-1 can help you develop similar procedures for other coherence scenarios.

2.6 Self-Modifying Code and L1P Coherence

No coherence is maintained between L1D and L1P. That means if the CORE wants to write or modify program code, the writes may only update L1D, L2 SRAM, or L2 cache, but not L1P. For the CORE to be able to execute the modified code, the addresses containing the instructions must not be cached in either L1D or L1P.

Consider an example where an interrupt vector table is to be modified during runtime, the following procedure has to be followed:

1. Disable interrupts
2. Perform CORE writes (STW) to modify code
3. Perform coherence operations:
   a. Perform an L1D Writeback–Invalidate operation
   b. Perform an L1P Invalidate operation
   c. Wait for the last operation to complete. Waiting for completion is done by polling the word count (xxWC) registers. This automatically ensures that any L1D write misses have drained from the write buffer. This is because polling a memory-mapped register is treated as a read miss that always causes the write buffer to be completely drained
4. Reenable interrupts

2.7 Changing Cache Configuration During Run-Time

This section explains how cache configurations may be safely changed during run-time.

2.7.1 Disabling External Memory Caching

Disabling external memory caching after it was enabled should not be generally necessary. However if it is, then the following considerations should be taken into account. If the MAR bit is changed from 1 to 0, external memory addresses already cached stay in the cache and accesses to those addresses still hit. The MAR bit is only consulted if the external memory address misses in L2. (This includes the case where L2 is all SRAM. Since there is no L2 cache, this can also be interpreted as an L2 miss).

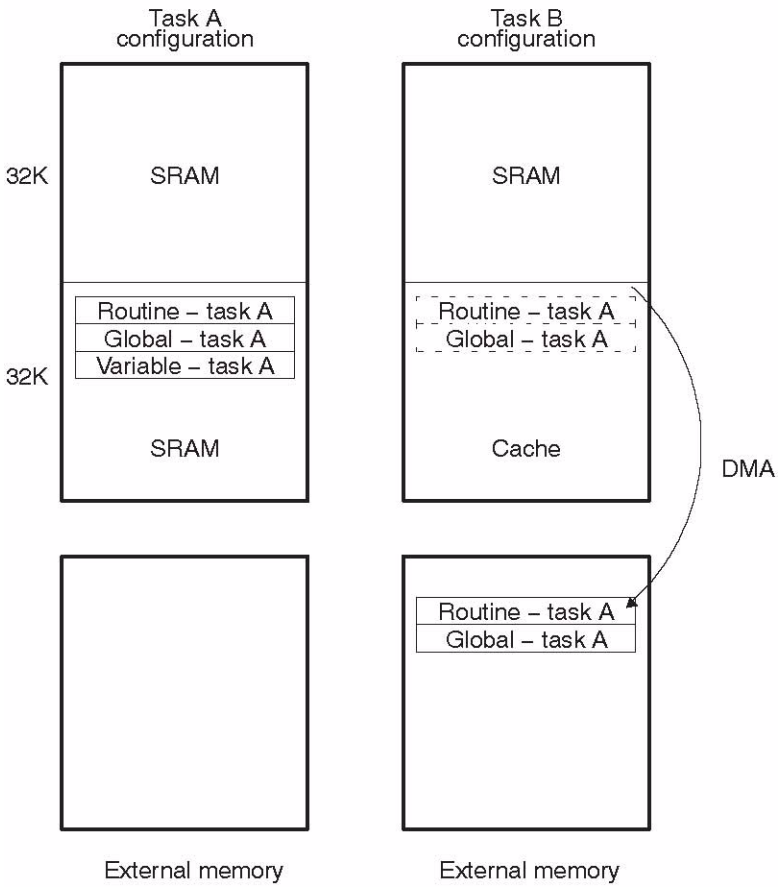
If all addresses in the respective external memory address space are made noncacheable, the addresses need to be written back and invalidated first (see Section 2.4.3 and Section 2.4.4 for a description of user-initiated cache control operations).

2.7.2 Changing Cache Sizes During Run-Time

Changing the size of caches during run time may be beneficial for some applications. Consider the following example for a C66x device with 64K bytes of L2. An application has two tasks, A and B. Task A benefits from 64K bytes of code and data being allocated in L2 SRAM, while task B would benefit from having 32K bytes of L2 cache. Assume the memory configuration as shown in Figure 2-7. The second 32K byte segment contains the routine, some global variables for task A (that need to be preserved during task B executes), and some variables for task A that after task switching are no longer needed.

The memory region where this routine and the variables reside can then be freed (assume no other sections are located in this 32K byte segment) by copying the code and the global variables to another memory region in external memory using a DMA. Then, the cache mode can be switched. The cache controller will automatically writeback–invalidate all cache lines before initializing with the new size. Note that changing of the L2 cache size will not cause any evictions from L1P or L1D cache. The size change operation can be performed by the function CACHE_setL2Size().

Figure 2-7 Changing L2 Cache Size During Runtime



To switch back to task A configuration, L2 cache line frames located in the 32K byte segment that is to be switched to SRAM have to be written back to external memory and invalidated. Since it is not known which external memory addresses are cached in these line frames, all of L2 is writeback–invalidated. This is automatically done by the cache controller when the cache size is switched. Code and global variables can then be copied back to their original location.

The exact procedures are given in Table 2-6. The same procedure applies to L1P and L1D caches.

Example 2-5 shows a C code example of how to change the L2 cache size. The corresponding linker command file is shown in Example 2-6.

Table 2-6 Procedure for Changing Cache Sizes for L1P, L1D, and L2

Switch To	Perform
More Cache (Less SRAM)	1. DMA or copy needed code/data out of SRAM addresses to be converted to cache. 2. Wait for completion of step 1. 3. Increase cache size usingCACHE_setL1pSize(), CACHE_setL1dSize(),or CACHE_setL2Size()
Less Cache (More SRAM)	1. Decrease Cache size usingCACHE_setL1pSize(),CACHE_setL1dSize(),or CACHE_setL2Size() 2. DMA or copy back any code/data needed. 3. Wait for completion of step 2.

Example 2-5 Changing L2 Cache Size Code

```
/* ----------------------------------------------------------------------*/
/* Buffer for Task A code and data in external memory */
/* ----------------------------------------------------------------------*/
#pragma DATA_SECTION(buffer_A, ".external")
unsigned char buffer_A[1024];
/* ----------------------------------------------------------------------*/
/* Main */
/* ----------------------------------------------------------------------*/
void main(void)
{
    int i;
    Uint32 id = DAT_XFRID_WAITNONE;
    /* ------------------------------------------------------------------*/
    /* Set L2 mode and open DAT*/
    /* ------------------------------------------------------------------*/

    CACHE_enableCaching(CACHE_CE00);
    CACHE_setL2Size(CACHE_0KCACHE);
    DAT_open(DAT_CHAANY, DAT_PRI_HIGH,0);
    /* ------------------------------------------------------------------*/
    /* Initialize state_A */
    /* ------------------------------------------------------------------*/
    for (i=0; i<N_STATE_A; i++)
    {
        state_A[i] = 1;
    }
    /* ------------------------------------------------------------------*/
    /* Task A -1*/
    /* ------------------------------------------------------------------*/
    process_A(state_A, N_STATE_A);
    process_AB(state_A, local_var_A, N_STATE_A);
    /* ------------------------------------------------------------------*/
    /* Switch to configuration for Task B with 32K cache: */
    /* 1) DMA needed code/data out of L2 SRAM addresses to be */
    /* converted to cache. */
    /* 2) Wait for completion of 1) */
    /* 3) Switch mode */
    /* Take address and word count information from map file */
    /* ------------------------------------------------------------------*/
    id = DAT_copy((void*)0x8000, buffer_A, 0x0120);
    DAT_wait(id);
    CACHE_setL2Size(CACHE_32KCACHE);
    /* ------------------------------------------------------------------*/
    /* Task B */
    /* Cache into L2, destroys code/data in the L2 segment that */
    /*previously was SRAM. */
    /* ------------------------------------------------------------------*/
    process_AB(ext_data_B, ext_data_B, N_DATA_B);
    /* ------------------------------------------------------------------*/
    /* Switch back to configuration for Task A with 0K cache */
    /* 1) Switch mode */
    /* 2)DMA back any code/data needed */
    /* 3) Wait for completion of 2) */
    /* Take address and word count information from map file */
    /* ------------------------------------------------------------------*/
    CACHE_setL2Size(CACHE_0KCACHE);
    id = DAT_copy(buffer_A, (void*)0x8000, 0x0120);
    DAT_wait(id);
    /* ------------------------------------------------------------------*/
    /* Task A -2 */
    /* ------------------------------------------------------------------*/
    process_A(state_A, N_STATE_A);
    process_AB(state_A,
    local_var_A, N_STATE_A);
    /* ------------------------------------------------------------------*/
    /* Exit*/
    /* ------------------------------------------------------------------*/
    DAT_close();
}
void process_A(unsigned char *x, int nx)
{
int i;
for (i=0; i<nx; i++)
            x[i] = x[i] * 2;
}
void process_AB(unsigned char *input, unsigned char *output, int size)
{
    int i;
```

```
    for (i=0; i<size; i++)
            output[i] = input[i] + 0x1;
}
```

End of Example 2-5

Example 2-6 Linker Command File for Changing L2 Cache Size Code

```
MEMORY
{
    L2_1: o = 00800000h l = 00008000h /*1st 32K segment: always SRAM */
    L2_2: o = 00808000h l = 00008000h /*2nd 32K segment:Task A-SRAM,Task B-Cache */
    CE0: o = 80000000h l = 01000000h /*external memory */
}
SECTIONS
{
    .cinit > L2_1
    .text > L2_1
    .stack > L2_1
    .bss > L2_1
    .const > L2_1
    .data > L2_1
    .far > L2_1
    .switch > L2_1
    .sysmem > L2_1
    .tables > L2_1
    .cio > L2_1
    .sram_state_A > L2_2
    .sram_process_A > L2_2
    .sram_local_var_A > L2_2
    .external > CE0
}
```

End of Example 2-6

Chapter 3

Optimizing for Cache Performance

This chapter describes cache optimization techniques from a programmer's point of view. The ideal scenario would be to have an application execute in a fast and large flat memory that is clocked at CORE speed. However, this scenario becomes more and more unrealistic the higher the CORE clock rate becomes. Introducing a cached-memory architecture inevitably causes some cycle count overhead compared to the flat memory model. However, since a cached-memory model enables the CORE to be clocked at a higher rate, the application generally executes faster (execution time = cycle count/clock rate). Still, the goal is to reduce the cache cycle overhead as much as possible. In some cases performance can be further improved by implementing algorithms with a cached architecture in mind.

3.1 Differences Between C66x and C64x DSP

Readers who are familiar with the C64x cache architecture may want to take note of features that are new or have changed for C66x devices. The features described in this chapter are listed below. For a complete list of new and changed features, see Appendix A on page A-1.

The width of the write buffer on C66x devices is increased to 128 bits; on C64x devices, the width is 64 bits. This results in fewer write buffer full stalls for write misses to sequential addresses that compensates for the lower draining rate of CORE/2 (CORE/1 on C64x devices).

The C66x devices add a tag update buffer that queues clean-to-dirty transitions to L2's copy of the L1D tag RAM (this, so called, shadow tag RAM is required for the snoop cache coherence protocol). Occasionally, this may result in buffer full stalls, if a stream of write hits makes previously clean cache lines dirty at a high rate.

The C66x devices add a high-bandwidth internal DMA (IDMA) between L1 and L2 that can be used to efficiently page data in and out of L1 SRAM. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for IDMA details.

Access and bank conflicts between different requestors are resolved according to the settings of the C66x bandwidth management. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for bandwidth management details.

The C66x cached controllers support cache freeze modes that prevent allocation of new lines. This can be particularly useful for L1P cache to prevent eviction of often reused code. See Section 3.4.3.1.

Due to higher stall counts per miss on C66x devices, eliminating misses and exploiting miss pipelining has become even more important. This is made easier on the C66x devices through the support of L1 SRAM (see Section 3.2.3), larger L1 cache capacity, low-overhead snooping, and noninclusivity of L2 cache.

As would be expected, the actual cache optimization methods are the same for C64x and C66x devices.

Exploiting L1D miss pipelining is critical for performance. On C64x devices, data miss pipelining reduces the stall count by 4; on C66x devices, data miss pipelining reduces the stall count by up to 7.5 (L2 SRAM with 0 wait states and 2 × 128-bit banking) or 9 (L2 SRAM with 1 wait states and 4 × 128-bit banking).

3.2 Cache Performance Characteristics

The performance of cache mostly relies on the reuse of cache lines. The access to a line in memory that is not yet in cache will incur CORE stall cycles. As long as the line is kept in cache, subsequent accesses to that line will not cause any stalls. Thus, the more often the line is reused before it is evicted from cache, the less impact the stall cycles will have. Therefore, one important goal of optimizing an application for cache performance is to maximize line reuse. This can be achieved through an appropriate memory layout of code and data, and altering the memory access order of the CORE. In order to perform these optimizations, you should be familiar with the cache memory architecture, in particular the characteristics of the cache memories such as line size, associativity, capacity, replacement scheme, read/write allocation, miss pipelining, and write buffer. These characteristics were described in Chapter 1. You also have to understand under what conditions CORE stalls occur and the cycle penalty associated with these stalls.

For this purpose, the next two sections present an overview of the C66x cache architecture detailing all important cache characteristics, cache stall conditions and associated stall cycles. These sections provide a useful reference for optimizing code for cache performance.

3.2.1 Stall Conditions

The most common stall conditions on C66x devices are:

- Cross Path Stall: When an instruction attempts to read a register via a cross path that was updated in the previous cycle, one stall cycle is introduced. The compiler automatically tries to avoid these stalls whenever possible.
- L1D Read and Write Hits: CORE accesses that hit in L1D SRAM or cache do not normally cause stalls, unless there is an access conflict with another requestor. Access priorities are governed by the bandwidth management settings. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for bandwidth management details. L1D requestors include CORE data access, IDMA or EDMA, snoops and cache coherence operations.
- L1D Cache Write Hits: CORE writes that hit in L1D cache do not normally cause stalls. However, a stream of write hits that make previously clean cache lines dirty at a high rate can cause stall cycles. The cause is a tag update buffer that queues clean-to-dirty transitions to L2's copy of the L1D tag RAM (this, so called, shadow tag RAM is required for the snoop cache coherence protocol).
- L1D Bank Conflict: L1D memory is organized in 8 × 32-bit banks. Parallel accesses that both hit in L1D and are to the same bank cause 1 cycle stall. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for special case exceptions.

- L1D Read Miss: Stall cycles are incurred for line allocations from L2 SRAM, L2 cache, or external memory. L1D read miss stalls can be lengthened by:
  - L2 Cache Read Miss: The data has to be fetched from external memory first. The number of stall cycles depends on the particular device and the type of external memory.
  - L2 Access/Bank Conflict: L2 can service only one request at a time. Access priorities are governed by the bandwidth management settings. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for bandwidth management details. L2 requestors include L1P (line fills), L1D (line fills, write buffer, tag update buffer, victim buffer), IDMA or EDMA, and cache coherence operations.
  - L1D Write Buffer Flush: If the write buffer contains data and a read miss occurs, the write buffer is first fully drained before the L1D read miss is serviced. This is required to maintain proper ordering of a write followed by a read. Write buffer draining can be lengthened by L2 access/bank conflicts and L2 cache write misses (the write buffer data misses L2 cache).
  - L1D Victim Buffer Writeback: If the victim buffer contains data and a read miss occurs, the contents are first written back to L2 before the L1D read miss is serviced. This is required to maintain proper ordering of a write followed by a read. The writeback can be lengthened by L2 access/bank conflicts.

Consecutive and parallel misses will be overlapped, provided none of the above stall lengthening condition occurs and the two parallel/consecutive misses are not to the same set.

- L1D Write Buffer Full: If an L1D write miss occurs and the write buffer is full, stalls occur until one entry is available. Write buffer draining can be lengthened by:
  - L2 Cache Read Miss: The data has to be fetched from external memory first. The number of stall cycles depends on the particular device and the type of external memory.
  - L2 Access/Bank Conflict: L2 can service only one request at a time. Access priorities are governed by the bandwidth management settings. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for bandwidth management details. L2 requestors include L1P (line fills), L1D (line fills, write buffer, tag update buffer, victim buffer), IDMA or EDMA, and cache coherence operations.
  - L1P Read Hits: CORE accesses that hit in L1P SRAM or cache do not normally cause stalls, unless there is an access conflict with another requestor or the access is to L1P ROM with wait-states. Access priorities are governed by the bandwidth management settings. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for bandwidth management details. L1P requestors include CORE program access, IDMA or EDMA, and cache coherence operations.

- L1P Read Miss: Stall cycles are incurred for line allocations from L2 SRAM, L2 cache, or external memory. L1P read miss stalls can be lengthened by:
  - L2 Cache Read Miss: The data has to be fetched from external memory first. The number of stall cycles depends on the particular device and the type of external memory.
  - L2 Access/Bank Conflict: L2 can service only one request at a time. Access priorities are governed by the bandwidth management settings. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for bandwidth management details. L2 requestors include L1P (line fills), L1D (line fills, write buffer, tag update buffer, victim buffer), IDMA or EDMA, and cache coherence operations.

Consecutive misses will be overlapped, provided none of the above stall lengthening condition occurs.

Figure 3-1 shows the C66x memory architecture detailing all important characteristics, stall conditions and associated stall cycles.

Figure 3-1 C66x Cache Memory Architecture

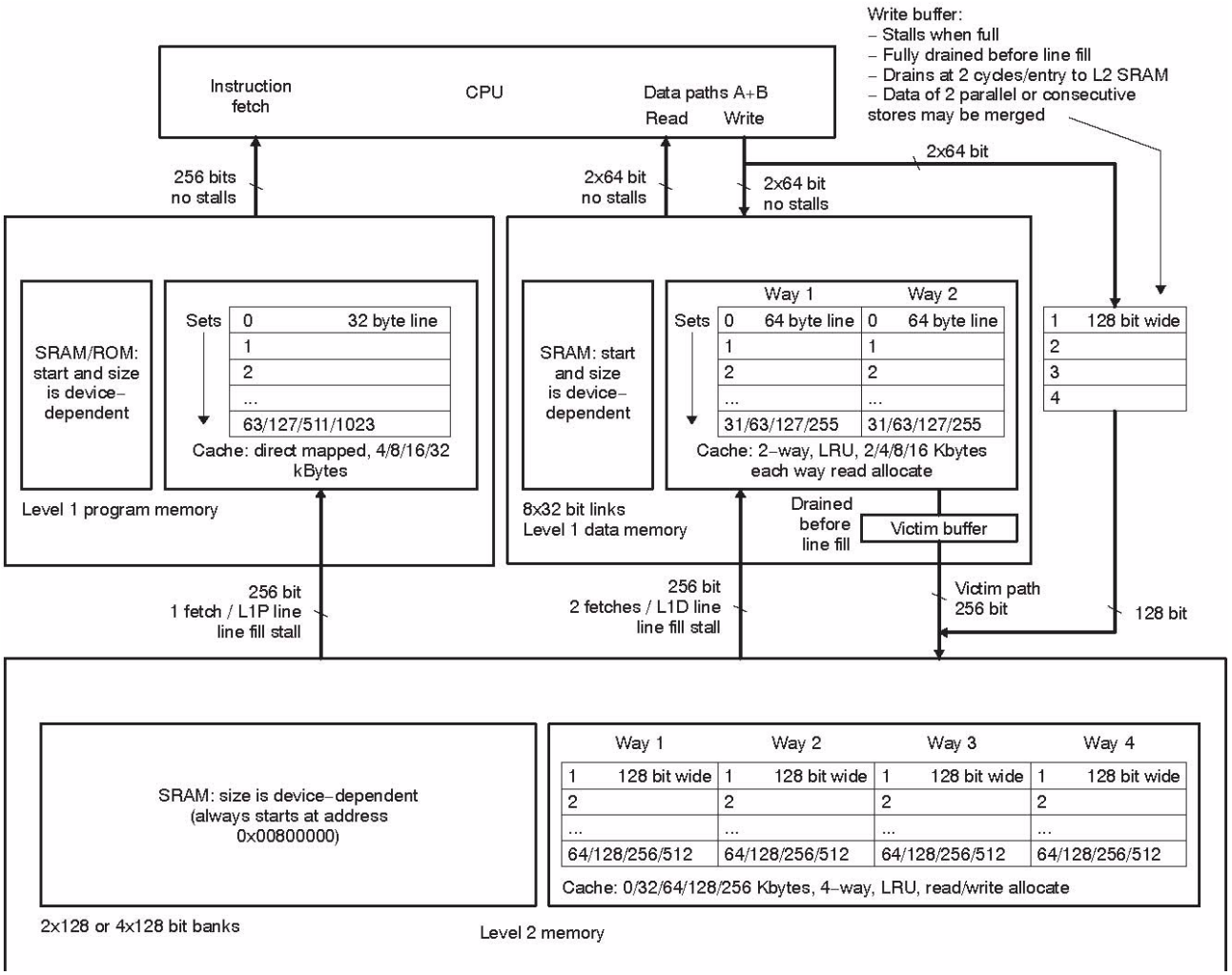


Table 3-1 and Table 3-2 list the actual stall cycles for two different device configurations. One configuration is for devices that have an L2 with 0 wait-states and 2 × 128-bit banks. The other configuration is for devices that have an L2 with 1 wait-state and 4 × 128-bit banks. See your device-specific data manual to determine the L2 type for a particular device.

Note on the notation of stall cycles: L1D cache stall cycles for C66x devices are sometimes specified as averages due to a varying L2 clock alignment relative to the memory access. Because L2 is clocked at CORE/2, a memory access occurring out-of-phase with the L2 clock reduces the number of stall cycles by

1. For instance, a read miss may cost either 10 or 11 stall cycles depending on clock phase alignment. This is then noted as 10.5 stall cycles.

Table 3-1 L1P Miss Pipelining Performance (Average Number of Stalls per Execute Packet)

Instructions per Execute Packet	L2 Type			
	0 Wait-State, 2 × 128-bit Banks		1 Wait-State, 4 × 128-bit Banks	
	L2 SRAM	L2 Cache	L2 SRAM	L2 Cache
1	0.000	0.000	0.000	0.000
2	0.001	0.497	0.167	0.499
3	0.501	1.247	0.751	1.249
4	0.997	1.997	1.329	1.999
5	1.499	2.747	1.915	2.749
6	2.001	3.497	2.501	3.499
7	2.497	4.247	3.079	4.249
8	2.999	4.997	3.665	4.999

Table 3-2 L1D Performance Parameters (Number of Stalls)

Parameter	L2 Type			
	0 Wait-State, 2 × 128-bit Banks		1 Wait-State, 4 × 128-bit Banks	
	L2 SRAM	L2 Cache	L2 SRAM	L2 Cache
Single Read Miss	10.5	12.5	12.5	14.5
2 Parallel Read Misses (pipelined)	10.5 + 4	12.5 + 8	12.5 + 4	14.5 + 8
M Consecutive Read Misses (pipelined)	10.5 + 3 × (M -1)	12.5 + 7 × (M -1)	12.5 + 3 × (M -1)	14.5 + 7 × (M -1)
M Consecutive Parallel Read Misses (pipelined)	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
Victim Buffer Flush on Read Miss	disrupts miss pipelining plus maximum 11 stalls	disrupts miss pipelining plus maximum 11 stalls	disrupts miss pipelining plus maximum 10 stalls	disrupts miss pipelining plus maximum 10 stalls
Write Buffer Drain Rate	2 cycles/entry	6 cycles/entry	2 cycles/entry	6 cycles/entry

3.2.2 C66x Pipelining of L1D Read Misses

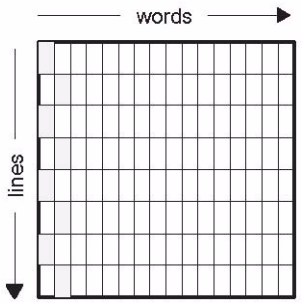
The C66x cache architecture pipelines L1D read misses and allows parallel and consecutive read miss stall cycles to be overlapped. See Table 3-2 for the stall cycle formulas.

This mechanism is further described in the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x. Miss pipelining will be disrupted, if the L1D stall is lengthened by any of the conditions listed in Section 3.2.1. Note that when accessing memory sequentially, misses are not overlapped since on a miss one full cache line is allocated and subsequent accesses will hit. Therefore, to achieve full overlapping of stalls, you have to access two new cache lines every cycle, that is, step through memory in strides that are equal to the size of two cache lines. This is realized in the assembly routine "touch", that can be used to allocate length bytes of a memory buffer *array into L1D. The routine loads (or touches) one byte each of two consecutive cache lines in parallel. To avoid bank conflicts, the two parallel loads are offset by one word. The access pattern is illustrated in Figure 3-2. The assembly routine is shown in Example 3-1.

If a line does not reside in L1D, the load will miss and the line allocated in L1D. If the line already was allocated, there is no effect. The data read by the load is not used. The routine takes (0.5 × M + 16) cycles plus any stall cycles for "Consecutive Parallel Read Misses" to allocate M lines.

Example: Consider a device with an L2 type with 0 wait-states and 2 × 128-bit banking. To allocate a 32K-byte array using the touch loop, it takes (0.5 × M + 16) + (10.5 + 4 × (M/2 -1) + 3 × M/2). With M = 32768 bytes/64-byte cache line size = 512, this is 2070.5 cycles. On the other hand, if each line had been allocated individually, this would have taken 512 × 10.5 = 5376 cycles, or 2.6× the number of cycles.

Figure 3-2 Memory Access Pattern of Touch Loop



Example 3-1 Touch Assembly Routine

```
* ========================================================================== *
* TEXAS INSTRUMENTS, INC. *
* NAME *
* touch *
* PLATFORM *
* C64x *
* USAGE *
* This routine is C callable, and has the following C prototype: *
* void touch *
* ( *
* const void *array, /* Pointer to array to touch */ *
* int length /* Length array in bytes */ *
* ); *
* This routine returns no value and discards the loaded data. *
* DESCRIPTION *
* The touch() routine brings an array into the cache by reading *
* elements spaced one cache line apart in a tight loop. This *
* causes the array to be read into the cache, despite the fact *
* that the data being read is discarded. If the data is already *
* present in the cache, the code has no visible effect. *
* When touching the array, the pointer is first aligned to a cache- *
* line boundary, and the size of the array is rounded up to the *
* next multiple of two cache lines. The array is touched with two *
* parallel accesses that are spaced one cache-line and one bank *
```

```
* apart. A multiple of two cache lines is always touched. *
* MEMORY NOTE *
* The code is ENDIAN NEUTRAL. *
* No bank conflicts occur in this code. *
* CODESIZE *
* 84 bytes *
* CYCLES *
* cycles = MIN(22, 16 + ((length + 124) / 128)) *
* For length = 1280, cycles = 27. *
* The cycle count includes 6 cycles of function-call overhead, but *
* does NOT include any cycles due to cache misses. *
.global _touch
.sect ".text:_touch"
_touch
B .S2 loop ; Pipe up the loop
|| MVK .S1 128, A2 ; Step by two cache lines
|| ADDAW .D2 B4, 31, B4 ; Round up # of iters
B .S2 loop ; Pipe up the loop
|| CLR .S1 A4, 0, 6, A4 ; Align to cache line
|| MV .L2X A4, B0 ; Twin the pointer
B .S1 loop ; Pipe up the loop
|| CLR .S2 B0, 0, 6, B0 ; Align to cache line
|| MV .L2X A2, B2 ; Twin the stepping constant
B .S2 loop ; Pipe up the loop
|| SHR .S1X B4, 7, A1 ; Divide by 128 bytes
|| ADDAW .D2 B0, 17, B0 ; Offset by one line + one word
[A1] BDEC .S1 loop, A1 ; Step by 128s through array
|| [A1] LDBU .D1T1 *A4++[A2], A3 ; Load from [128*i + 0]
|| [A1] LDBU .D2T2 *B0++[B2], B4 ; Load from [128*i + 68]
|| SUB .L1 A1, 7, A0
loop:
[A0] BDEC .S1 loop, A0 ; Step by 128s through array
|| [A1] LDBU .D1T1 *A4++[A2], A3 ; Load from [128*i + 0]
|| [A1] LDBU .D2T2 *B0++[B2], B4 ; Load from [128*i + 68]
|| [A1] SUB .L1 A1, 1, A1
BNOP .S2 B3, 5 ; Return
* ========================================================================== *
* End of file: touch.asm *
* -------------------------------------------------------------------------- *
* Copyright © 2001 Texas Instruments, Incorporated. *
* All Rights Reserved. *
* ========================================================================== *
```

End of Example 3-1

3.2.3 Optimization Techniques Overview

The focus of this user's guide is on efficient use of the L1 caches. Since L1 characteristics (capacity, associativity, line size) are more restrictive than those of L2 cache, optimizing for L1 almost certainly implies that L2 cache is also used efficiently. Typically, there is not much benefit in optimizing only for L2 cache. It is recommended to use L2 cache for the general-purpose parts of the application with largely unpredictable memory accesses (general control flow, etc.). L1 and L2 SRAM should be used for time-critical signal processing algorithms. Data can be directly streamed into L1 SRAM using EDMA or IDMA, or into L2 SRAM using EDMA. Memory accesses can then be optimized for L1 cache.

There are two important ways to reduce the cache overhead:

1. Reduce the number of cache misses (in L1P, L1D, and L2 cache): This can be achieved by:
   a. Maximizing cache line reuse:
      i. Access *all* memory *locations* within a cached line. Since the data was allocated in cache causing expensive stall cycles, it should be used.
      ii. The *same* memory *locations* within a cached line should be reused as often as possible. Either the same data can be reread or new data written to already cached locations so that subsequent reads will hit.
   b. Avoiding eviction of a line as long as it is being reused:

i. Evictions can be *prevented*, if data is allocated in memory such that the number of cache ways is not exceeded when it is accessed. (The number of ways is exceeded if more lines map to the same set than the number of cache ways available.)

ii. If this is not possible, evictions may be *delayed* by separating accesses to the lines that cause the eviction further apart in time.

iii. Also, one may have lines *evicted* in a *controlled* manner relying on the LRU replacement scheme such that only lines that are no longer needed are evicted.

2. Reduce the number of stall cycles per miss: This can be achieved by exploiting miss pipelining.

Methods for reducing the number of cache misses and number of stalls per miss are described in this chapter.

A good strategy for optimizing cache performance is to proceed in a top-down fashion, starting on the application level, moving to the procedural level, and if necessary considering optimizations on the algorithmic level. The optimization methods for the application level tend to be straightforward to implement and typically have a high impact on overall performance improvement. If necessary, fine tuning can then be performed using lower level optimization methods. Hence, the structure of this chapter reflects the order that one may want to address the optimizations.

3.3 Application-Level Optimizations

On an application and system level, the following considerations are important for good cache performance.

3.3.1 Streaming to External Memory or L1/L2 SRAM

For streaming data from/to a peripheral or coprocessor using DMA, it is recommended to allocate the streaming buffers in L1 or L2 SRAM. This has several advantages over allocating the buffers in external memory:

1. L1 and L2 SRAM are closer to the CORE; therefore, latency is reduced. If the buffers were located in external memory, data would be first written from the peripheral to external memory by the DMA, cached by L2, then cached by L1D, before reaching the CORE
2. Cache coherence is automatically maintained by the cache controller for data accesses to L2 SRAM (and is not applicable at all to L1 SRAM). If the buffers are located in external memory, you have to take care to maintain coherence by manually issuing L2 cache coherence operations. In some cases, buffers may have to be allocated in external memory due to memory capacity restrictions. Section 2.4 describes in detail how to manage cache coherence.
3. No additional latency due to coherence operations. The latency can be thought of as adding to the time required for processing the buffered data. In a typical double buffering scheme, this has to be taken into account when choosing the size of the buffers.

For rapid-prototyping applications, where implementing DMA double-buffering schemes are considered too time consuming and would like to be avoided, allocating all code and data in external memory and using L2 as All Cache may be an appropriate way. Following the simple rules for using L2 cache coherence operations described in Section 2.4, this is a fast way to get an application up and running without the need to perform DSP-style optimizations. Once the correct functioning of the application has been verified, bottlenecks in the memory management and critical algorithms can be identified and optimized.

3.3.2 Using L1 SRAM

C66x devices provide L1D and L1P SRAM that may be used for code and data that is sensitive to cache penalties, for instance:

- Performance critical code or data
- Code or data that is shared by many algorithms
- Code or data that is accessed frequently
- Functions with large code size or large data structures
- Data structures with irregular accesses that would make cache less efficient
- Streaming buffers (for example, on devices where L2 is small and better configured as cache)

Since the size of L1 SRAM is limited, the decision of what code and data to allocate in L1 SRAM needs to be made carefully. Allocating large amount of L1 SRAM may require reducing L1 cache size that could mean lower performance for code and data in L2 and external memory.

L1 SRAM size can be kept smaller if code and data can be copied to L1 SRAM as required, making use of code and/or data overlays. IDMA can be used to very-fast page in code or data from L2 SRAM. If code/data is to be paged in from external memory, EDMA must be used. However, very-frequent paging may add more overhead than caching. So a trade-off must be found between the SRAM and cache size.

3.3.3 Signal Processing versus General-Purpose Processing Code

It may be beneficial to distinguish between DSP-style processing and general-purpose processing in an application.

Since control and data flow of DSP processing are usually well understood, the code better lends itself to a more careful optimization than general-purpose code. General-purpose processing is typically dominated by straight-line execution, control flow, and conditional branching. This code typically does not exhibit much parallelism and execution depends on many conditions and tends to be largely unpredictable. That is, data memory accesses are mostly random, and access to program memory is linear with many branches. This makes optimization much more difficult. Therefore, in the case when L2 SRAM is insufficient to hold code and data of the entire application, it is recommended to allocate general-purpose code and associated data in external memory and allow L2 cache to handle memory accesses. This makes more L2 SRAM memory available for performance-critical signal processing code. Due to the unpredictable nature of general-purpose code, L2 cache should be made as large as possible. The cache that can be configured between 32K bytes and 256K bytes.

DSP code and data may benefit from being allocated in L2 SRAM or L1 SRAM. Allocation in L2 SRAM reduces cache overhead and gives you more control over memory accesses since only level 1 cache is involved whose behavior is easier to analyze. This allows you to make some modifications to algorithms in the way the CORE is accessing data, and/or to alter data structures to allow for more cache-friendly memory access patterns.

Allocation in L1 SRAM eliminates any caching altogether and requires no memory optimization except for bank conflicts.

3.4 Procedural-Level Optimizations

Procedural-level optimizations are concerned with changing the way data and functions are allocated in memory, and the way functions are called. No changes are made to individual algorithms, that is algorithms (for example, FIR filters, etc.) that were implemented for a flat memory model are used as is. Only the data structures that are accessed by the algorithm are optimized to make more efficient use of cache. In most cases these type of optimizations are sufficient, except for some algorithms such as the FFT whose structure has to be modified in order to take advantage of cache. Such a cache-optimized FFT is provided in the C66x DSP Library (DSPLIB).

The goal is to reduce the number of cache misses and/or the stall cycles associated with a miss. The first can be achieved by reducing the amount of memory that is being cached (see Section 3.4.1) and reusing already cached lines. Reuse can be achieved by avoiding evictions and writing to preallocated lines. Stall cycles of a miss can be reduced by exploiting miss pipelining.

We can distinguish between three different read miss scenarios:

1. All data/code of the working set fits into cache (no capacity misses by definition), but conflict misses occur. The conflict misses can be eliminated by allocating the code or data contiguously in memory. This is described in Section 3.4.3 and Section 3.4.4.
2. The data set is larger than cache, contiguously allocated, and not reused. Conflict misses occur, but no capacity misses (because data is not reused). The conflict misses can be eliminated, for instance by interleaving cache sets. This is discussed in Section 3.4.5.
3. The data set is larger than cache, capacity misses (because same data is reused) and conflict misses occur. Conflict and capacity misses can be eliminated by splitting up data sets and processing one set at a time. This method is referred to as blocking or tiling and is discussed in Section 3.4.6.

Avoiding stalls that are caused directly or indirectly by the write buffer are described in Section 3.4.7.

Processing chains, in which the results of one algorithm form the input of the next algorithm, provide an opportunity to eliminate all cache misses except for the compulsory misses of the first algorithm in the chain. This is explained in Section 3.4.2. A more comprehensive example that demonstrates this important concept is provided in Section 3.4.2.

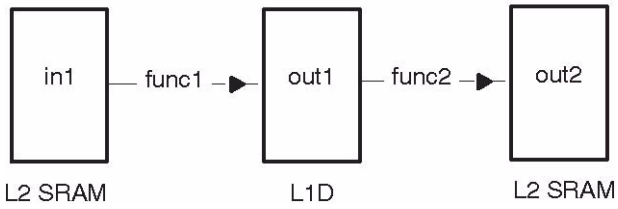
3.4.1 Reduce Memory Bandwidth Requirements by Choosing Appropriate Data Type

It should be ensured that memory-efficient data types are chosen. For instance, if the data is maximum 16-bits wide, it should be declared as short rather than *integer*. This halves the memory requirements for the array, which also reduces the number of compulsory misses by a factor of 2. This typically only requires a minor change in the algorithm to accept the new data type. Additionally, the algorithm is likely to execute much faster, since smaller data containers may allow SIMD optimizations to be performed by the compiler. Especially in the cases where an application is ported from another platform to a DSP system, inefficient data types may exist.

3.4.2 Processing Chains

Often the results of one algorithm form the input of the next algorithm. If the algorithms operate out-of-place (that is, the results are placed in an array different from the input), the input array gets allocated in L1D, but the output is passed through the write buffer to next lower memory level (L2 or external memory). The next algorithm then again suffers miss penalties when reading the data. On the other hand, if the output of the first algorithm were written to L1D, then the data could be directly reused from cache without incurring cache stalls. There are many possible configurations for processing chains. The concept is shown in Figure 3-3.

Figure 3-3 Processing Chain with Two Functions



Consider Example 3-2, a 4-channel filter system consisting of a FIR filter followed by a dot product. The FIR filter in the first iteration allocates in[ ] and h[ ] in L1D and writes out[ ] to L2 SRAM. Subsequently, out[ ] and w[ ] are allocated in L1D by the dotprod routine. For the next iteration, the FIR routine writes its results to L1D, rather L2 SRAM, and the function dotprod does not incur any read misses.

In total, four arrays, in[ ], h[ ], out[ ], and w[ ] are allocated in L1D. If it is assumed that the total data working set required for one iteration fits into L1D, conflict misses can still occur if more than two of the arrays map to the same sets (since L1D is 2-way set-associative). As described in Section 3.4.4, these arrays should be allocated contiguously in memory to avoid conflict misses. What exact memory allocation is chosen depends on the size of the arrays and the capacity of L1D.

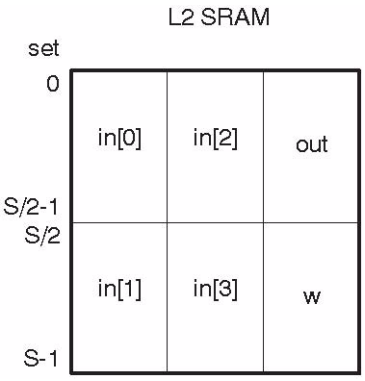
Example 3-2 Channel FIR/Dot Product Processing Chain Routine

```
#define NX NR+NH-1
short in [4][NX]; /* input samples */
short out [NR]; /* FIR output */
short w [NR]; /* weights for dot product */
short h [4][NH]; /* FIR filter coefficients */
short out2 ; /* final output */
for (i=0; i<4; i++)
{
fir(in[i], h[i], out, NR, NH);
out2 = dotprod(out, w, NR);
}
```

End of Example 3-2

The number of input samples, NX, shall be chosen such that the array occupies about one–forth of L1D. We assume that NH filter taps occupy two cache lines. The number of output samples produced is then NR = NX – NH + 1. Figure 3-4 shows how the individual arrays map to the L1D cache sets. We can neglect the coefficient array since it occupies only 4 × NH = 8 cache lines. It can be seen that within one iteration no more that two arrays map the same sets, that is, no conflict misses will occur. Capacity misses will also not occur since the total size of the data set accessed within one iteration fits into L1D.

Figure 3-4 Memory Layout for Channel FIR/Dot Product Processing Chain Routine



3.4.3 Avoiding L1P Conflict Misses

In this read miss scenario, all code of the working set fits into cache (no capacity misses by definition), but conflict misses occur. This section first explains how L1P conflict misses are caused, then describes how the conflict misses can be eliminated by allocating the code contiguously in memory.

The L1P set number is determined by the memory address modulo the capacity divided by the line size. Memory addresses that map to the same set and are not contained in the same cache line will evict one another.

Compiler and linker do not give considerations to cache conflicts, and an inappropriate memory layout may cause conflict misses during execution. This section describes how most of the evictions can be avoided by altering the order in which functions are linked in memory. Generally, this can be achieved by allocating code that is accessed within some local time window contiguously in memory.

Consider the code in Example 3-3. Assume that function_1 and function_2 have been placed by the linker such that they overlap in L1P, as shown in Figure 3-5. When function_1 is called the first time, it is allocated in L1P causing three misses (1). A following call to function_2 causes its code to be allocated in L1P, resulting in five misses (2). This also will evict parts of the code of function_1, lines 3 and 4, since these lines overlap in L1P (3). When function_1 is called again in the next iteration, these lines have to be brought back into L1P, only to be evicted again by function_2. Hence, for all following iterations, each function call causes two misses, totaling four L1P misses per iteration.

These type of misses are called conflict misses. They can be completely avoided by allocating the code of the two functions into nonconflicting sets. The most straightforward way this can be achieved is to place the code of the two functions contiguously in memory (4).

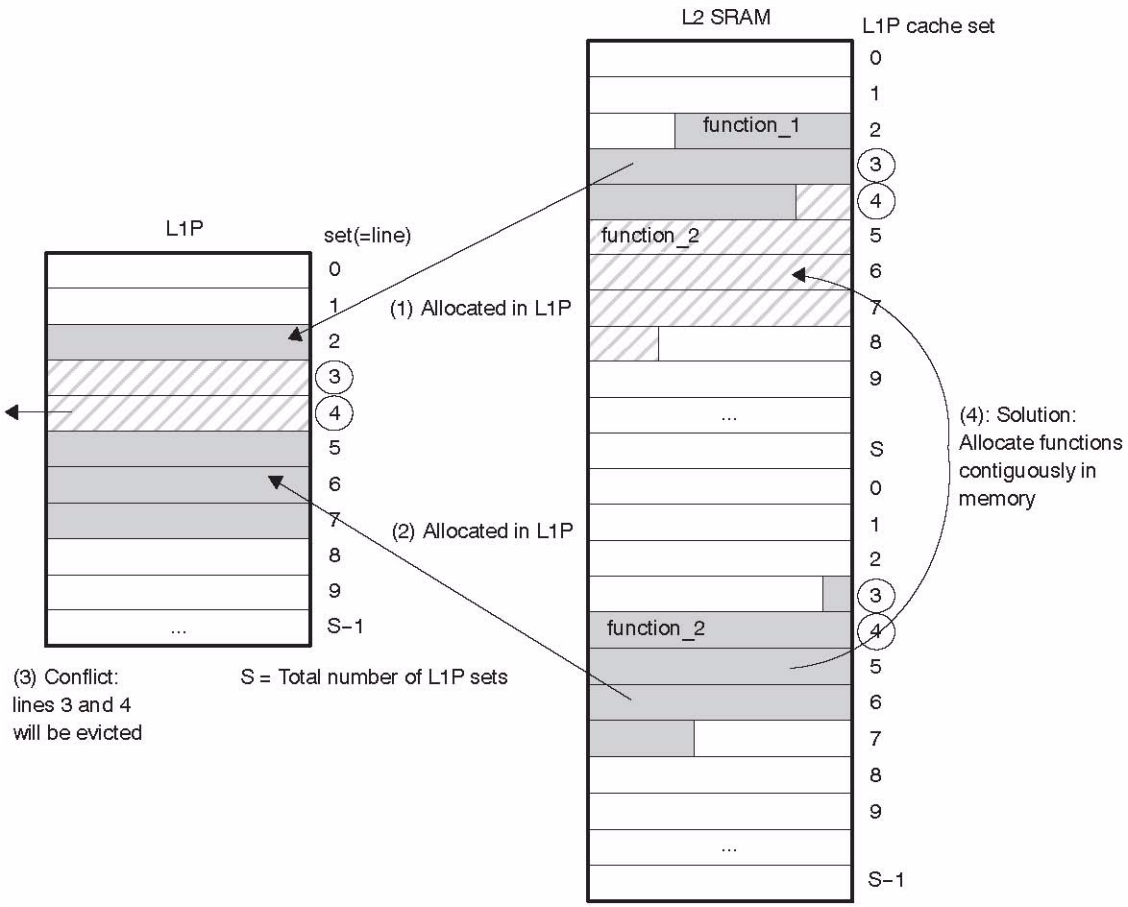
Note that it also would be possible to move function_2 to any place where none of its sets conflicts with function_1. This would prevent eviction as well; however, the first method has the advantage that you do not need to worry about absolute address placement, but can simply change the order in which the functions are allocated in memory.

Example 3-3 L1P Conflicts Code

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

End of Example 3-3

Figure 3-5 Avoiding L1P Evictions



Note—With code generation tools 5.0 (CCS 3.0) and later, the GROUP directive **must** be used to force a specific link order.

There are two ways for allocating functions contiguously in memory:

- Use the compiler option –mo to place each C and linear assembly function into its own individual section (assembly functions have to be placed in sections using the .sect directive). Inspect the map file to determine the section names for the functions chosen by the compiler. In the example, the sections names are .text:_function_1 and .text:_function_2. Now, the linker command file can be specified as:

```
MEMORY
```

```
{
        L2SRAM: o = 00800000h l = 00010000h
        EXT_MEM: o = 80000000h l = 01000000h
}
SECTIONS
{
            .cinit > L2SRAM
            .GROUP > L2SRAM
        {
        .text:_function_1
        .text:_function_2
        .text
}
        .stack > L2SRAM
        .bss > L2SRAM
        .const > L2SRAM
        .data > L2SRAM
        .far > L2SRAM
        .switch > L2SRAM
        .sysmem > L2SRAM
        .tables > L2SRAM
        .cio> L2SRAM
        .external > EXT_MEM
}
```

The linker will link all sections in exactly the order specified within the GROUP statement. In this case, the code for function_1 is followed by function_2 and then by all other functions located in the section .text. No changes are required in the source code. However, be aware that using the -mo compiler option can result in overall code size growth because any section containing code will be aligned at a 32–byte boundary.

Note that the linker can only place entire sections, but not individual functions that reside in the same section. In case of precompiled libraries or object files that have multiple functions in a section or were compiled without -mo, there is no way to reassign individual functions to different sections without recompiling the library.

- To avoid the disadvantage of using –mo, only the functions that require contiguous placement may be assigned individual sections by using the #pragma CODE_SECTION before the definition of the functions:

```
#pragma CODE_SECTION(function_1,".funct1")
#pragma CODE_SECTION(function_2,".funct2")
void function_1(){...}
void function_2(){...}
```

The linker command file would then be specified as:

```
...
SECTIONS
{
        .cinit > L2SRAM
        .GROUP > L2SRAM
    {
        .funct1 .funct2
        .text
    }
        .stack > L2SRAM
...
}
```

Those functions should be considered for reordering that are repeatedly called within the same loop, or within some time frame.

If the capacity of the cache is not sufficient to hold all functions of a loop, the loop may have to be split up in order to achieve code reuse without evictions. This may increase the memory requirements for temporary buffers to hold output data. Assume that the combined code size of function_1 and function_2, as shown in Example 3-4, is larger

than the size of L1P. In Example 3-5, the code loop has been split so that both functions can be executed from L1P repeatedly, considerably reducing misses. However, the temporary buffer tmp[ ] now has to hold all intermediate results from each call to function_1.

Example 3-4 Combined Code Size is Larger than L1P

```
for (i=0; i<N; i++)
{
    function_1(in[i], tmp);
    function_2(tmp, out[i]);
}
```

End of Example 3-4

Example 3-5 Code Split to Execute from L1P

```
for (i=0; i<N; i++)
{
    function_1(in[i], tmp[i]);
}
for (i=0; i<N; i++)
{
    function_2(tmp[i], out[i]);
}
```

End of Example 3-5

3.4.3.1 Freezing L1P Cache

The C66x cache controllers allow you to put caches into freeze mode that prevents allocation of new lines. After freezing, the contents of cache will not be evicted by conflicts (note that all other cache actions behave as normal, for example, dirty bit updates, LRU updates, snooping, cache coherence operations).

The freeze mode of L1P cache can be controlled through the CSL functions:

CACHE_freezeL1p();

CACHE_unfreezeL1p();

This allows code to be forcefully retained in cache. Generally this is useful if code that is reused would be evicted in between by other code that is executed only once, such as interrupt service routines. Not caching code that is not reused has no impact on its performance, and at the same time eliminates misses on cached code that is reused.

An exception may be code that contains non-SPLOOP loops, since every iteration would miss. On the other hand, SPLOOP loops do not suffer from this problem since they are executed from the CORE internal loop buffer. For more information on SPLOOP, refer to the *TMS320C66x CPU and Instruction Set Reference Guide* "Related Documentation from Texas Instruments" on page ø-x.

3.4.4 Avoiding L1D Conflict Misses

In this read miss scenario, all data of the working set fits into cache (no capacity misses by definition), but conflict misses occur. This section first explains how L1D conflict misses are caused, then describes how the conflict misses can be eliminated by allocating data contiguously in memory.

The L1D set number is determined by the memory address modulo the capacity of one cache way divided by the line size. In a direct-mapped cache such as L1P, these addresses would evict one another if those addresses are not contained in the same cache line. However, in the 2-way set-associative L1D, two conflicting lines can be kept in cache without causing evictions. Only if another third memory location is allocated that maps to that same set, one of the previously allocated lines in this set will have to be evicted (which one will be evicted is determined by the least-recently-used rule).

Compiler and linker do not give considerations to cache conflicts, and an inappropriate memory layout may cause conflict misses during execution. This section describes how most of the evictions can be avoided by altering the memory layout of arrays. Generally, this can be achieved by allocating data that is accessed within the same local time window contiguously in memory.

Optimization methods similar to the ones described for L1P in Section 3.4.3 can be applied to data arrays. However, the difference between code and data is that L1D is a 2-way set-associative cache and L1P is direct-mapped. This means that in L1D, two data arrays can map to the same sets and still reside in L1D at the same time. The following example illustrates the associativity of L1D.

Consider the dotprod routine shown in Example 3-6 that computes the dot product of two input vectors.

Example 3-6 Dot Product Function Code

```
int dotprod
(
    const short *restrict x,
    const short *restrict h,
    int nx
)
{
    int i, r = 0;
    for (i=0;i<nx; i++)
    {
        r += x[i] * h[i];
    }
    return r;
}
```

End of Example 3-6

Assume we have two input vectors in1 and in2, and two coefficient vectors w1 and w2. We would like to multiply each of the input vectors with each of the coefficient vectors, in1 × w1, in2 × w2, in1 × w2, and in2 × w1. We could use the following call sequence of dotprod to achieve this:

```
r1 = dotprod(in1, w1, N);
r2 = dotprod(in2, w2, N);
r3 = dotprod(in1, w2, N);
r4 = dotprod(in2, w1, N);
```

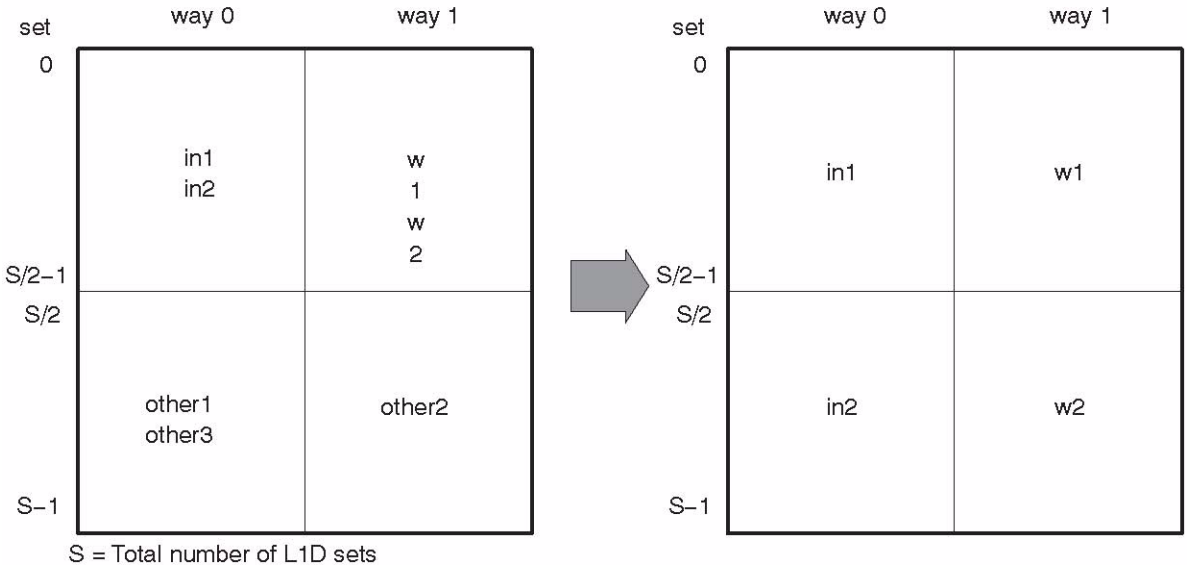
Further assume that each array is one-fourth the total L1D capacity, such that all four arrays fit into L1D. However, assume that we have given no consideration to memory layout and declared the arrays as:

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

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

The arrays other1, other2, and other3 are used by other routines in the same application. It is assumed that the arrays are allocated contiguously in the section .data in the order they are declared. The assigned addresses can be verified in the map file (generated with the option –m). Since each way in L1D is half the size of the total capacity, all memory locations that are the size of one way apart map to the same set. In this case, in1, in2, w1, and w2 all map to the same sets in L1D. A layout for L1D is shown on the left in Figure 3-6. Note that this is only one possible configuration of many. The exact configuration depends on the start address of the first array, in1, and the state of the LRU bit (which decides the way the line is allocated). However, all configurations are equivalent in terms of cache performance.

Figure 3-6 Mapping of Arrays to L1D Sets for Dot Product Example



The first call to dotprod allocates in1 and w1 into L1D, as shown in Figure 3-6. This causes S compulsory misses, where S is the total number of sets. The second call causes in1 and w1 to be evicted and replaced with in2 and w2, which causes another S misses. The third call reuses w2, but replaces in2 with in1 resulting in S/2 misses. Finally, the last call again causes S misses, because in1 and w2 are replaced with in2 and w1.

To reduce the read misses, we can allocate the arrays contiguously in memory as follows:

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

We grouped together the definitions of the arrays that are used by the routine. Now all arrays, in1, in2, w1, and w2 can fit into L1D as shown on the right in Figure 3-6. Note that due to the memory allocation rules of the linker, it cannot always be assured that consecutive definitions of arrays are allocated contiguously in the same section (for example, const arrays will be placed in the .const section and not in .data). Therefore, the arrays must be assigned to a user-defined section, for instance:

```
#pragma DATA_SECTION(in1, ".mydata")
#pragma DATA_SECTION(in2, ".mydata")
#pragma DATA_SECTION(w1, ".mydata")
#pragma DATA_SECTION(w2, ".mydata')
#pragma DATA_ALIGN(in1, 32)
short in1 [N];
short in2 [N];
short w1 [N];
short w2 [N];
```

Additionally, the arrays are aligned at a cache line boundary to save some extra misses.

Note that it may be necessary to align the arrays at different memory banks to avoid bank conflicts, for example:

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

Exploiting miss pipelining can further reduce the cache miss stalls. The touch loop discussed in Section 3.2.2 is used to preallocate all arrays, in1, in2, w1, and w2, in L1D. Since all arrays are allocated contiguously in memory, one call of the touch routine is sufficient:

```
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);
```

3.4.5 Avoiding L1D Thrashing

In this read miss scenario, the data set is larger than cache, contiguously allocated, but data is not reused. Conflict misses occur, but no capacity misses (since data is not reused). This section describes how the conflict misses can be eliminated, for instance, by interleaving cache sets.

Thrashing is caused if more than two read misses occur to the same set evicting a line before all of its data was accessed. Provided all data is allocated contiguously in memory, this condition can only occur if the total data set accessed is larger than the L1D capacity. These conflict misses can be completely eliminated by allocating the data set contiguously in memory and pad arrays as to force an interleaved mapping to cache sets.

Consider the weighted dot product routine shown in Example 3-7.

Example 3-7 Weighted Dot Product

```
int w_dotprod(const short *restrict w, const short *restrict x, const short
*restrict h, int N)
{
    int i, sum = 0;
    _nassert((int)w % 8 == 0);
    _nassert((int)x % 8 == 0);
    _nassert((int)h % 8 == 0);
```

```
    #pragma MUST_ITERATE(16,,4)
    for (i=0; i<N; i++) sum += w[i] * x[i] * h[i];
        return sum;
}
```

End of Example 3-7

If the three arrays w[ ], x[ ], and h[ ] are allocated in memory such that they are all aligned to the same set, L1D thrashing occurs. The contents of the L1D set, at the time when an access is made, is listed in Table 3-3. It can be seen that whenever an array element is attempted to be read, it is not contained in L1D. Consider the first iteration of the loop, all three arrays are accessed and cause three read misses to the same set. The third read miss evicts a line just allocated by one of the two previous read misses. Assume that first w[0] and then x[0] is accessed, causing one full line of w[ ] and x[ ] to be allocated in L1D. If there was no further allocation to the same set, accesses to w[1] and x[1] in the next iteration would be cache hits. However, the access to h[0] causes the line of w[ ] allocated by the previous access to w[0] to be evicted (because it was least-recently-used) and a line of h[ ] to be allocated in its place. In the next iteration, w[1] causes a read miss, evicting the line of x[ ]. Next, x[1] is accessed that was just evicted, causing another read miss and eviction of the line of h[ ]. This pattern repeats for every iteration of the loop. Since each array is evicted just before its line is reused, every single read access in the routine causes a read miss.

Table 3-3 Contents of an L1D Set at the Time When an Array is Accessed (Weighted Dot Product Example)

Read Access To	Way 0	Way 1	LRU
w[0]	–	–	0
x[0]	w	–	1
h[0]	w	x	0
w[1]	h	x	1
x[1	]h	w	0
h[1]	x	w	1

These conflict misses can be completely eliminated by allocating the data set contiguously in memory and pad arrays as to force an interleaved mapping to cache sets. For instance:

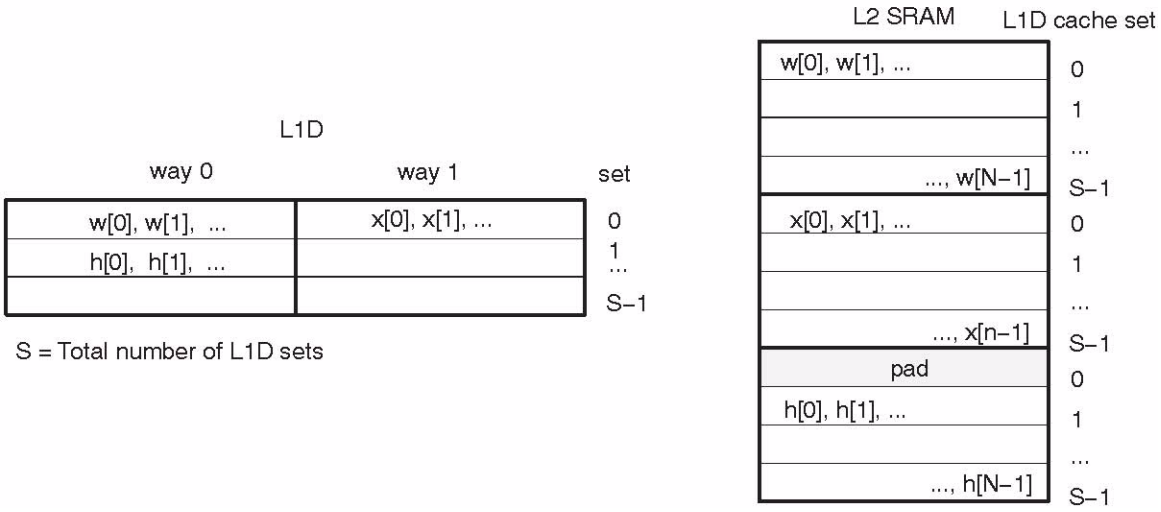
```
#pragma DATA_SECTION(w, ".mydata")
#pragma DATA_SECTION(x, ".mydata")
#pragma DATA_SECTION(pad,".mydata")
#pragma DATA_SECTION(h, ".mydata")
#pragma DATA_ALIGN (w, CACHE_L1D_LINESIZE)
short w [N]; short x [N];
char pad [CACHE_L1D_LINESIZE];
short h [N];
```

The linker command file would then be specified as:

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

This causes allocation of the array h[ ] in the next set, thus avoiding eviction of w[ ]. Now all three arrays can be kept in L1D. This memory configuration is shown in Figure 3-7. The line of array h[ ] will be only evicted when the data of one line has been consumed and w[ ] and x[ ] are allocated in the next set. Eviction of h[ ] is irrelevant since all data in the line has been used and will not be accessed again.

Figure 3-7 Memory Layout and Contents of L1D After the First Two Iterations



3.4.6 Avoiding Capacity Misses

In this read miss scenario, data is reused, but the data set is larger than cache causing capacity and conflict misses. These misses can be eliminated by splitting up data sets and processing one subset at a time. This method is referred to as blocking or tiling.

Consider the dot product routine that is called four times with one reference vector and four different input vectors:

```
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);
```

Assume that each array is twice the L1D capacity. We expect compulsory misses for in1[ ] and w[ ] for the first call. For the remaining calls, we expect compulsory misses for in2[ ], in3[ ], and in4[ ], but would like to reuse w[ ] from cache. However, after each call, the beginning of w[ ] has already been replaced with the end of w[ ], since the capacity is insufficient. The following call then suffers again misses for w[ ].

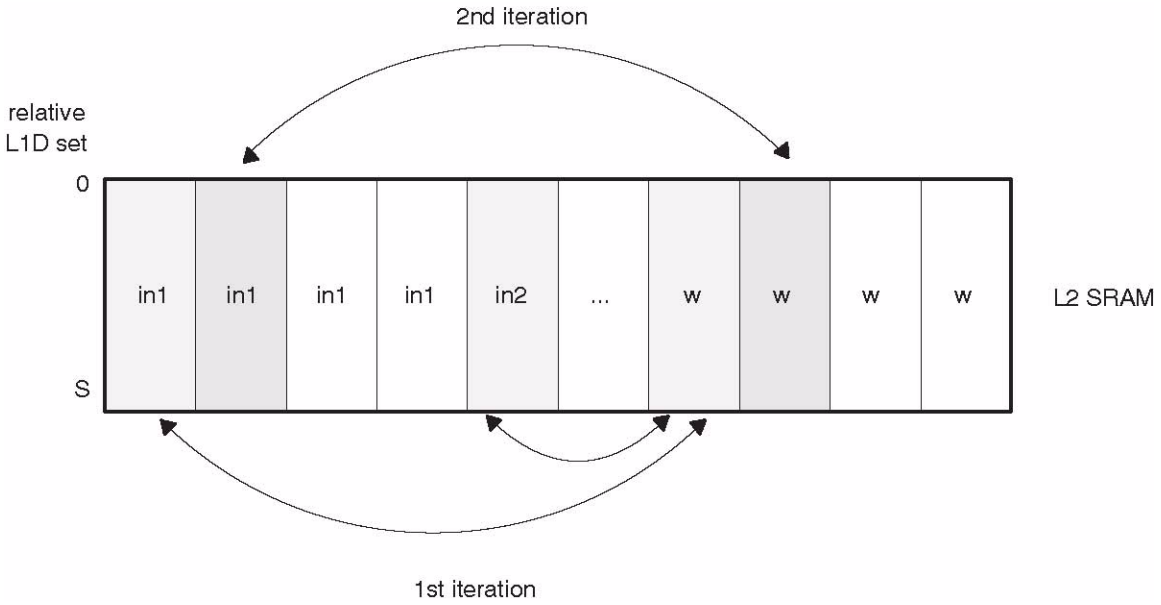
The goal is to avoid eviction of a cache line before it is reused. We would like to reuse the array w[ ]. This memory configuration is shown in Figure 3-8. The first line of w[ ] will be the first one to be evicted when the cache capacity is exhausted. In this example, the cache capacity is exhausted after N/4 outputs have been computed, since this required N/4 × 2 arrays = N/2 array elements to be allocated in L1D. If we stop

processing in1[ ] at this point and start processing in2[ ], we can reuse the elements of w[ ] that we just allocated in cache. Again, after having computed another N/4 outputs, we skip to processing in3[ ] and finally to in4[ ]. After that, we start computing the second N/4 outputs for in1[ ], and so on.

The restructured code for the example would look like this:

```
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);
}
```

Figure 3-8 Memory Layout for Dotprod Example



We can further reduce the number of read miss stalls by exploiting miss pipelining. The touch loop is used to allocate w[ ] once at the start of the iteration; then before each call of dotprod, the required input array is allocated:

```
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);
}
```

It is important to note that the LRU scheme automatically retains the line that hits (w[ ] in this case), as long as two lines in the same set are always accessed in the same order. (Assume that way 0 in set X is accessed before way 1 in set X. The next time set X is accessed, it should be in the same order: way 0, then way 1). This LRU behavior cannot

be assured if the access order changes. Example: If after dotprod array w[ ] is LRU and array in[ ] is MRU, w[ ] was accessed before in[ ]. If the next dotprod accesses w[ ] first again, the access will hit and the line of w[ ] turns MRU and is protected from eviction. However, if now the touch loop is used, in[ ] is accessed before w[ ]. Accesses to in[ ] will miss and evict w[ ] since it is LRU. Therefore, it has to be ensured that after each dotprod w[ ] is MRU. This is achieved by retouching w[ ] before allocating the next in[ ] with touch. This forces w[ ] to become MRU and is protected form eviction. The extra touch loop will not cost many cycles since no cache misses occur, that is, (number of lines)/2 + 16 cycles.

In this example, arrays w[ ] and in[ ] should be aligned to different memory banks to avoid bank conflicts.

```
#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];
```

3.4.7 Avoiding Write Buffer Related Stalls

The L1D write buffer can be the cause for additional stalls. Generally, write misses do not cause stalls since they pass through the write buffer to the lower level memory (L2 or external memory). However, the depth of the write buffer is limited to four entries. To make more efficient use of each 128-bit wide entry, the write buffer merges consecutive write misses to sequential addresses into the same entry. If the write buffer is full and a another write miss occurs, the CORE stalls until an entry in the buffer becomes available. Also, a read miss causes the write buffer to be completely drained before the miss is serviced. This is necessary to ensure proper read-after-write ordering (the read that caused the miss may access data that is still in the write buffer). The number of cycles it takes to drain the write buffer adds to the normal read miss stall cycles. For additional information, see the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x.

Write buffer related stalls can be easily avoided by allocating the output buffer in L1D cache. Writes will then hit in L1D rather than being passed on to the write buffer. Consider the constant-vector add routine in Example 3-8.

Example 3-8 Add Constant to Vector Function

```
void vecaddc(const short *restrict x, short c, short *restrict r, int nx)
{
    int i;
    for (i = 0 ; i < nx; i++) r[i] = x[i] + c;
}
```

End of Example 3-8

Assume the scenario shown in Example 3-9. A constant c is added to four input vectors in[4][N] and the results are then used to compute the dot product with the reference vector ref[ ].

In the first iteration, vecaddc may suffer read miss stalls for allocating in[0], and write buffer stalls while writing results to out[ ]. Also, dotprod will see read miss stalls for out[ ] and ref[ ]. If arrays out[ ] and ref[ ] can be retained in L1D for the remaining iterations, only compulsory misses for in[ ] will be incurred. Since out[ ] is now allocated in L1D, writes will hit instead of passing through the write buffer.

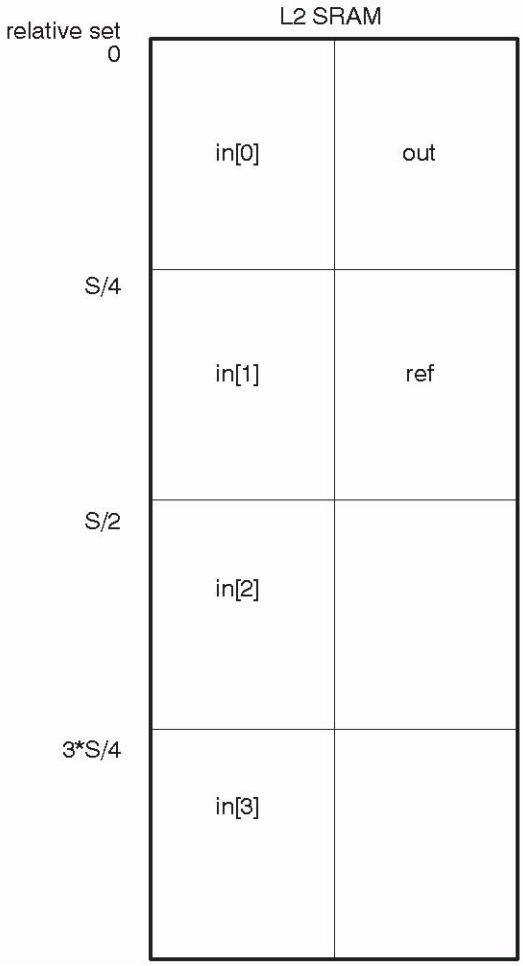
Example 3-9 Vecaddc/Dotprod Code

```
short in[4][N];
short out [N];
short ref [N];
short c, r; for (i=0; i<4; i++)
{
    vecaddc(in[i], c, out, N);
    r = dotprod(out, ref, N);
}
```

End of Example 3-9

The size of each array shall be 2048 elements such that one array occupies one-eighth of L1D, as shown in Figure 3-9. An optimized C version of the vecaddc routine was used that computes eight results every 2 cycles in the inner loop, that is, it takes N/4 cycles to execute plus some cycles for set-up code. Thus, we expect to see 512 execute cycles for vecaddc. The routine accesses 2048 elements, 4096 bytes spanning 64 cache lines. Assuming an L2 type with 1 wait state and 4 × 128-bit banks, we expect to see 64 misses × 12.5 stalls = 800 stall cycles. Additionally, there will be write buffer related stalls. Two STDW instructions are issued every 2 cycles in the kernel. When the output array is not in L1D (for the first iteration in Example 3-9), the write buffer fills at an average rate of one entry every 2 cycles because the two double words are merged into one entry. Since the write buffer drains at the same rate, there will not be any write buffer full conditions. However, every time a read miss occurs, the write buffer will be drained completely to maintain proper program ordering. Due to support for write merging, the write buffer does not generally suffer write buffer full stalls, except when there is a stream of write misses occurring out of order.

Figure 3-9 Memory Layout for Vecaddc/Dotprod Example



The interaction of write buffer related stalls and read misses is listed in Table 3-4. Consider the loop prolog and kernel shown in Example 3-10. Every other cycle, 16 bytes are read from the input array. Therefore, after 8 execute cycles, 16 bytes × 8/2 cycles = 64 bytes are consumed which equals one cache line. The write buffer entries shall be denoted A, B, C, ..., etc. In the first execution cycle of the prolog, one read miss and one read hit occurs that costs 12.5 stall cycles. The subsequent 3 LDDW||LDDW's hit in L1D. The write buffer starts filling up in execute cycle 8 (the predicate for STW on cycle 6 is false). On execute cycle 9, the next read miss occurs. The write buffer still contains A that needs to be drained taking one cycle. Then the write buffer starts filling again. The pattern from execute cycle 9 to 16 now repeats. In summary, we expect to see the following number of L1D stall cycles:

$12.5 + ((12.5 + 1) \times 63) = 863$

The dotprod routine sees 128 read misses since it accesses 4096 elements. We expect to see 128 misses × 12.5 cycles = 1600 stall cycles.

For iterations 2 to 4, vecaddc will only suffer read miss stalls for the in[ ] array. Any write buffer related stalls will no longer occur since the output array was allocated in L1D by the dotprod routine in the previous iteration. Also, the dotprod routine will not incur any stalls since both out[ ] and ref[ ] arrays are held in L1D.

Table 3-4 Interaction of Read Miss and Write Buffer Activity for the First Call of Vecaddc (n = 0 to 62)

Execute Cycle	Read Activity	Write Buffer Contents
1	read miss	–
2	–	–
3	hit	–
4	–	–
5	hit	–
6	–	–
7	hit	–
8	–	A
9 + 8 × n	read miss, 1 write buffer drain stall	A
10+8×n	–	B
11+8×n	hit	B
12+8×n	–	C
13+8×n	hit	C
14+8×n	–	D
15+8×n	hit	D
16+8×n	–	E

Example 3-10 C64x Assembly Code for Prolog and Kernel of Routine vecaddc

```
- - - - - - - - - - - - - - - - - - - - - - - - -
L1: ; PIPED LOOP PROLOG
LDDW .D2T2 *++B9(16),B7:B6 ; (P) |10|
|| [ A0] BDEC .S1 L2,A0 ; (P)
|| LDDW .D1T1 *A8++(16),A5:A4 ; (P) |10|
ZERO .D1 A1
PACK2 .L1 A3,A3,A3
|| LDDW .D2T2 *++B9(16),B7:B6 ; (P) @|10|
|| [ A0] BDEC .S1 L2,A0 ; (P) @
|| LDDW .D1T1 *A8++(16),A5:A4 ; (P) @|10|
SUB .D2X A6,8,B8
|| MV .D1 A6,A9
|| MVKH .S1 0x10000,A1 ; init prolog collapse predicate
;**--------------------------------------------------------------------------*
L2: ; PIPED LOOP KERNEL
ADD2 .S2X B7,A3,B5 ; |10|
|| [ A0] BDEC .S1 L2,A0 ; @@
|| LDDW .D1T1 *A8++(16),A5:A4 ; @@@|10|
|| LDDW .D2T2 *++B9(16),B7:B6 ; @@@|10|
[ A1] MPYSU .M1 2,A1,A1 ;
|| [!A1] STDW .D1T1 A7:A6,*A9++(16) ; |10|
|| [!A1] STDW .D2T2 B5:B4,*++B8(16) ; |10|
|| ADD2 .S2X B6,A3,B4 ; @|10|
|| ADD2 .S1 A5,A3,A7 ; @|10|
|| ADD2 .L1 A4,A3,A6 ; @|10|
;**--------------------------------------------------------------------------*
```

End of Example 3-10

```
- - - - - - - - - - - - - - - - - - - - - - - - -
```

3.5 On-Chip Debug Support

The C66x devices support a feature that allows read-out of the cache tag RAM (on earlier version of some C66x devices, this feature is only supported on simulator platforms). This feature is exposed in Code Composer Studio IDE (version 3.2 or higher) through the Cache Tag RAM Viewer. The viewer displays for each cache line the cache type, set number, way number, valid/dirty/LRU bits and the line address (with symbols). This allows you to analyze cache behavior by single-stepping through the algorithm and observing the changes in the cache. This helps with choosing the appropriate optimization method and verifying the results of the optimization.

Appendix A

Cache Differences Between C66x DSP and C64x DSP

Readers who are familiar with the C64x cache architecture may want to take note of features that are new or have changed for C66x devices. The features described in this user guide are listed in Table A-1.

Table A-1 Cache Differences Between C66x DSP and C64x DSP (Part 1 of 2)

Feature	Difference
Memory Sizes and Types	On C66x devices, each L1D and L1P implement SRAM additionally to cache. The size of cache is user-configurable and can be set to 4K, 8K, 16K, or 32K bytes. The amount of available SRAM is device dependent and specified in the device-specific data sheet. On C64x devices, only cache with a fixed size of 16K bytes is implemented.
	On C66x devices, the maximum possible size of L2 is increased. See the device-specific data sheet for the actual amount of available L2 memory. L2 cache size configurations are the same as on C64x devices.
Cacheability	The cacheability settings of external memory addresses (through MAR bits) only affect L1D and L2 caches on C66x devices; that is, program fetches to external memory addresses are always cached in L1P regardless of the cacheability setting. This is not the case on C64x devices, where the settings affects all caches, L1P, L1D, and L2.
	The cacheability control of external memory addresses covers the entire external address space on C66x devices. In contrast, on C64x devices only a subset of the address space is covered.
Snooping Protocol	The snooping cache coherence protocol on C66x devices directly forwards data to L1D cache and the DMA. On C64x devices, invalid and writeback cache lines to maintain coherence. The C66x snooping mechanism is more efficient since it eliminates cache miss overhead caused by invalidates.
	The snoop coherence protocol on C66x devices does not maintain coherence between L1P cache and L2 SRAM, as is the case on C64x devices. This is the responsibility of the programmer.
Cache Coherence Operations	On C66x devices, the L2 cache coherence operations always operate on L1P and L1D, even if L2 cache is disabled. This is not the case on C64x devices, which requires the explicit use of L1 coherence operations.
	C66x devices support a complete set of range and global L1D cache coherence operations. In contrast, C64x devices support only L1D range invalidate and writeback-invalidate.
	On cache size changes, C66x devices automatically writeback–invalidate cache before initializing it with the new size. In contrast, C64x devices required an explicit writeback–invalidate to be issued by the programmer (however, this is handled as part of the CSL function).
	On C66x devices, L2 cache is non-inclusive of L1D and L1P. This means that a line eviction from L2 will not cause the corresponding lines in L1P and L1D to be evicted. However, this is the case on C64x devices. The advantage of non-inclusivity is that line allocations in L2 due to program fetches will not evict data from L1D cache, and line allocations in L2 due to data accesses will not evict program code from L1P. This helps reduce the number of cache misses.

Table A-1 Cache Differences Between C66x DSP and C64x DSP (Part 2 of 2)

Feature	Difference
Cache Performance and Optimization	The width of the write buffer on C66x devices is increased to 128 bits; on C64x devices, the width is 64 bits. This results in fewer write buffer full stalls for write misses to sequential addresses that compensates for the lower draining rate of CPU/2 (was CPU/1 on C64x DSP).
	The C66x devices add a tag update buffer that queues clean-to-dirty transitions to L2's copy of the L1D tag RAM (this so-called shadow tag RAM is required for the snoop cache coherence protocol). Occasionally this may result in buffer full stalls, if a stream of write hits makes previously clean cache lines dirty at a high rate.
	C66x devices add a high-bandwidth internal DMA (IDMA) between L1 and L2 that can be used to efficiently page data in and out of L1 SRAM. See the *TMS320C66x CorePac User Guide* in for details on the IDMA.
	Access and bank conflicts between different requestors are resolved according to the settings of C66x bandwidth management. See the *TMS320C66x CorePac User Guide* in "Related Documentation from Texas Instruments" on page ø-x for details on bandwidth management.
	C66x cached controllers support cache freeze modes that prevents allocation of new lines. This can be particularly useful for L1P cache to prevent eviction of often reused code. See Section 3.4.3.1.
	Due to higher stall counts per miss on C66x devices, eliminating misses and exploiting miss pipelining has become even more important. This is made easier on the C66x device through the support of L1 SRAM (see Section 3.3), larger L1 cache capacity, low-overhead snooping and non-inclusivity of L2 cache.
	As would be expected, the actual cache optimization methods are the same for C64x and C66x DSPs.
	Exploiting L1D miss pipelining is critical for performance. Whereas on C64x DSP data miss pipelining reduced the stall count by 4; on C66x DSP, the stall count is reduced by up to 7.5 (L2 SRAM with 0 wait-state and 2 × 128-bit banking) or 9 (L2 SRAM with 1 wait-state and 4 × 128-bit banking).

Appendix B

C66x DSP Cache Coherence

In the cases where no hardware coherence protocol exists, it is the programmer's responsibility to maintain cache coherence. For this purpose, C66x DSP memory controllers support cache coherence operations that can be initiated by the program. The coherence operations include:

- Invalidate (INV): Evicts cache lines and discards data.
- Writeback (WB): Writes back data, lines stay in cache and are marked as clean.
- Writeback–Invalidate (WBINV): Writes back data and evicts cache lines.

They are available for L1P, L1D, and L2 cache. Note that L2 coherence operations always operate first on L1P and L1D.

Table B-1 and Table B-2 show the coherence matrices for the C66x DSP memory system. If a copy of a physical address (L2 SRAM or external memory) exists in cache at the time of a write access by a source entity, the coherence matrices indicate how the data written is made visible to the read access by the destination entity. This is achieved by different methods:

1. Forward the new data to a cache or memory visible to the destination entity: snoop–write, L1D WB/WBINV, L2 WB/WBINV.
2. Forward the new data directly to the destination entity: snoop–read.
3. Remove the copy from cache to make the memory containing the new data visible to the destination entity: L1P INV, L1D INV/WBINV, L2 INV/WBINV.

Part of making data visible to the destination is also ensuring that the data is not corrupted by any eviction of dirty lines. Evictions could overwrite data written by another entity, if the addresses written are for some reason still dirty in cache. Evictions are part of general CORE memory activity and are not generally predictable. How this is achieved is noted in the coherence matrices.

Note that in order to practically meet some of the conditions set out in the coherence matrices, a cache line must not contain any false addresses, that is, only contains addresses that are meant to be operated on by the coherence operation. This is achieved by aligning the start and end address of buffers at cache line boundaries. See the following section for further details.

Note—Practically, some conditions can only be assured if there are no false addresses or stray CORE accesses. See text box in Table B-1 and Table B-2 for details.

Table B-1 Coherence Matrix for L2 SRAM Addresses

Source	Destination	Location of Line at the Time of the Write Access	
(Write Access)	(Read Access)	L1P Cache	L1D Cache
DMA	DMA	No action required since inherently coherent (L1P cache does not affect visibility).	L1D WB, INV, or WBINV to avoid potential corruption of newly written data: Line must not be dirty at the time of the DMA write access.
	CORE Data Path	No action required since inherently coherent (L1P cache does not affect visibility).	Snoop-write: Data written to L2 SRAM and directly forwarded to L1D cache.
	CORE Fetch Path	L1P INV for visibility: Line must be invalid at the time of the first CORE fetch access after the write.	L1D WB, INV, or WBINV to avoid potential corruption of newly written code: Line must not be dirty at the time of the DMA write access.
CORE Data Path	DMA	No action required since inherently coherent (L1P cache does not affect visibility).	Snoop-read: Data directly forwarded to DMA without updating L2 SRAM.
	CORE Data Path	No action required since inherently coherent (L1P cache does not affect visibility).	No action required since inherently coherent.
	CORE Fetch Path	L1P INV for visibility: Line must be invalid at the time of the first CORE fetch access after the write.	L1D WB or WBINV for visibility: Dirty line with new code must have been written back by the time the fetch access is made.

Table B-2 Coherence Matrix for an External Memory Address (Part 1 of 2)

Source	Destination	Address Location at the Time of the Write Access		
(Write Access)	(Read Access)	L1P Cache	L1D Cache	L2 Cache
DMA/Other	DMA/Other	No action required since inherently coherent (L1P cache does not affect visibility).	L1D WB, INV, or WBINV to avoid potential corruption of newly written data: Line must not be dirty at the time of the DMA/other write access.	L2 WB, INV, or WBINV to avoid potential corruption of newly written data: Line must not be dirty at the time of the DMA/other write access.
	CORE Data Path	No action required since inherently coherent (L1P cache does not affect visibility).	L1D WB, INV, or WBINV to avoid potential corruption of newly written data: Line must not be dirty at the time of the DMA/other write access. L1D INV or WBINV for visibility: Line must be invalid at the time of the first CORE read access after the write.	L2 WB, INV, or WBINV to avoid potential corruption of newly written data: Line must not be dirty at the time of the DMA/other write access. L2 INV or WBINV for visibility: Line must be invalid at the time of the first CORE read access after the write.
	CORE Fetch Path	L1P INV for visibility: Line must be invalid at the time of the first CORE fetch access after the write.	L1D WB, INV, or WBINV to avoid corruption of newly written code: Line must not be dirty at the time of the DMA/other write access.	L2 WB, INV or WBINV to avoid potential corruption of newly written code: Line must not be dirty at the time of the DMA/other write access. L2 INV or WBINV for visibility: Line must be invalid at the time of the first CORE fetch access after the write.

Table B-2 Coherence Matrix for an External Memory Address (Part 2 of 2)

Source	Destination	Address Location at the Time of the Write Access		
(Write Access)	(Read Access)	L1P Cache	L1D Cache	L2 Cache
CORE Data Path	DMA/Other	No action required since inherently coherent (L1P cache does not affect visibility).	L1D WB or WBINV for visibility: Dirty line with new data must have been written back by the time the DMA/other read access is made.	L2 WB or WBINV for visibility: Dirty line with new data must have been written back by the time the DMA/other read access is made.
	CORE Data Path	No action required since inherently coherent (L1P cache does not affect visibility).	No action required since inherently coherent.	No action required since inherently coherent.
	CORE Fetch Path	L1P INV for visibility: Line must be invalid at the time of the first CORE fetch access after the write.	L1D WB or WBINV for visibility: Dirty line with new code must have been written back by the time the CORE fetch access is made. No action required since inherently coherent.	No action required since inherently coherent.

The most common scenario is DMA-to-data and data-to-DMA. Examples for the DMA-to-fetch case are code overlays and for the data-to-fetch case code overlays, copying boot code (memcpy), and self-modifying code. DMA-to-DMA is an atypical use case. Consider for instance, data written by a DMA to an address in external memory that is destined for the CORE data path. If at the time of writing a copy of the address is held in L2 cache, first, any potential corruption of the new data through dirty line evictions must be avoided and, second, the new data must be made visible (readable) to the CORE data path since it is written "underneath" L2 cache. Data corruption can be avoided by making the line clean (through writeback) or removing it from cache altogether (through invalidate). Visibility is achieved by invalidating the address, so that a CORE read access picks up the new data from external memory rather than the old data in L2 cache. Practically, you would not operate on individual lines as the coherence matrices might suggest. Coherence operations rather are initiated on blocks of addresses by specifying the start address and the length.

Note that stray CORE accesses can reverse the effects of coherence operations. It is assumed here that they do not exist or have been eliminated. If not, then a stray access could potentially reallocate and/or redirty a line just before or even during a DMA/other access. The results of this are unpredictable.

In order to assure the requirements set out in the coherence matrices, there are some important practical implications:

- Any requirements for visibility can be assured if the block coherence operation is initiated any time after the last write and completes before the first read access to that block.
- The requirement for visibility "Line must be invalid at the time of the first read/fetch access after the write" can also be assured if the block coherence operation is completed before the first write and there are no false addresses. See the following section for further details on false addresses.
- The requirement for avoiding data corruption, that is, "Line must not be dirty at the time of the DMA/other write access" can be assured if the block coherence operation completes before the first write access by the DMA/other, but only if there are no false addresses. See the following section for further details on false addresses.

- To avoid data corruption through the use of the invalidate operation (without writeback), false addresses must be eliminated. See the following section for further details on false addresses.

Some considerations that simplify the use of coherence operations:

- It must be assumed that an address is held in all caches, since it is generally not known where an individual address is held. Thus, all coherence operations should be performed for a given source–destination scenario. Practically however, initiating an L2 coherence operation is sufficient in the case of external memory addresses, since any L2 cache coherence operation implicitly operates first on L1D and L1P. The exception is the data-to-fetch path scenario for which separate L1D and L1P coherence operations need to be performed (note that this applies to L2 SRAM as well as external memory addresses).
- If it is certain that DMA/other never writes to lines dirty in cache, writing back or invalidating the line before the DMA/other access is not required.
- The two coherence operations required for visibility and avoidance of data corruption can be collapsed into one by completing an INV or WBINV before the first write access by the DMA/other. Again, this only works if there are no false addresses.

The following figures show the correct timing for the use of user-initiated cache coherence operations in each scenario.

Figure B-1 External Memory: DMA Write, CORE Read (Data)

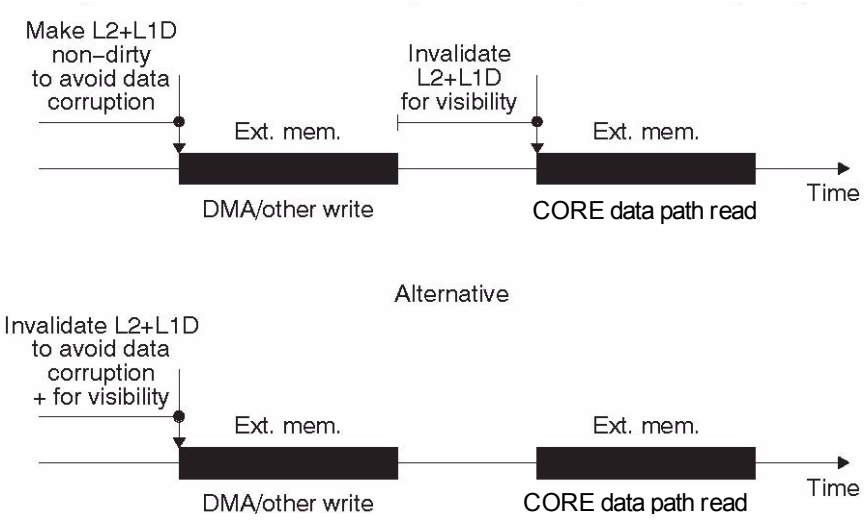


Figure B-2 External Memory: DMA Write, CORE Read (Code)

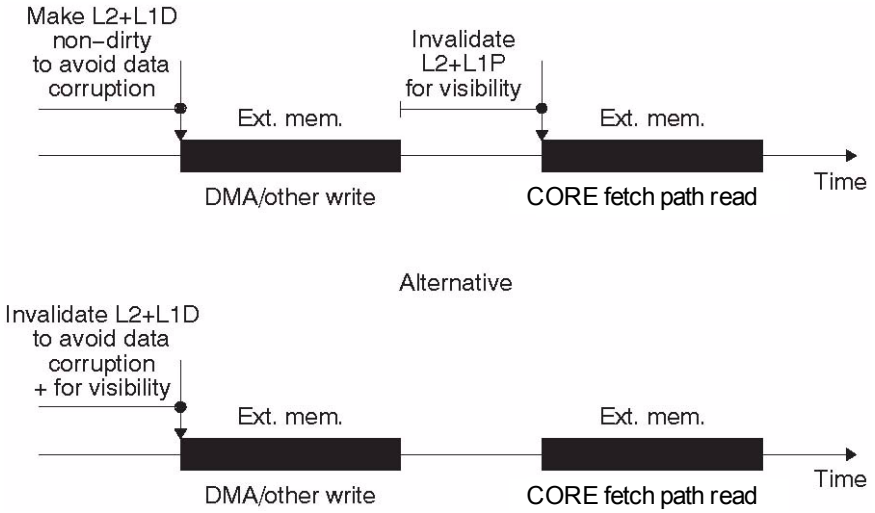


Figure B-3 External Memory: CORE Write, DMA Read (Data)

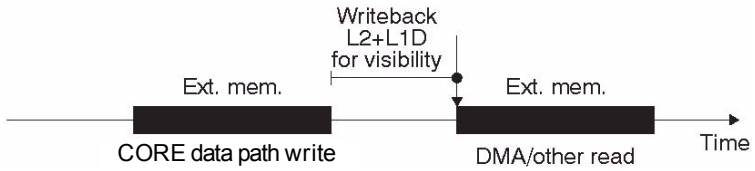


Figure B-4 L2 SRAM/External Memory: CORE Write (Data), CORE Read (Code)

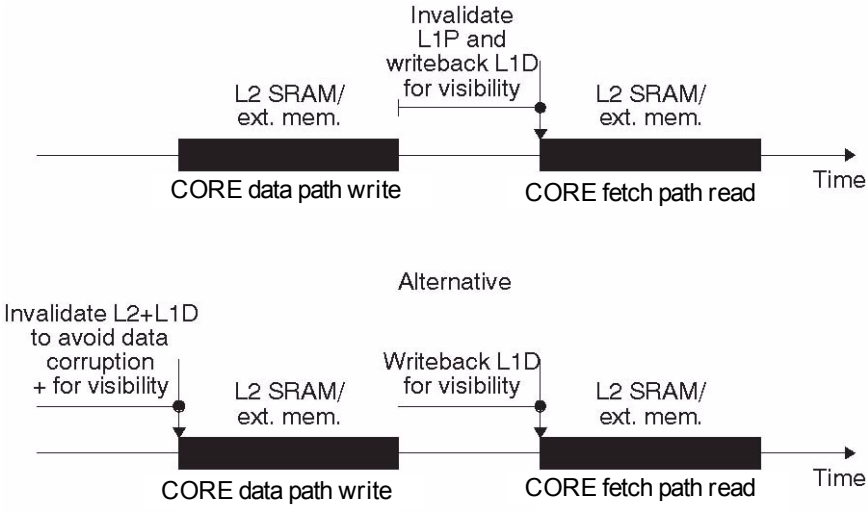
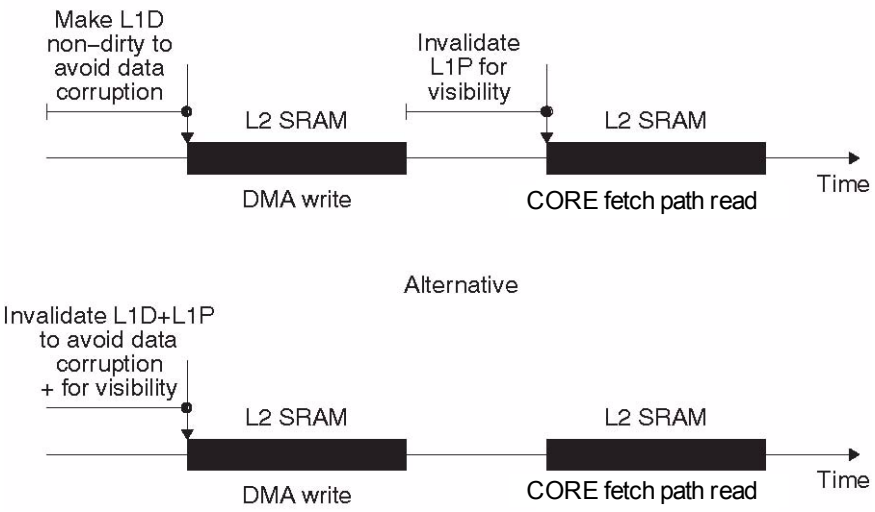


Figure B-5 L2 SRAM: DMA Write, CORE Read (Code)



B.1 Eliminating False Addresses

In the coherence matrices, it is assumed that each line only contains addresses that are meant to be operated on. Addresses that were not meant to be operated on are referred to as false addresses. If they existed then:

- The effect of coherence operations that were meant to make data visible to the CORE could be undone, but only if they were performed before the write access. The condition stated in the coherence matrix is that the “line must be invalid at the time of the first read/fetch access after the write”. However, if the CORE accessed false addresses after lines were already invalidated the line might be allocated again before the write instead of after as required.
- The effect of coherence operations that were meant to eliminate potential data corruption of newly written data by the DMA/other could be undone. The condition stated in the coherence matrix is that the “line must not be dirty at the time of the DMA/other write access”. However, if the CORE wrote to false addresses in cache after the line was already made clean or invalidated (through WB, INV, or WBINV), it might be made dirty again.
- The use of L1D INV or L2 INV would cause loss of data if these false addresses were recently written by the CORE but not yet written back to physical memory. The use of WBINV instead of INV would avoid this type of data corruption.

Since it is difficult to control CORE accesses to false addresses, it is strongly recommended that false addresses are eliminated. This is done by aligning the start address of a buffer in external memory at an L2 cache line size boundary and making its length a multiple of the L2 cache line size (128 bytes). For L2 SRAM addresses, the L1D cache line size (64 bytes) may be used instead, and for the CORE data path versus fetch path coherence case, the L1P cache line size (32 bytes) may be used (regardless of L2 SRAM or external memory addresses).

Index

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265