TMS320C6000 DSP Cache User's Guide

Literature Number: SPRU656A
May 2003

TEXAS INSTRUMENTS

Preface

Read This First

About This Manual

This document explains the fundamentals of memory caches and describes how to efficiently utilize the TMS320C6000™ DSP two-level cache-based memory architecture. It shows how to maintain coherence with external memory, how to use DMA to reduce memory latencies, and how to optimize your code to improve cache efficiency. Project collateral discussed in this user guide can be downloaded from http://www.ti.com/lit/zip/SPRU656.

How to Use this Manual

Novice users unfamiliar with memory caches should read this document starting with Chapter 1, which lays the fundamentals for all later chapters. More advanced users interested in issues related to cache coherence may directly skip to Chapter 2. Users who are familiar with the C6000™ DSP memory architecture and do not experience coherence problems, but want to learn how to optimize their application code for higher performance by reducing cache overhead cycles may directly go to Chapter 3.

Notational Conventions

This document uses the following conventions.

- Hexadecimal numbers are shown with the suffix h. For example, the following number is 40 hexadecimal (decimal 64): 40h.
- Program listings and examples are shown in a `special typeface` similar to a typewriter's.

  Here is a sample program listing:

```
#include <csl.h>
#include <csl_cache.h>
CSL_init();
CACHE_enableCaching(CACHE_CE00);
CACHE_setL2Mode(CACHE_48KCACHE);
```

Related Documentation From Texas Instruments

The following documents describe the C6000™ devices and related support tools. Copies of these documents are available on the Internet at www.ti.com. *Tip:* Enter the literature number in the search box provided at www.ti.com.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the TMS320C6000™ CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6000 Peripherals Reference Guide (literature number SPRU190) describes the peripherals available on the TMS320C6000™ DSPs.

TMS320C6000 Technical Brief (literature number SPRU197) gives an introduction to the TMS320C62x™ and TMS320C67x™ DSPs, development tools, and third-party support.

TMS320C64x Technical Overview (SPRU395) gives an introduction to the TMS320C64x™ DSP and discusses the application areas that are enhanced by the TMS320C64x VelociTI™.

TMS320C6000 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000™ DSPs and includes application program examples.

TMS320C6000 Code Composer Studio Tutorial (literature number SPRU301) introduces the Code Composer Studio™ integrated development environment and software tools.

Code Composer Studio Application Programming Interface Reference Guide (literature number SPRU321) describes the Code Composer Studio™ application programming interface (API), which allows you to program custom plug-ins for Code Composer.

TMS320C6x Peripheral Support Library Programmer's Reference (literature number SPRU273) describes the contents of the TMS320C6000™ peripheral support library of functions and macros. It lists functions and macros both by header file and alphabetically, provides a complete description of each, and gives code examples to show how they are used.

TMS320C6000 Chip Support Library API Reference Guide (literature number SPRU401) describes a set of application programming interfaces (APIs) used to configure and control the on-chip peripherals.

Trademarks

Code Composer Studio, C6000, C62x, C64x, C67x, TMS320C6000, TMS320C62x, TMS320C64x, TMS320C67x, and VelociTI are trademarks of Texas Instruments.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

This chapter discusses the basic operation of memory caches and describes the operation of the TMS320C6000™ DSP two-level cache architecture.

Topic **Page**

1.1 Purpose of This User's Guide

This user's guide describes how the cache-based memory system of the C621x, C671x, and C64x™ DSPs 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 cache (L1P) and a dedicated data cache (L1D) on the first level. Accesses by the CPU to the these first level caches can complete without CPU pipeline stalls. If the data requested by the CPU is not contained in cache, it is fetched from the next lower memory level, L2 or external memory. A detailed technical description of the C621x/C671x memory architecture is given in *TMS320C621x/671x DSP Two-Level Internal Memory Reference Guide* (SPRU609) and the C64x memory architecture is given in *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610).

The following topics are covered in this user's guide:

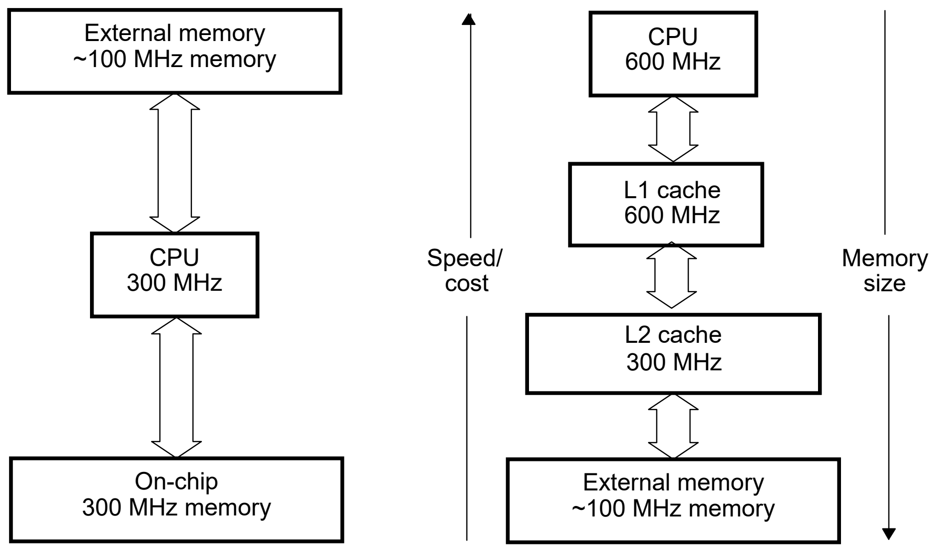
- ▭ The necessity of caches in high-performance DSPs
- ▭ General introduction into cache-based architectures
- ▭ Configuring and using the cache on the C621x, C671x, and C64x devices
- ▭ Maintaining coherence of the cache with external memory
- ▭ Linking code and data for increased cache efficiency
- ▭ Code-optimization techniques for increased cache efficiency

1.2 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 CPU 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 CPU 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 CPU stalls. If the CPU clock was now increased to 600 MHz, the internal memory could only service CPU accesses every two CPU cycles and the CPU 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 CPU 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 CPU speed would be far too expensive.

Figure 1–1. Flat Versus Hierarchical Memory Architecture



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 CPU that can be accessed without stalls. The next lower memory levels are increasingly larger but also slower the further away they are from the CPU. 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.

1.3 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:

$y[0] = h[0] \times x[0] + h[1] \times x[1] + ... + h[5] \times x[5]$

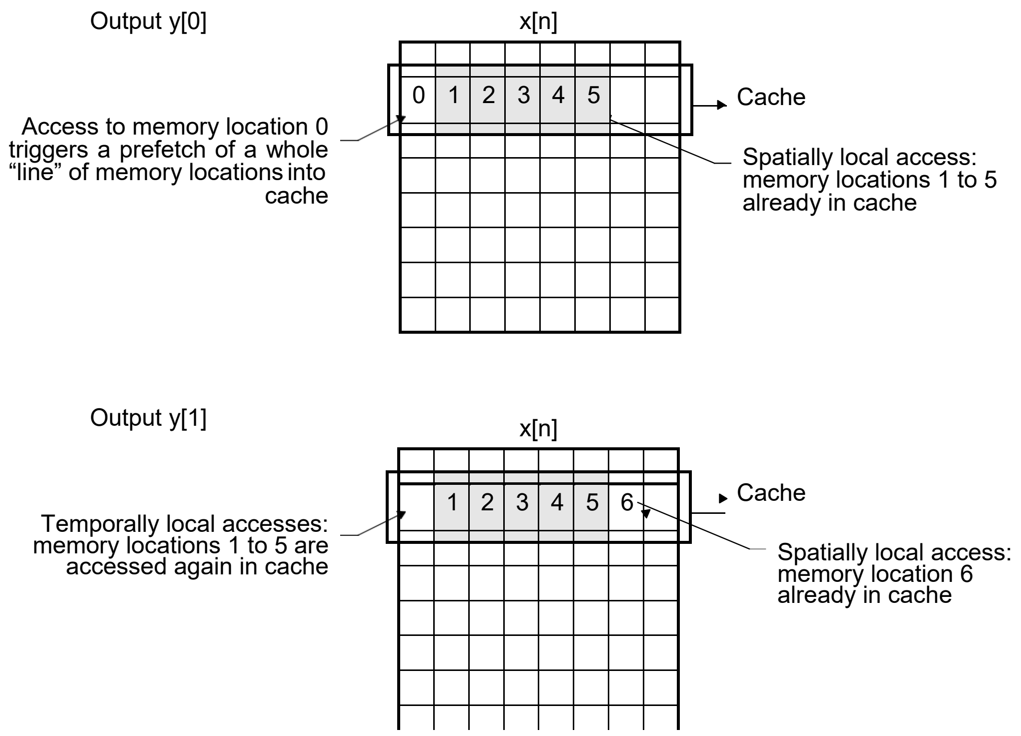
$y[1] = h[0] \times x[1] + h[1] \times x[2] + ... + h[5] \times x[6]$

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 CPU 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 CPU speed from the high-level cache memory.

Figure 1–2. Access Pattern of a 6-Tap FIR Filter


Output y[0]
x[n]
0
1
2
3
4
5
Cache
Access to memory location 0 triggers a prefetch of a whole “line” of memory locations into cache
Spatially local access: memory locations 1 to 5 already in cache
Output y[1]
x[n]
1
2
3
4
5
6
Cache
Temporally local accesses: memory locations 1 to 5 are accessed again in cache
Spatially local access: memory location 6 already in cache

1.4 Cache Memory Architecture Overview

The C6000™ DSP memory architecture consists of a two-level internal cache-based memory architecture plus external memory. Level 1 cache is split into program (L1P) and data (L1D) cache. On C64x devices (Figure 1–3), each L1 cache is 16 Kbytes; on C621x/C671x devices (Figure 1–4), each L1 cache is 4 Kbytes. All caches and data paths shown in Figure 1–3 and Figure 1–4 are automatically managed by the cache controller. Level 1 cache is accessed by the CPU without stalls. Level 2 memory is configurable and can be split into L2 SRAM (addressable on-chip memory) and L2 cache for caching external memory locations. On a C6416 DSP for instance, the size of L2 is 1 Mbyte; on a C621x/C671x device, the size of L2 is 64 Kbytes. External memory can be several Mbytes large. The access time depends on the memory technology used but is typically around 100 to 133 MHz.

Figure 1–3. C64x Cache Memory Architecture

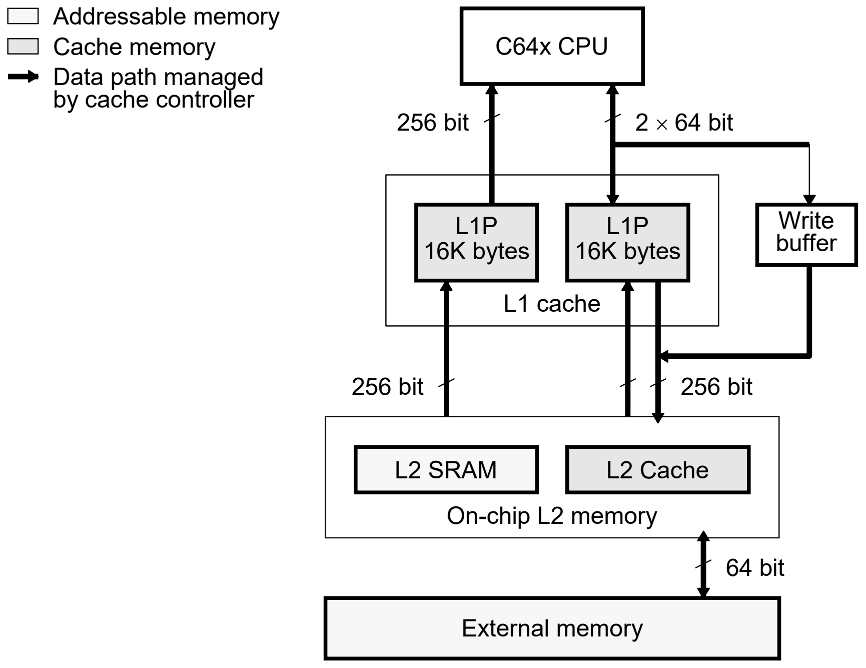
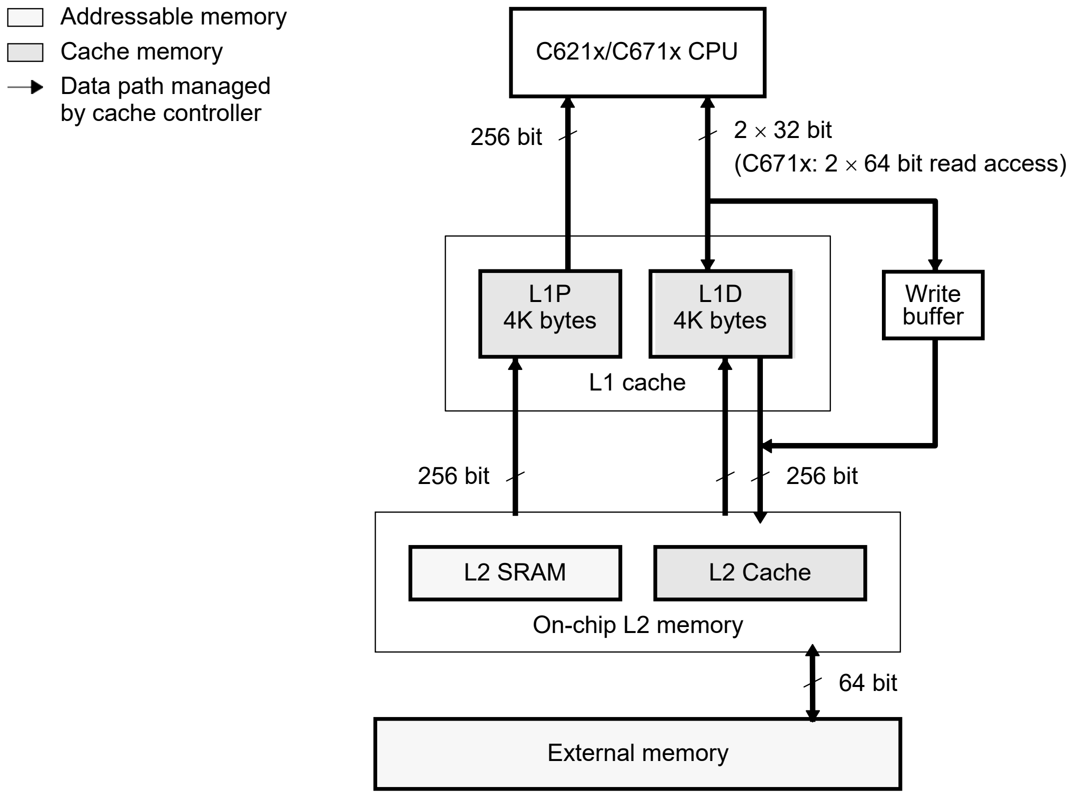


Figure 1–4. C621x/C671x Cache Memory Architecture



1.5 Cache Basics

This section explains the different types of cache architectures and how they work. Generally, you can distinguish between direct-mapped caches and set-associative caches. The caches described use the C64x L1P (direct-mapped) and L1D (set-associative) as examples; however, the concept is the same for C621x/C671x DSPs and is similar for all cache-based computer architectures. This section focuses on the behavior of the cache system. Performance considerations, including various stall conditions and associated stall cycles are discussed in section 3.1, *Cache Performance Characteristics*.

1.5.1 Direct-Mapped Caches

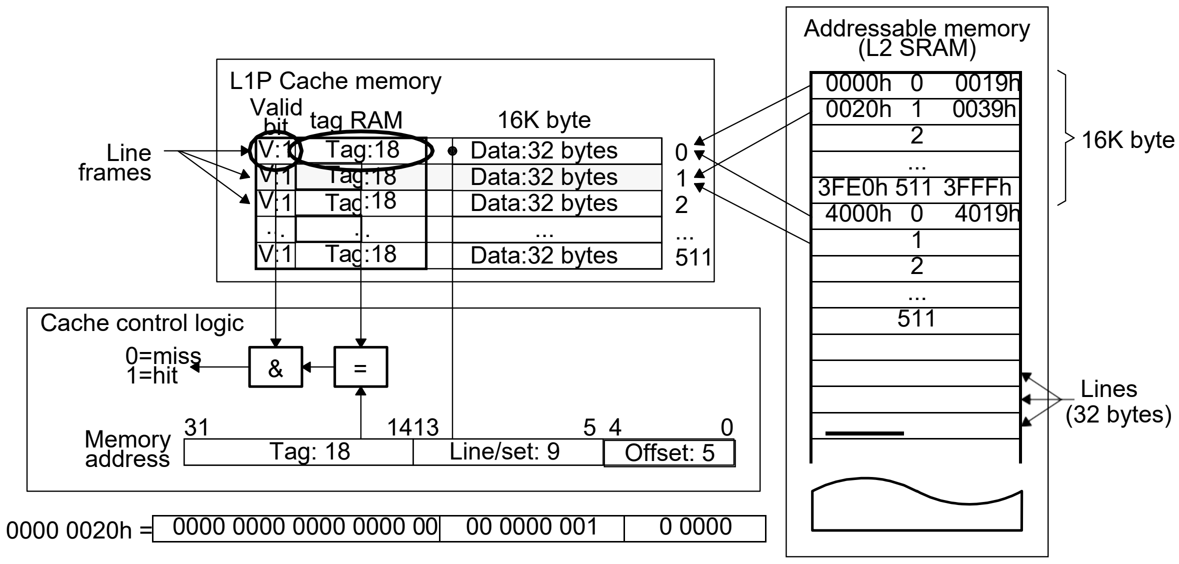
The C64x program cache (L1P) is used as an example to explain how a direct-mapped cache functions. Whenever the CPU accesses instructions in memory, the instructions are brought into L1P. The characteristics of the C64x and the C621x/C671x L1P caches are summarized in Table 1–1.

Table 1–1. L1P Characteristics

Characteristic	C621x/C671x DSP	C64x DSP
Organization	Direct-mapped	Direct-mapped
Protocol	Read Allocate	Read Allocate
CPU access time	1 cycle	1 cycle
Capacity	4 Kbytes	16 Kbytes
Line size	64 bytes	32 bytes
Single miss stall	5 cycles	8 cycles
Miss pipelining	No	Yes

Figure 1–5 shows the architecture of the C64x L1P that consists of the cache memory and the cache control logic. Additionally, addressable memory (L2 SRAM or external memory) is shown. The cache memory is 16 Kbytes large 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 1–5, 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 1–5. C64x L1P Architecture



1.5.1.1 Read Misses

Consider a CPU 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 CPU makes a request to read address 0020h, the cache controller splits up the address into three portions as shown in Figure 1–6.

Figure 1–6. Memory Address from Cache Controller

31 14	13 5	4 0
Tag	Set	Offset

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 set to 1 to indicate that the set now contains valid data. The fetched data is also forwarded to the CPU, 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.

1.5.1.2 Read Hits

The cache controller splits up the address into the three portions, as shown in Figure 1–6. 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 CPU. The access is completed.

1.5.2 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.

1.5.2.1 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 1.5.3.

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.

1.5.2.2 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.

1.5.3 Set-Associative Caches

Set-associative caches have multiple *cache ways* to reduce the probability of conflict misses. The C64x L1D is a 2-way set-associative cache with 16 Kbytes capacity (8 Kbytes per way) and 64-byte lines. The C621x/C671x L1D is also a 2-way set-associative cache, but with 4 Kbytes capacity (2 Kbytes per way) and 32-byte lines. The characteristics of the L1D caches are summarized in Table 1–2.

Table 1–2. L1D Characteristics

Characteristic	C621x/C671x DSP	C64x DSP
Organization	2-way set-associative	2-way set-associative
Protocol	Read Allocate, Write-back	Read Allocate, Write-back
CPU access time	1 cycle	1 cycle
Capacity	4 Kbytes	16 Kbytes
Line size	32 bytes	64 bytes
Single read miss stall (L2 SRAM)	4 cycles	6 cycles
Single read miss stall (L2 Cache)	4 cycles	8 cycles
Miss pipelining	No	Yes
Multiple consecutive misses (L2 SRAM)	4 cycles	4 + 2 × M cycles
Multiple consecutive misses (L2 Cache)	4 cycles	6 + 2 × M cycles
Write miss	Passed through 4 × 32-bit write buffer. Only stalls when full.	Passed through 4 × 64-bit write buffer. Only stalls when full.

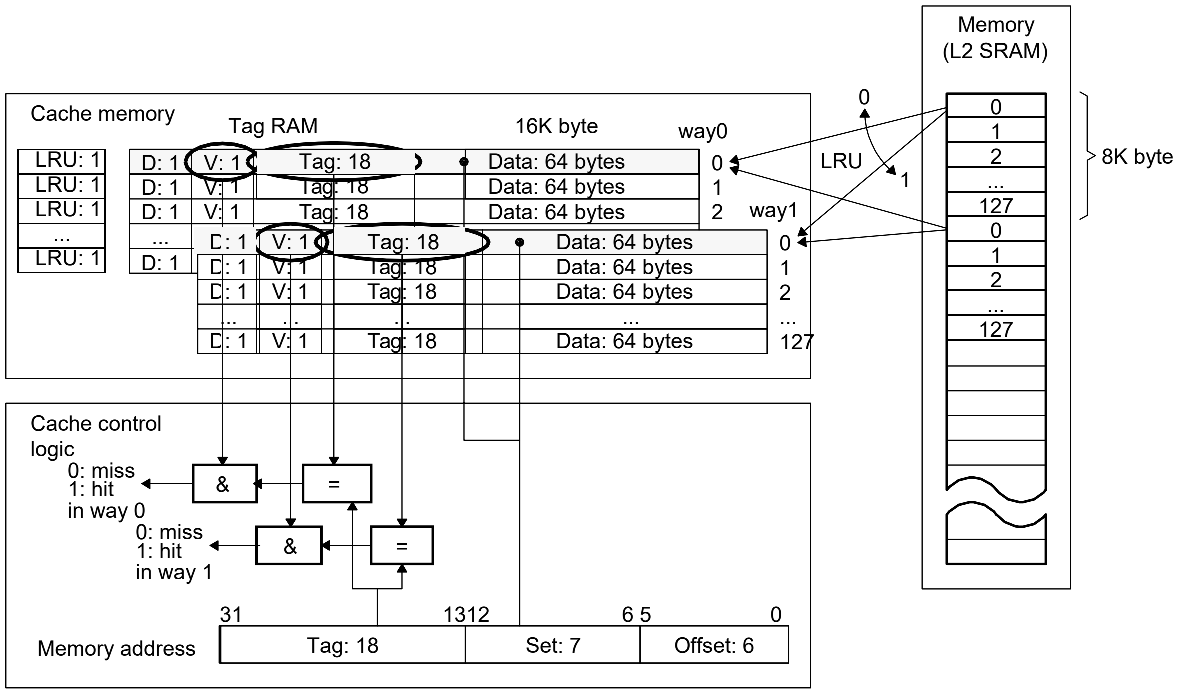
The difference to a direct-mapped cache is that in a 2-way cache each set consists of two line frames, one line frame in way 0 and another line frame in way 1. A line in memory still maps to one set, but now can be stored in either of the two line frames. In this sense, a direct-mapped cache can also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

The set-associative cache architecture is explained by examining how misses and hits are handled for the C64x L1D cache. Its architecture is shown in Figure 1–7. Hits and misses are determined the same 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.

1.5.3.1 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.

Figure 1–7. C64x L1D Architecture



1.5.3.2 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.

1.5.3.3 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 and Figure 1–4). The write buffer consists of 4 entries. On C621x/C671x devices, each entry is 32-bits wide; on C64x devices, each entry is 64-bits wide.

1.5.3.4 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 CPU write access is written back to memory at a later time. To write back modified data, you have to know which line was written to by the CPU. For this purpose, every cache line has a *dirty bit* (D) associated with it. Initially, the dirty bit is zero. As soon as the CPU 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 CPU, and the LRU bit indicates that way 0 is to be replaced on the next miss. If the CPU 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.

1.5.4 Level 2 (L2) Cache

Until now, it was assumed that there is one level of cache memory between the CPU 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 for 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.

The C6000 L2 memory space can 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 1–3.

Table 1–3. L2 Cache Characteristics

Characteristic	C621x/C671x DSP	C64x DSP
Organization	1-, 2-, 3-, or 4-way set-associative (depending on selected cache capacity)	4-way set-associative
Protocol	Read and write allocate Write-back	Read and write allocate Write-back
Capacity	16/32/48/64 Kbytes	32/64/128/256 Kbytes
Line size	128 bytes	128 bytes
Replacement strategy	Least recently used	Least recently used

1.5.4.1 Read Misses and Hits

Consider a CPU read request to a cacheable external memory address that misses in L1 (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 2.2, *Coherence*). The portion of the 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 CPU. Again, if the new line replaces a dirty line in L1, its contents are first written back to L2 cache.

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

Some external memory addresses may be configured as noncacheable. In this case, the requested data is simply forwarded from external memory to the CPU without being stored in any of the caches (see section 2.1, *Configuring L2 Cache*, for more information).

1.5.4.2 Write Misses and Hits

If a CPU 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 CPU 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 CPU write data.

Note that some external memory addresses may be configured as noncacheable. In this case, the data is directly updated in external memory without being stored in cache.

Chapter 2

Using Cache

This chapter explains how to enable and configure cache for C621x/C671x and C64x devices. It also describes the cache coherence protocol employed by the cache controller and provides examples for common application scenarios.

Since in a cache-based memory system, multiple copies of the same memory location containing different data may exist simultaneously, a protocol must be followed that ensures that a device different from the CPU (for example, a peripheral) does not access an out-of-date copy of a memory location. This protocol is referred to as a cache coherence protocol.

Important! Whenever external memory caching is enabled and the EDMA is used to transfer to/from external memory, it is your responsibility to maintain cache coherence. Failing to do so almost certainly results in incorrect functioning of the application.

Topic **Page**

2.1 Configuring L2 Cache

After a reset, 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) commands. Additionally, in the linker command file the memory to be used as L2 SRAM has to be specified. Since L2 cache cannot be used for code or data placement by the linker, all sections must be linked into L2 SRAM or external memory.

Further, you can control whether external memory addresses are cacheable or noncacheable. Each external memory address space of 16 Mbytes is controlled by one MAR bit (0: noncacheable, 1:cacheable). The MAR registers are documented in *TMS320C621x/671x DSP Two-Level Internal Memory Reference Guide* (SPRU609) and *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610). For instance, to enable caching on a C6211/6711 device for the external memory range from 8000 0000h to 80FF FFFFh, the CSL function `CACHE_enableCaching(CACHE_CE00)` can be used. This sets register MAR0 to 1. For C64x devices, the function `CACHE_enableCaching(CACHE_EMIFA_CE00)` would be called to set register MAR128 to 1. After the MAR bit is set for an external memory space, new addresses accessed by the CPU will be cached in L2 cache or, if L2 is all SRAM, in L1. After a reset, caching for external memory address space is disabled.

See *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401) for more information on how to use the cache CSL functions.

2.1.1 C6211/C6711 Cache Configurations

On C6211/C6711 devices, the linker command file for an example configuration of 16K SRAM and 48 Kbytes 3-way cache is shown in Figure 2–1.

The required CSL command sequence to enable caching of external memory locations and to enable L2 cache is shown in Figure 2–2.

In Figure 2–2, the first command initializes the CSL. Then caching of the external memory space CE00h, which corresponds to the first 16 Mbytes in external memory, is enabled by setting the appropriate MAR bit. Finally, L2 cache size is set to 48 Kbytes.

Figure 2–1. C6211/C6711 Cache Configuration Linker Command File

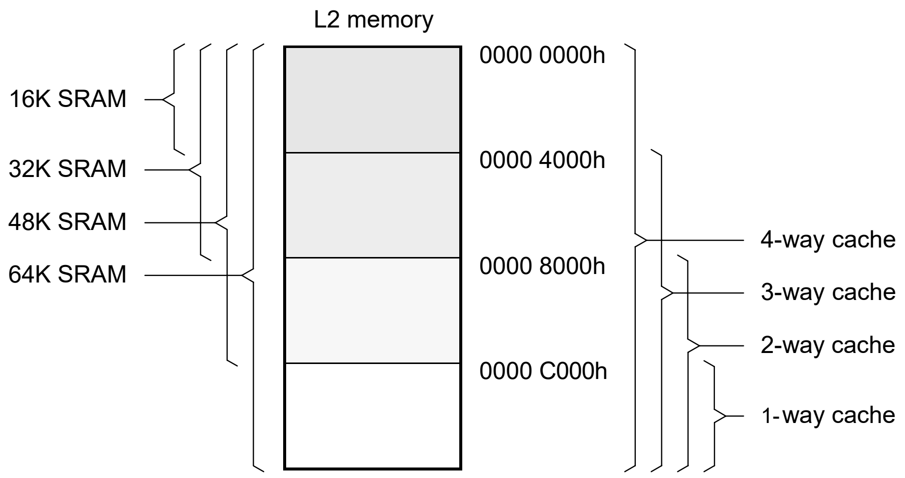
```
MEMORY
{
       L2SRAM:       origin = 00000000h   length = 00004000h
       CE0:          origin = 80000000h   length = 01000000h
}

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
}
```

Figure 2–2. C6211/C6711 CSL Command Sequence to Enable Caching

```
#include     <csl.h>
#include     <csl_cache.h>
...
CSL_init();
CACHE_enableCaching(CACHE_CE00);
CACHE_setL2Mode(CACHE_48KCACHE);
```

Figure 2–3 shows all possible cache configurations for C6211/C6711 devices. Slightly different configurations may exist for other C621x/C671x devices. See your device-specific datasheet.

Figure 2–3. C6211/C6711 L2 Memory Configurations



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 Figure 2–1 and Figure 2–2. Therefore, the corresponding `CACHE_setL2Mode()` calls and `MEMORY` definitions are following.

Important! Do not define memory that is to be used as cache under the `MEMORY` directive. This memory is not valid for the linker to place code or data in.

- 64K SRAM, 0K cache:

```
CACHE_setL2Mode(CACHE_0KCACHE);
    L2SRAM:  origin = 00000000h   length = 00010000h
```

- 48K SRAM, 16K cache:

```
CACHE_setL2Mode(CACHE_16KCACHE);
    L2SRAM:  origin = 00000000h   length = 0000C000h
```

- 32K SRAM, 2-way 32K cache:

```
CACHE_setL2Mode(CACHE_32KCACHE);
    L2SRAM:  origin = 00000000h   length = 00008000h
```

- 16K SRAM, 3-way 48K cache:

```
CACHE_setL2Mode(CACHE_48KCACHE);
    L2SRAM:  origin = 00000000h   length = 00004000h
```

- 0K SRAM, 4-way 64K cache: No code/data can be linked into L2 SRAM.

```
CACHE_setL2Mode(CACHE_64KCACHE);
```

2.1.2 C64x Cache Configurations

The following description is for C64x devices with 1024 Kbytes of L2 memory. For C64x devices with different L2 sizes, see the device-specific datasheet. On C64x devices, the linker command file for a configuration of 992K SRAM and 32 Kbytes 4-way cache is shown in Figure 2–4.

The required CSL command sequence to enable caching of external memory locations and to enable L2 cache is shown in Figure 2–5.

In Figure 2–5, the first command initializes the CSL. Then caching of the external memory space CE00h, which corresponds to the first 16 MBytes in external memory connected to EMIFA, is enabled by setting the appropriate MAR bit. Finally, L2 cache size is set to 32 Kbytes.

Figure 2–4. C64x Cache Configuration Linker Command File

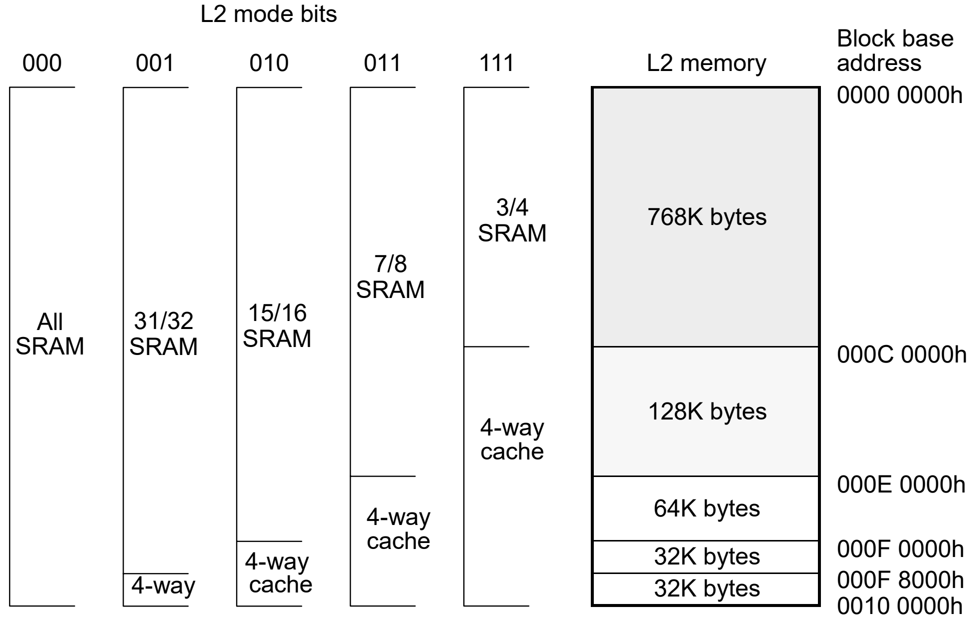
```
MEMORY
{
       L2SRAM:       origin = 00000000h   length = 000F8000h
       CE0:          origin = 80000000h   length = 01000000h
}

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
}
```

Figure 2–5. C64x CSL Command Sequence to Enable Caching

```
#include      <csl.h>
#include      <csl_cache.h>
...
CSL_init();
CACHE_enableCaching(CACHE_EMIFA_CE00);
CACHE_setL2Mode(CACHE_32KCACHE);
```

Figure 2–6 shows all possible cache configurations for C64x devices with 1024 Kbytes of L2 memory. Slightly different configurations may exist for other C64x devices. See your device-specific datasheet.

Figure 2–6. C64x L2 Memory Configurations



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 Figure 2–4 and Figure 2–5. Therefore, the corresponding `CACHE_setL2Mode()` calls and `MEMORY` definitions are following.

Important! Do not define memory that is to be used as cache under the `MEMORY` directive. This memory is not valid for the linker to place code or data in.

- 1024K SRAM, 0K cache:

```
CACHE_setL2Mode(CACHE_0KCACHE);
    L2SRAM:  origin = 00000000h   length = 00100000h
```

- 992K SRAM, 32K cache:

```
CACHE_setL2Mode(CACHE_32KCACHE);
    L2SRAM:  origin = 00000000h   length = 000F8000h
```

- 960K SRAM, 64K cache:

```
CACHE_setL2Mode(CACHE_64KCACHE);
    L2SRAM:  origin = 00000000h   length = 000F0000h
```

- 896K SRAM, 128K cache:

```
CACHE_setL2Mode(CACHE_128KCACHE);
    L2SRAM:  origin = 00000000h   length = 000E0000h
```

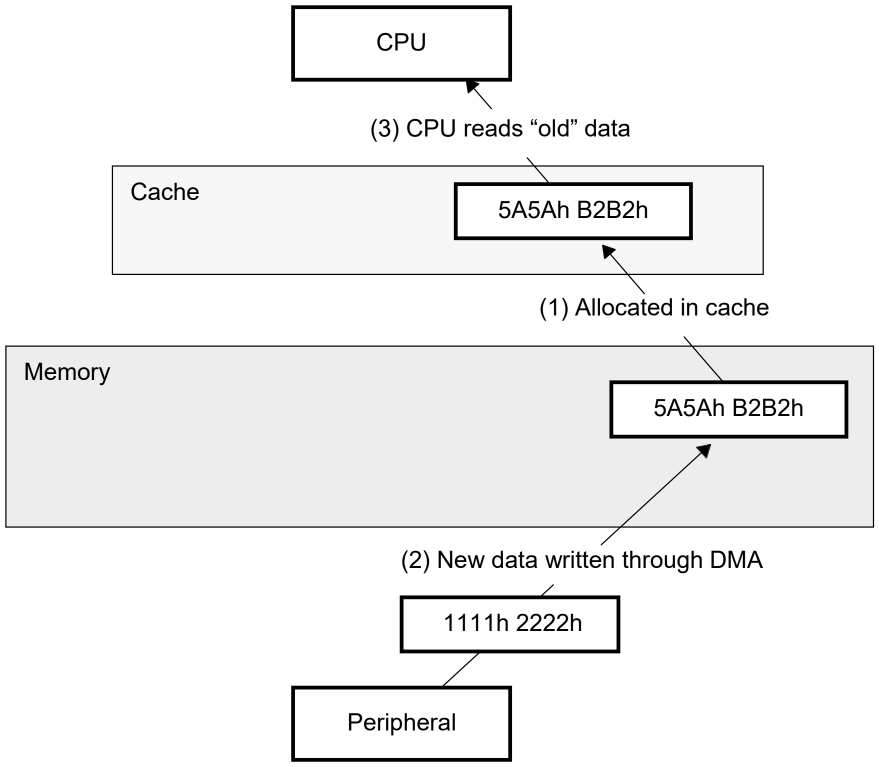
- 768K SRAM, 256K cache:

```
CACHE_setL2Mode(CACHE_256KCACHE);
    L2SRAM:  origin = 00000000h   length = 000C0000h
```

2.2 Coherence

Generally if multiple devices, such as the CPU or peripherals, share the same cacheable memory region, cache and memory can become incoherent. Consider the system shown in Figure 2–7. Suppose the CPU accesses a memory location that gets subsequently allocated in cache (1). Later, a peripheral is writing data to this same location that is meant to be read and processed by the CPU (2). However, since this memory location is kept in cache, the memory access hits in cache and the CPU reads the old data instead of the new data (3). A similar problem occurs if the CPU 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*.

Figure 2–7. Cache Coherence Problem



Coherence needs to be addressed if the following is true:

- Multiple devices (CPUs, peripherals, DMA controllers) 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.

C621x/C671x and C64x DSPs *automatically* maintain cache coherence for accesses by the CPU and EDMA to L2 SRAM 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 location is first updated with modified data from L1D before its value is returned to the DMA. On a DMA write, the corresponding L1D cache line is invalidated.

For accesses by the CPU and EDMA to external memory, it is your responsibility to maintain cache coherence. For this purpose, the cache controller offers various commands that allow it to manually keep the cache coherent for accesses by the CPU and EDMA to external memory.

A less frequently occurring case is coherency between L1D and L1P, which also is your responsibility. However, measures need only be taken in very special cases such as self-modifying code and boot loading.

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

For a more formal definition of cache coherence and its relation to memory consistency models, see *TMS320C621x/671x DSP Two-Level Internal Memory Reference Guide* (SPRU609) and *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610).

2.2.1 Snoop Commands

Before describing cache coherence mechanisms and operations, it is beneficial to first understand the basic underlying protocols that are used for all coherence operations. The cache controller supports snoop commands to maintain coherence between the L1 caches and L2 SRAM/cache. 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, typically, a writeback-invalidate, a writeback, or an invalidate only of the corresponding cache line is triggered. The C6000 cache controller supports the following snoop commands:

- L1D Snoop Command (C64x devices only):
  - Writes back a line from L1D to L2 SRAM/cache
  - Used for DMA reads of L2 SRAM
- L1D Snoop-Invalidate Command:
  - Writes back a line from L1D to L2 SRAM/cache and invalidates it in L1D
  - Used for DMA writes to L2 SRAM and user-controlled cache operations
- L1P Invalidate Command:
  - Invalidates a line in L1P
  - Used for DMA write of L2 SRAM and user-controlled cache operations

Note that the DMA is not allowed to access addresses that map to L2 cache.

2.2.2 Cache Coherence Protocol for DMA Accesses to L2 SRAM

To illustrate the coherence protocols, assume a peripheral is writing data through the DMA to an input buffer located in L2 SRAM. Then the CPU 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 2–8 and is:

1) The peripheral requests a write access to a line in L2 SRAM that maps to set 1 in L1D.

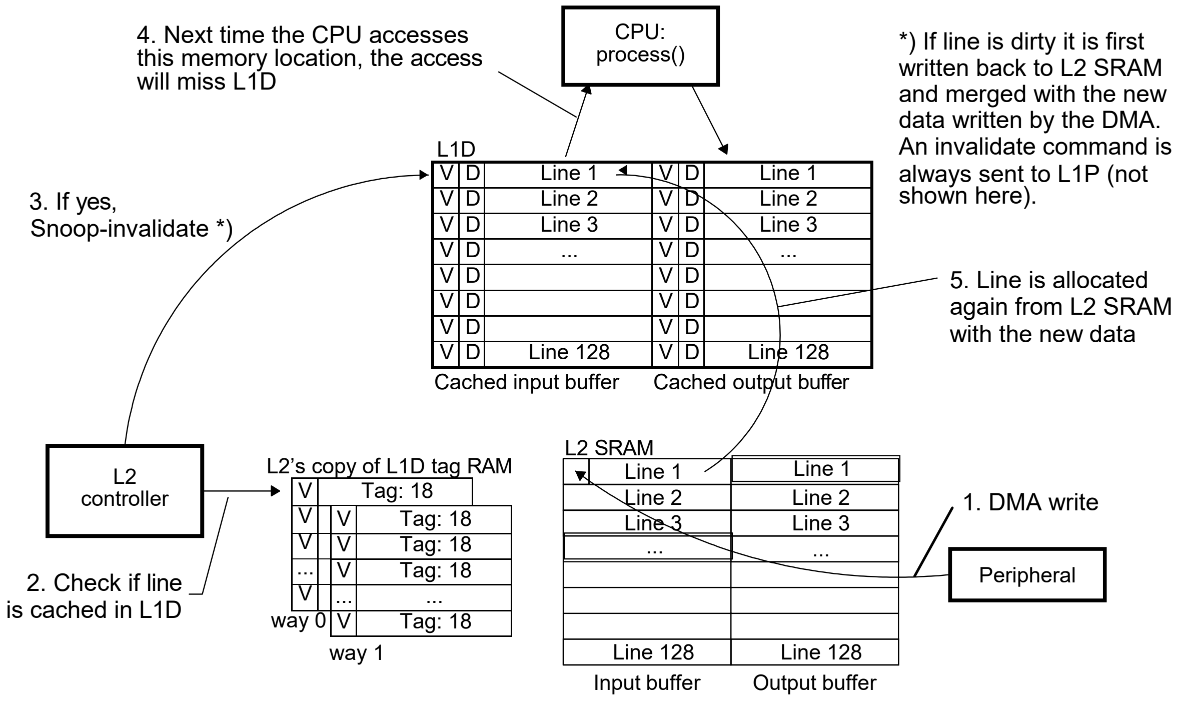
2) The L2 cache controller checks its local copy of the L1D tag RAM and determines if the line that was just requested is cached in L1D (by checking the valid bit and the tag). If the line is not cached in L1D, no further action needs to be taken and the data is written to memory.

3) If the line is cached in L1D, the L2 controller sends a *snoop-invalidate* command to L1D. This clears the valid bit of the corresponding line, invalidates the line. If the line is dirty, it is written back to L2 SRAM. Then the new data from the peripheral is written to L2 SRAM.

4) The next time the CPU accesses this memory location, the access will miss in L1D and the line containing the new data written by the peripheral is allocated in L1D and read by the CPU. If the line had not been invalidated, the CPU would have read the “old” value that was cached in L1D.

Note that the L2 controller sends an *invalidate* command to L1P. This is necessary in case program code is to be written to L2 SRAM. No data needs to be written back in this case since data in L1P is never modified.

Figure 2–8. DMA Write to L2 SRAM


4. Next time the CPU accesses this memory location, the access will miss L1D
CPU: process()
*) If line is dirty it is first written back to L2 SRAM and merged with the new data written by the DMA. An invalidate command is always sent to L1P (not shown here).
L1D
3. If yes, Snoop-invalidate *)
V D Line 1 Line 2 Line 3 ... Line 128
Cached input buffer
Cached output buffer
5. Line is allocated again from L2 SRAM with the new data
L2 controller
L2's copy of L1D tag RAM
V Tag: 18
way 0
way 1
2. Check if line is cached in L1D
L2 SRAM
Input buffer
Output buffer
1. DMA write
Peripheral

The procedure for a DMA read is shown in Figure 2–9 and is:

1) The CPU 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.

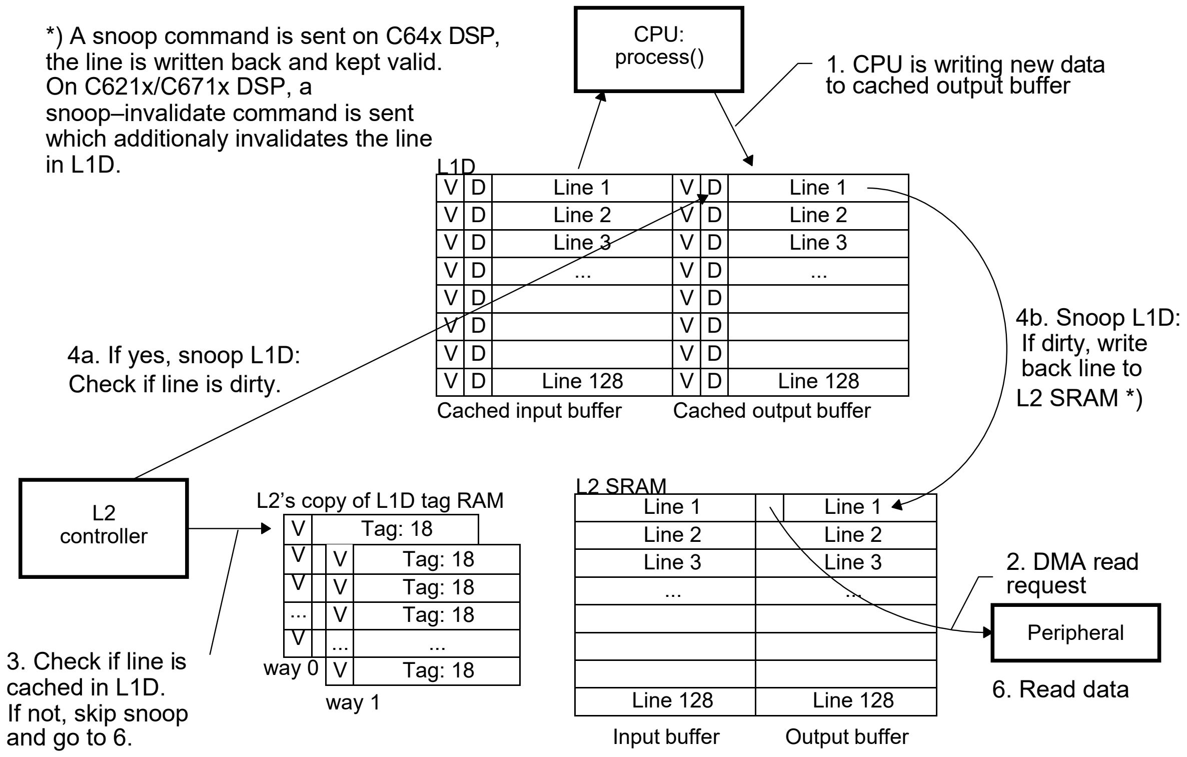
2) When the peripheral issues a DMA read request to the memory location in L2 SRAM, the controller checks to determine if the line that contains the memory location requested is cached in L1D. In this example, we already assumed that it is cached. However, if it was not cached, no further action would be taken and the peripheral would complete the read access.

3) If the line is cached, the L2 controller sends a *snoop* 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 causes the dirty line to be written back to L2 SRAM. This is the case in this example, since we assumed that the CPU has written to the output buffer.

5) Finally, the read access completes the peripheral reading of the “new” data written by the CPU.

Figure 2–9. DMA Read of L2 SRAM


*) A snoop command is sent on C64x DSP, the line is written back and kept valid. On C621x/C671x DSP, a snoop–invalidate command is sent which additionaly invalidates the line in L1D.
CPU: process()
1. CPU is writing new data to cached output buffer
L1D
V D Line 1
V D Line 2
V D Line 3
...
V D Line 128
Cached input buffer
Cached output buffer
4b. Snoop L1D: If dirty, write back line to L2 SRAM *)
4a. If yes, snoop L1D: Check if line is dirty.
L2 controller
L2’s copy of L1D tag RAM
V Tag: 18
way 0
way 1
L2 SRAM
Line 1
Line 2
Line 3
Line 128
Input buffer
Output buffer
2. DMA read request
Peripheral
6. Read data
3. Check if line is cached in L1D. If not, skip snoop and go to 6.

2.2.2.1 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 2–10. The idea is that while the CPU is processing data from one set of buffers (for example, InBuffA and OutBuff A), the peripherals are writing/ reading data using the other set of buffers (InBuffB and OutBuff B) such that the DMA data transfer may occur in parallel with CPU processing.

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

1) Buffer InBuffB is being filled, while the CPU is processing data in InBuffA. The lines of InBuffA are allocated in L1D. Data is processed by the CPU and is written through the write buffer to OutBuffA (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 CPU is processing InBuffB. For InBuffA, the L2 cache controller automatically takes care of invalidating the corresponding lines in L1D through snoop-invalidates. The CPU will then allocate the line again from L2 SRAM with the new data, rather than reading the cached line containing the old data. 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 Figure 2–11 shows how a double buffering scheme could be realized. A complete example Code Composer Studio™ (CCS) project is available in the accompanying zip archive (L2_DOUBLE_BUF).

Figure 2–10. Double Buffering in L2 SRAM

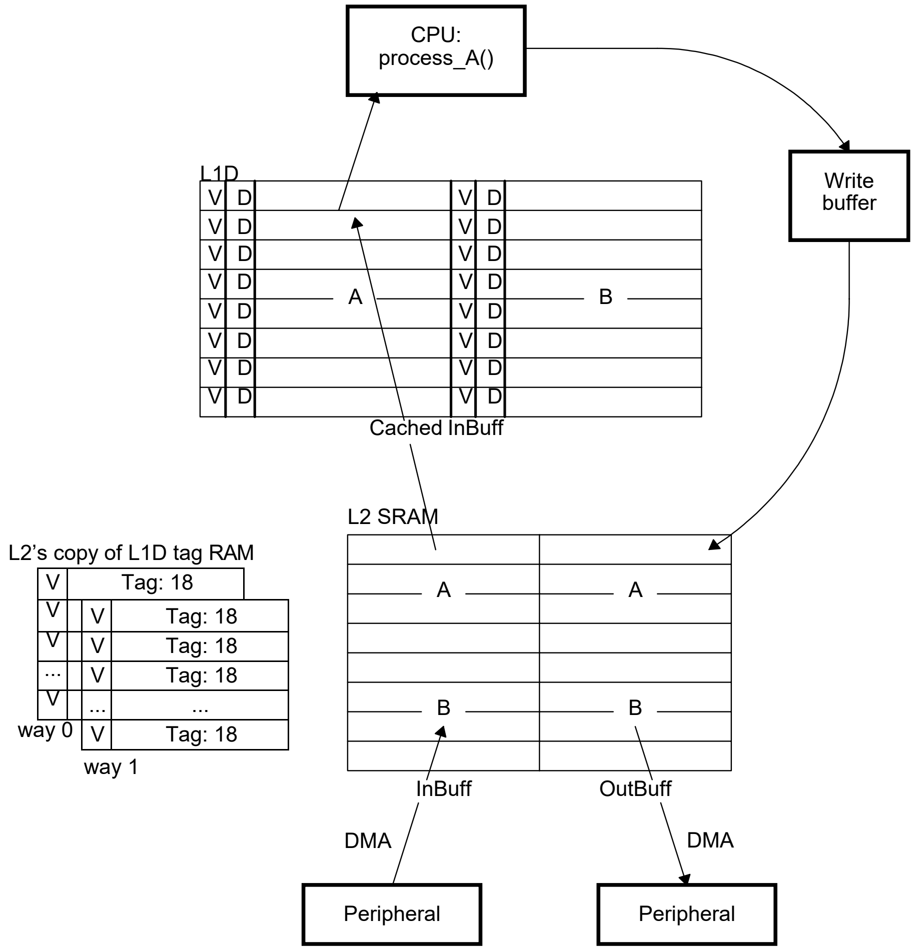


Figure 2–11. L2SRAM DMA Double Buffering Code Example

```
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);

}
```

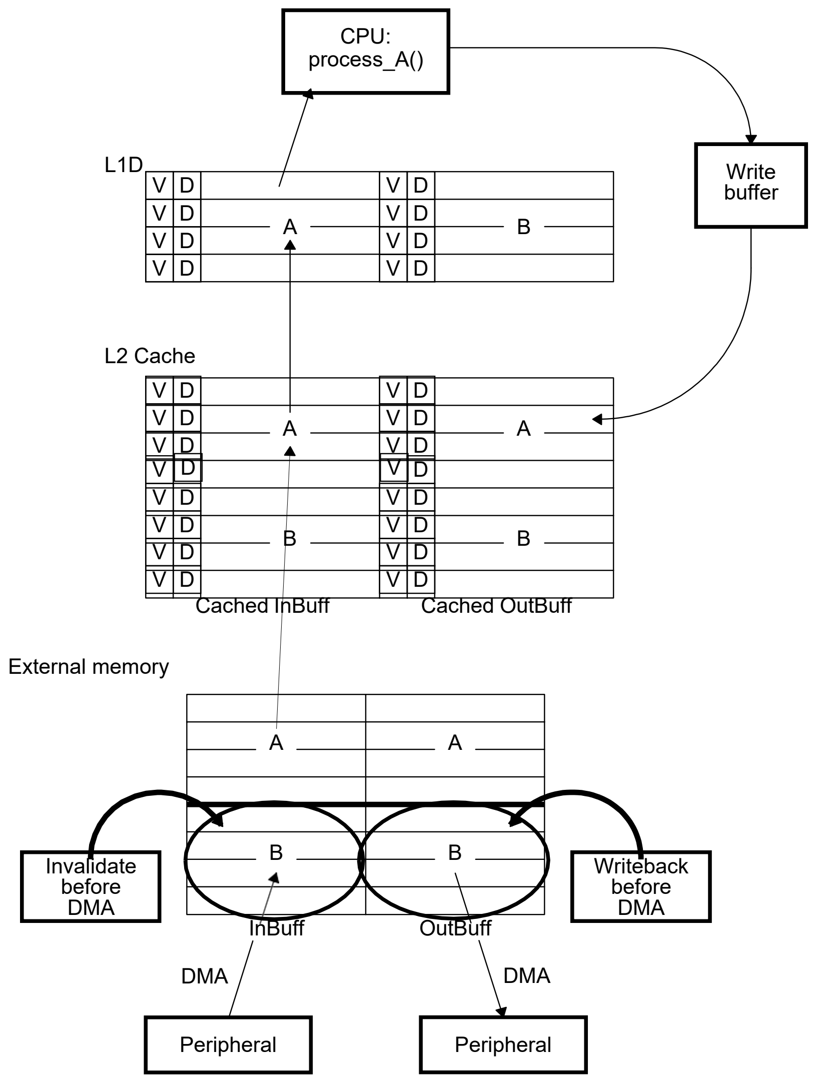
2.2.2.2 Maintaining Coherence Between External Memory and Cache

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 your responsibility to maintain coherence. Again, the CPU 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 2–12, assume that transfers already have occurred, both InBuff and OutBuff are cached in L2 cache, and InBuff is cached in L1D. Further assume that the CPU 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.

We already know from the previous example what the L2 cache controller did to keep L2 SRAM coherent with L1D. We have to do exactly the same here to ensure that external memory is kept coherent with L2 cache, and L2 cache with L1D.

Figure 2–12. Double Buffering in External Memory



To maintain coherence, you have to imitate for external memory what the cache controller does for L2 SRAM accesses. Whenever data is written to an input buffer, the cache controller would *invalidate* the corresponding line in the cache. Similarly, here 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 CPU will reallocate these lines from external memory next time the input buffer is read.

The chip support library (CSL) provides a set of routines that allow the required cache coherence operations to be initiated. Before the DMA write transfer starts, a w*riteback-invalidate* (or alternatively an *invalidate* on C64x devices) has to be completed. The start address of the buffer in external memory and the number of bytes need to be specified:

- C621x/C671x devices, which only support writeback-invalidate, or C64x devices:

  ```
  CACHE_wbInvL2(InBuffB, BUFSIZE, CACHE_WAIT);
  ```

- For C64x devices, an invalidate-only operation is also supported that completes faster:

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

Similarly, before OutBuffB is transferred to the peripheral, the data first has to be *written back* from L1D and L2 cache to external memory. This is done by issuing a w*riteback* operation (C621x/C671x and C64x devices):

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

Again, this is necessary since the CPU writes data only to the cached copies of the memory locations of OutBuffB that still may reside in L1D and L2 cache.

Additionally, a wait flag is specified. 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 CPU to continue execution of the program while the coherence operation is performed in the background. However, care must be taken that the CPU is not accessing addresses that the cache controller is operating since this causes memory corruption. 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 can be found in section 2.2.3.

The pseudo-code in Figure 2–13 shows exactly in which order the cache coherence calls and the DMA transfers should occur. A complete example CCS project is available in the accompanying zip archive.

Figure 2–13. External Memory DMA Double Buffering Code Example

```
for (i=0; i<(DATASIZE/BUFSIZE)-2; i+=2)
{
/* ----------------------------------------------------- */
/*   InBuffA -> OutBuffA Processing                       */
/* ----------------------------------------------------- */
      CACHE_wbInvL2(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_wbInvL2(InBuffA, BUFSIZE, CACHE_WAIT);
      <DMA_transfer(peripheral, InBuffA, BUFSIZE)>

      CACHE_wbL2(OutBuffA, BUFSIZE, CACHE_WAIT);
      <DMA_transfer(OutBuffA, peripheral, BUFSIZE)>

      process(InBuffB, OutBuffB, BUFSIZE);

}
```

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 2.2.3. These requirements can be achieved as:

```
#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, a CSL macro can be used that automatically rounds array sizes up to the next multiple of a cache line size. The macro is defined as:

```
#define CACHE_ROUND_TO_LINESIZE(cache,elcnt,elsize)        \
        ((CACHE_#cache#_LINESIZE *                         \
          ((elcnt)*(elsize)/CACHE_#cache#_LINESIZE) + 1) / \
           (elsize))
```

The array definitions above would then look as follows:

```
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)];
```

2.2.3 Usage Guidelines for L2 Cache Coherence Operations

Table 2–1 shows an overview of available L2 cache coherence operations for C621x/C671x and C64x devices. Note that these operations have no effect if L2 cache is disabled; in this case, refer to section 2.2.4. Table 2–1 has to be interpreted as follows. First, the cache controller checks if an external memory address within the specified range is cached in L2 cache. If yes, it then issues a snoop-invalidate command to L1D (and invalidate command to L1P, if required) to make L2 and L1 coherent. Then the appropriate operation is performed on L2 cache.

Table 2–1. L2 Cache Coherence Operations

Scope	Coherence Operation	CSL Command	Operation on L2 Cache	L1D Snoop Commands	L1P Snoop Commands
Range	Invalidate L2 (C64x devices only)	CACHE_invL2 (external memory start address, byte count, wait)	All lines within range invalidated	L1D snoop-invalidate (any returned dirty data is discarded)	L1P invalidate
	Writeback L2	CACHE_wbL2 (external memory start address, byte count, wait)	Dirty lines within range written back All lines within range kept valid	L1D snoop-invalidate	None
	Writeback–Invalidate L2	CACHE_wbInvL2 (external memory start address, byte count, wait)	Dirty lines within range written back All lines within range invalidated	L1D snoop-invalidate	L1P invalidate
All L2 Cache	Writeback All L2	CACHE_wbAllL2(wait)	All dirty lines in L2 written back All lines in L2 kept valid	L1D snoop-invalidate	None
	Writeback-Invalidate All L2	CACHE_wbInvAllL2 (wait)	All dirty lines in L2 written back All lines in L2 invalidated	L1D snoop-invalidate	L1P invalidate

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, arrays in external memory that are accessed by both CPU and EDMA mustbe:

- A multiple of cache lines large
- Aligned at a cache line boundary

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 $4 \times 65\ 535$, that is, one L2 cache operation can operate on at most 256 Kbytes. 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 cache coherence operations are only required if the CPU and DMA share a cacheable region of external memory, that is, if the CPU reads data written by the DMA and vice versa.

The safest rule is 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. DMA Scenarios With Coherence Operation Required

Scenario		Coherence Operation Required
1)	DMA reads data written by the CPU	Writeback *before* DMA starts
2)	DMA writes data that is to be read by the CPU	Invalidate or Writeback-Invalidate *before* DMA starts
3)	DMA modifies data written by the CPU that data is to be read back by the CPU	Writeback-Invalidate *before* DMA starts

In scenario 3, the DMA may modify data that was written by the CPU and that data is then read back by the CPU. This is the case if the CPU 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 CPU needs to be committed to external memory and the buffer has to be invalidated.

2.2.4 Maintaining Coherence Between External Memory and L1 Caches

In case L2 cache is disabled (configured as all SRAM) and external memory caching is enabled, L1D and L1P cache coherence operations have to be issued to maintain coherence between external memory and L1 caches. Although using external memory without L2 cache is not a recommended configuration because overall application performance may suffer, it is explained here for comprehensiveness. The same guidelines outlined for L2 cache in section 2.2.3 should be followed; however, instead of the L2 cache operations, the L1D/L1P cache operations in Table 2–3 are used.

Table 2–3. L1D/L1P Cache Coherence Operations

Scope	Coherence Operation	CSL Command	L1D Snoop Commands	L1P Snoop Commands
Range	Invalidate L1D (C64x devices only)	CACHE_invL1d (external memory start address, byte count, wait)	L1D Snoop-Invalidate (any returned dirty data is discarded)	None
	Writeback-Invalidate L1D	CACHE_wbInvL1d (external memory start address, byte count, wait)	L1D Snoop-Invalidate	C621x/C671x devices: L1P Invalidate C64x devices:None
	Invalidate L1P	CACHE_invL1p (external memory start address, byte count, wait)	None	L1P Invalidate
All	Invalidate L1P	CACHE_invL1pAll()	None	L1P Invalidate

2.3 Switching Cache Configuration During Run-Time

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

2.3.1 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 set 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.

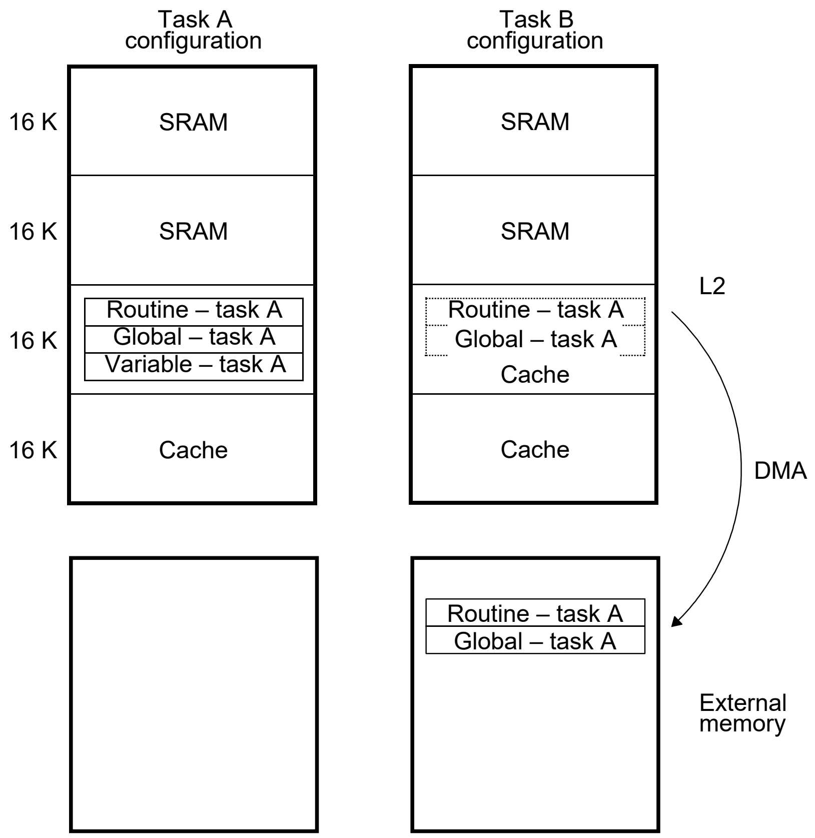
If all addresses in the respective external memory address space are made noncacheable, the addresses need to be written back and invalidated first (see sections 2.2.3 and 2.2.4 for a description of user-initiated cache control operations). If external memory addresses are only kept in L1D, in the case of L2 all SRAM mode, an L1D Writeback-Invalidate operation has to be performed.

2.3.2 Changing L2 Cache Size During Run-Time

Changing the size of L2 cache during run time may be beneficial for some applications. Consider the following example for a C621x/C671x device (same concept applies to C64x devices). An application has two tasks: A and B. Task A benefits from 48 Kbytes of code and data being allocated in L2 SRAM, while task B would benefit from having 32 Kbytes of L2 cache. Assume the memory configuration as shown in Figure 2–14. The third 16 Kbyte 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 16 Kbyte segment) by copying the code and the global variables to another memory region in external memory using a DMA. Then, all memory addresses in the 16 Kbyte segment that reside in L1D or L1P have to be writeback-invalidated since those addresses no longer exist after switching the segment to cache mode. Then, the cache mode can be switched. Finally, 8 cycles of NOP need to be executed. The writeback-invalidate, mode switch operation, and execution of 8 NOPs is all performed by the function `CACHE_setL2Mode()`.

Figure 2–14. Changing L2 Cache Size During Run-Time (C6211/C6711 Devices)



To switch back to task A configuration, L2 cache line frames located in the 16 Kbyte 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, an L2 Writeback-Invalidate All has to be performed. This also snoop-invalidates L1D and invalidates L1P. Then the cache mode can be switched and code and global variables copied back to their original location.

The exact procedures are given in Table 2–4. The same procedure applies to C621x/C671x and C64x devices. Note that for C64x devices, an additional L2 Writeback-Invalidate All for switching to a mode with more L2 cache is required (because the organization of the cache ways for C64x L2 cache is always 4-ways, regardless of size), but this has been integrated into the function `CACHE_setL2Mode()`.

Table 2–4. Procedure for Switching Between L2 Mode

Switch To	Perform
More L2 Cache (Less L2 SRAM)	1) DMA needed code/data out of L2 SRAM addresses to be converted to cache. Note: L1D Snoop is triggered by DMA that will invalidate L1D.
	2) Wait for completion of step 1.
	3) Increase L2 Cache size: `CACHE_setL2Mode()`
Less L2 Cache (More L2 SRAM)	1) Decrease L2 Cache size: `CACHE_setL2Mode()`
	2) DMA back any code/data needed.
	3) Wait for completion of step 2.

Note that switching from an all SRAM mode to a mode with L2 cache after having accessed cacheable external memory is not recommended since this could lead to incoherence problems. This is because external addresses may still be left in L1D without being contained in L2 cache. To ensure that all external addresses are writeback-invalidated, you would have to perform an L1D Writeback-Invalidate All operation, which is not available. In the case that you know exactly which addresses reside in L1D, a range writeback-invalidate operation could be performed.

Figure 2–15 shows a C code example of how to realize the above L2 mode switching example. The corresponding linker command file is shown in Figure 2–16 (page 2-29).

Figure 2–15. L2 Mode Switching C Code Example (C621x/C671x Devices)

```
/* ---------------------------------------------------------------------- */
/*  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;

       /* --------------------------------------------------------------- */
       /*  Initialize CSL, set L2 mode and open DAT                       */
       /* --------------------------------------------------------------- */
       CSL_init();

       CACHE_enableCaching(CACHE_CE00);

       CACHE_setL2Mode(CACHE_16KCACHE);

       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);
```

Figure 2–15. L2 Mode Switching C Code Example (C621x/C671x Devices) (Continued)

```
/* ------------------------------------------------------------- */
/*  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_setL2Mode(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 16K 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_setL2Mode(CACHE_16KCACHE);
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);
```

Figure 2–15. L2 Mode Switching C Code Example (C621x/C671x Devices) (Continued)

```
       /* -------------------------------------------------------------- */
       /*  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;
}
```

Figure 2–16. Linker Command File for L2 Mode Switching C Code Example

```
MEMORY
{
 L2_12: o = 00000000h  l = 00008000h /*1st and 2nd 16K segments: always  SRAM    */
 L2_3:  o = 00008000h  l = 00004000h /*3rd 16K segment:Task A–SRAM,Task B–Cache */
 L2_4:  o = 0000C000h  l = 00004000h /*4th 16K segment: always Cache            */
 CE0:   o = 80000000h  l = 01000000h /*external memory                          */
}

SECTIONS
{
       .cinit              >      L2_12
       .text               >      L2_12
       .stack              >      L2_12
       .bss                >      L2_12
       .const              >      L2_12
       .data               >      L2_12
       .far                >      L2_12
       .switch             >      L2_12
       .sysmem             >      L2_12
       .tables             >      L2_12
       .cio                >      L2 12

       .sram_state_A       >      L2_3
       .sram_process_A     >      L2_3
       .sram_local_var_A   >      L2_3

       .external           >      CE0
}
```

2.4 Self-Modifying Code and L1P Coherence

No coherence is maintained between L1D and L1P. That means if the CPU wants to modify program code, the writes may only update L1D, L2 SRAM, or L2 cache, but not L1P. For the CPU 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 run-time, the following procedure has to be followed:

1) Disable interrupts.
2) Perform CPU writes (STW) to modify code.
3) Perform coherence operations:
   a) If C621x/C671x devices:
      i) Perform an L1D Writeback-Invalidate operation (includes L1P Invalidate).
      ii) Wait for operation to complete.
   b) If C64x devices:
      i) Perform an L1D Writeback-Invalidate operation.
      ii) Perform an L1P Invalidate operation.
      iii) 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.

2.5 Summary of Coherence Properties

The memory system of C621x/C671x and C64x devices has the following coherence properties. The first two properties are concerned with data accesses and the third property is concerned with write accesses to program code.

1) L1D and L2 SRAM present a coherent memory system to the CPU and peripherals. The CPU views L2 SRAM through L1D; the peripherals view L2 SRAM directly. Coherence is maintained automatically by the cache controller.

2) External memory is not part of the coherent memory system. The CPU views data in cacheable external memory through L1D and L2 cache. Since the cache controller does not maintain coherence, the CPU may never be able to see the same data as a peripheral at the same address in external memory. If and when required, coherence among CPU and peripherals can be achieved by manually issuing cache coherence commands to the cache controller. Note that in the case of only one single device accessing an external memory region, coherence is assured (that is, a read by a device that follows a write by *the same* device to the same location returns the written value). Only if external memory is shared among multiple devices, coherence between those devices is not maintained.

3) Write accesses by the DMA controller or a peripheral to program code in L2 SRAM *are* kept coherent with L1P reads. This allows for transferring program code from external memory to L2 SRAM through DMA without the need to manually invalidate L1P. However, write accesses by the CPU to program code in L2 SRAM or L1D are *not* kept coherent with L1P reads. Also, write accesses by the CPU or peripherals to program code in external memory are *not* kept coherent with L1P reads. Due to this incoherence, self-modifying code is not supported automatically by the hardware. If required, self-modifying code may be software controlled by manually issuing cache coherence commands.

From a typical software application point of view, properties 1 and 2 are the most important. Note that only coherence of data is addressed. Maintaining coherence for program code, which is addressed by property 3, is not usually required except in the special case of self-modifying code.

In a typical application, L2 memory is split into L2 cache and L2 SRAM portions. The first property mostly concerns those parts of an application that do not utilize external memory and keep data/code in L2 SRAM. For these parts, no action by the programmer is required regarding coherence. The second property is relevant for those parts of the application that do utilize external memory for data or code. If L2 is configured as all SRAM, external memory is cached in L1 only. However, this configuration is not recommended since a considerable performance penalty can be expected.

2.6 Old and New CSL Cache API's

The CSL cache coherence APIs have been renamed to better reflect the actual operation. If you are encouraged to switch to the new APIs, the old APIs still work, but are no longer updated. Also, some new C64x cache operations were not supported by the old CSL version. Table 2–5 and Table 2–6 show the correct function calls for the new API that should be used to replace the old API. Note that the new API expects byte counts whereas the old API expected word counts. Also, the old CSL routines waited for completion of the coherence operations. To achieve the same behavior with the new routines, CACHE_WAIT has to be used.

Table 2–5. CSL API's for L2 Cache Operations

Scope	Coherence Operation	Old CSL Command	New CSL Command
Range	L2 Invalidate (C64x devices only)	N/A	CACHE_invL2(start address, byte count, CACHE_WAIT)
	L2 Writeback	CACHE_flush(CACHE_L2, start address, word count)	CACHE_wbL2(start address, byte count, CACHE_WAIT)
	L2 Writeback-Invalidate	CACHE_clean(CACHE_L2, start address, word count)	CACHE_wbInvL2(start address, byte count, CACHE_WAIT)
All L2 Cache	L2 Writeback All	CACHE_flush(CACHE_L2ALL, [ignored], [ignored])	CACHE_wbAllL2(CACHE_WAIT)
	L2 Writeback-Invalidate All	CACHE_clean(CACHE_L2ALL, [ignored], [ignored])	CACHE_wbInvAllL2(CACHE_WAIT)

Table 2–6. CSL API's for L1 Cache Operations

Scope	Coherence Operation	Old CSL Command	New CSL Command
Range	L1D Invalidate (C64x devices only)	N/A	CACHE_invL1d(start address, byte count, CACHE_WAIT)
	L1D Writeback-Invalidate	CACHE_flush(CACHE_L1D, start address, word count)	CACHE_wbInvL1d(start address, byte count, CACHE_WAIT)
	L1P Invalidate	CACHE_invalidate(CACHE_L1P, start address, word count)	CACHE_invL1p(start address, byte count, CACHE_WAIT)
All	L1P Invalidate All	CACHE_invalidate(CACHE_L1PALL, [ignored], [ignored])	CACHE_invAllL1p()

Chapter 3

Optimizing for Cache Performance

This chapter discusses 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 CPU speed. However, this scenario becomes more and more unrealistic the higher the CPU 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 CPU 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.

Topic **Page**

3.1 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 CPU 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 CPU. 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 discussed in Chapter 1, *Introduction*. You also have to understand what conditions CPU stalls occur and the cycle penalty associated with these stalls.

For this purpose, the next two sections present an overview of the C621x/C671x and C64x cache architecture, respectively, detailing all important cache characteristics, cache stall conditions and associated stall cycles. These sections provide a useful reference for optimizing code for cache performance.

3.1.1 C621x/C671x Stall Conditions

The most common stall conditions on C621x/C671x devices are:

- L1D Dual-Port Memory Access Conflict: A parallel write/write or write/read hit to the same 32-bit word causes a 1 cycle stall.
- L1D Read Miss: 4 cycles per miss to perform a line allocation from L2 SRAM or L2 cache. 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 ratio of the CPU and EMIF clock rate and EDMA latencies.
  - L2 Access Conflict: L2 can only service one request at a time. If L1P and L1D access L2 simultaneously for line allocation, L1P is given priority.
  - L2 Bank Conflict: Since an L2 access requires 2 cycles to complete, accesses to the same bank on consecutive cycles cause a stall. For instance, one additional stall cycle is caused if an L1D line allocation access occurs on the next cycle after a write buffer, L1P line allocation or EDMA access to the same bank. Note, simultaneous accesses to the same bank fall under L2 Access Conflict case.

  - 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 Conflicts, L2 Bank Conflicts, and L2 Write Misses (the write buffer data misses L2).
  - L1D Victim Buffer Writeback: If the victim buffer contains data and a read miss occurs, its 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 Conflicts and L2 Bank Conflicts.

Parallel misses will be overlapped, provided none of the above stall lengthening condition occurs between the misses and the two parallel misses are not to the same set. Two parallel misses take 7 cycles to complete, 3.5 cycles per miss.

- L1D Write Buffer Full: The 4 × 32-bit buffer drains at 2 cycles per entry. If an L1D write miss occurs and the write buffer is full, the CPU is stalled until one entry is available. Write buffer draining can be lengthened by:
  - L2 Cache Write Miss: The line has to be fetched from external memory first (L2 cache is write allocate). The number of stall cycles depend on the ratio of CPU and EMIF clock rate and EDMA latencies.
  - L2 Access Conflict: L2 can only service one request at a time. L1P has priority over L1D requests. If a simultaneous access by the write buffer and L1P to L2 memory occurs, L1P is given priority.
  - L2 Bank Conflict: Since an L2 access requires 2 cycles to complete, accesses to the same bank on consecutive cycles cause a stall. For instance, one additional stall cycle is caused if a write buffer access occurs on the next cycle after another write buffer, an L1D line allocation, an L1P line allocation, or an EDMA access to the same bank. Note, simultaneous accesses to the same bank fall under L2 Access Conflict case.
- L1P Read Miss: 5 cycles per miss to perform a line allocation from L2 SRAM or L2 cache. Can be lengthened by:
  - L2 Cache Read Miss: The program data has to be fetched from external memory first. The number of stall cycles depends on the ratio of CPU and EMIF clock rate and EDMA latencies.

  - L2 Bank Conflict: Since an L2 access requires 2 cycles to complete, accesses to the same bank on consecutive cycles cause a stall. For instance, one additional stall cycle is caused if a L1P line allocate access occurs on the next cycle after a write buffer, L1D line allocate, or EDMA access to the same bank.

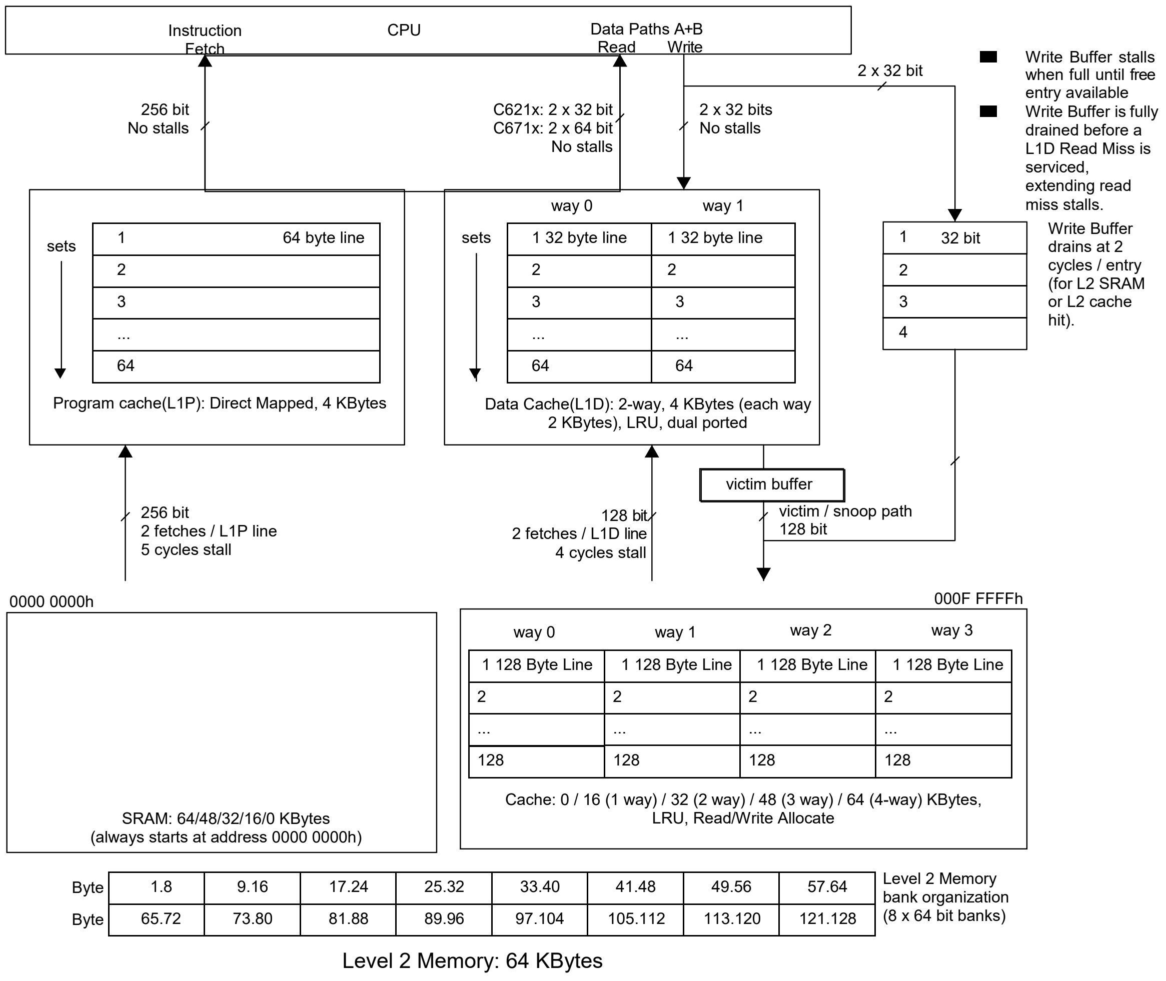
- Snoops: Stalls may occur due to snooping (used by EDMA accesses or cache coherence operations). Every time a snoop accesses the L1D tag RAM, the CPU is stalled (also if there is no simultaneous access request to L1D by the CPU). If the snoop hits L1D, CPU requests to L1D are stalled until the writeback-invalidate operation is complete. If the snoop and the CPU request occur simultaneously, the CPU request is given higher priority.

Figure 3–1 shows the C621x/C671x memory architectures detailing all important characteristics, stall conditions, and associated stall cycles.

3.1.2 C621x/C671x Pipelining of Read Misses

The C621x/C671x cache architecture pipelines read misses and allows parallel read miss stall cycles to be overlapped. While a single miss takes 4 stall cycles, two parallel read misses consume only 7 cycles, 3.5 cycles per miss. This mechanism is described in *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609).

Figure 3–1. C621x/C671x Cache Memory Architecture


CPU
Instruction Fetch
Data Paths A+B
Read Write
256 bit
No stalls
C621x: 2 x 32 bit
C671x: 2 x 64 bit
No stalls
2 x 32 bits
No stalls
2 x 32 bit
Write Buffer stalls when full until free entry available
Write Buffer is fully drained before a L1D Read Miss is serviced, extending read miss stalls.
Write Buffer drains at 2 cycles / entry (for L2 SRAM or L2 cache hit).
sets
1 64 byte line
2
3
...
64
Program cache(L1P): Direct Mapped, 4 KBytes
way 0
way 1
1 32 byte line
Data Cache(L1D): 2-way, 4 KBytes (each way 2 KBytes), LRU, dual ported
1 32 bit
4
victim buffer
victim / snoop path
128 bit
256 bit
2 fetches / L1P line
5 cycles stall
128 bit
2 fetches / L1D line
4 cycles stall
0000 0000h
000F FFFFh
SRAM: 64/48/32/16/0 KBytes
(always starts at address 0000 0000h)
way 2
way 3
1 128 Byte Line
128
Cache: 0 / 16 (1 way) / 32 (2 way) / 48 (3 way) / 64 (4-way) KBytes, LRU, Read/Write Allocate
Byte
1.8
9.16
17.24
25.32
33.40
41.48
49.56
57.64
65.72
73.80
81.88
89.96
97.104
105.112
113.120
121.128
Level 2 Memory bank organization (8 x 64 bit banks)
Level 2 Memory: 64 KBytes

3.1.3 C64x Stall Conditions

The most common stall conditions on C64x 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 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 *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610) for special case exceptions.

- L1D Read Miss: 6 cycles per miss to perform line allocation from L2 SRAM and 8 cycles per miss to perform line allocation from L2 cache. 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 ratio of CPU and EMIF clock rate and EDMA latencies.

  - L2 Access Conflict: L2 can service only one request at a time. L1P has priority over L1D requests. While some stall cycles of L1D and L1P read miss servicing may overlap, if a simultaneous access to L2 memory occurs, L1P is given priority.

  - L2 Bank Conflict: Since an L2 access requires 2 cycles to complete, accesses to the same bank on consecutive cycles cause a stall. For instance, one additional stall cycle is caused if an L1D line allocation access occurs on the next cycle after a write buffer, L1P line allocation, or EDMA access to the same bank. Note, simultaneous accesses to the same bank fall under L2 Access Conflict case.

  - 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 Conflicts, L2 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 Conflicts and L2 Bank Conflicts.

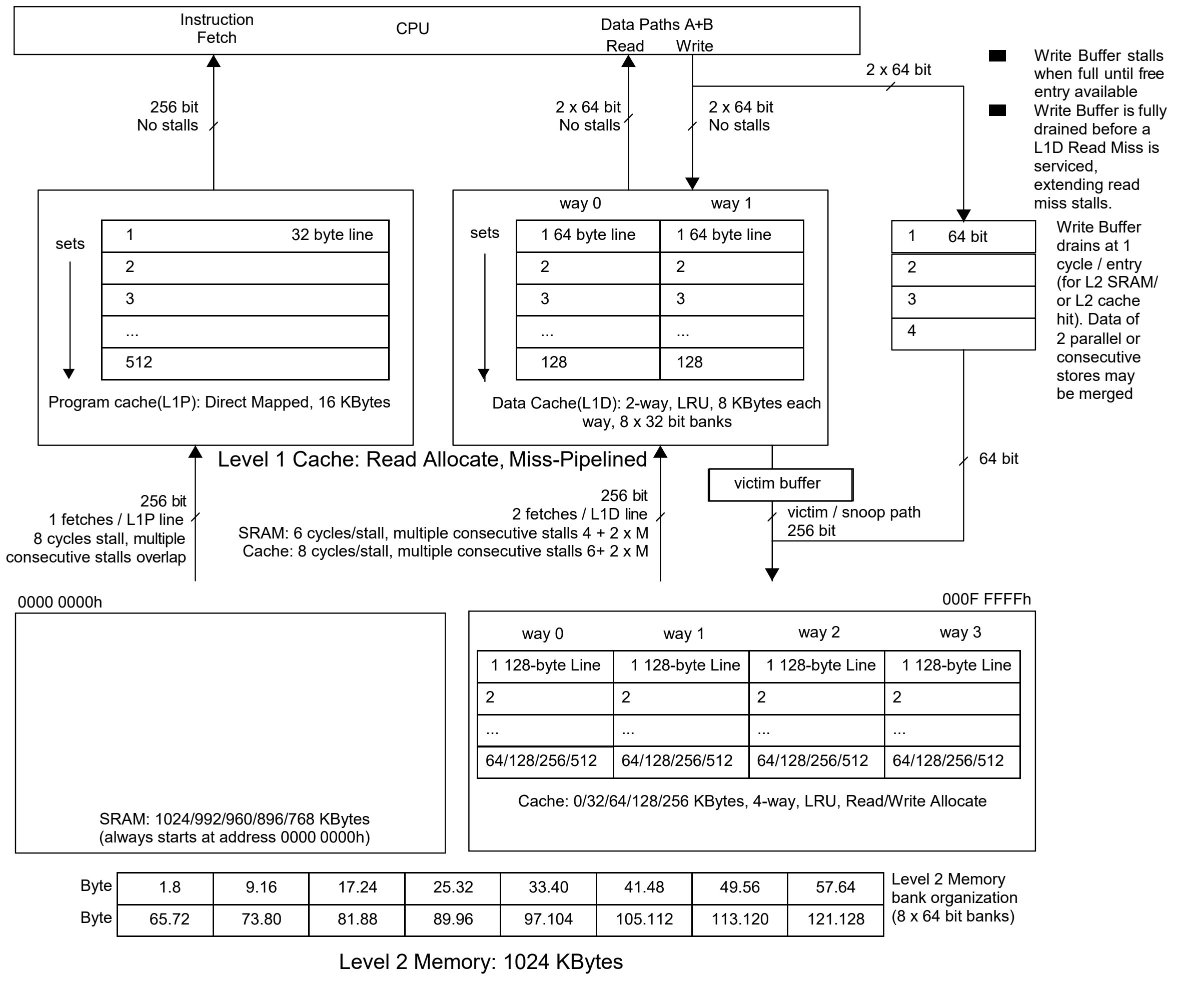
Consecutive and parallel misses will be overlapped, provided none of the above stall lengthening condition occurs and the two parallel/consecutive misses are not to the same set: $4 + 2 \times M$ cycles for M misses to L2 SRAM, and $6 + 2 \times M$ cycles for M misses to L2 cache.

- L1D Write Buffer Full: The 4×64-bit buffer drains at 1 cycle per entry. 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 Write Miss: The line has to be fetched from external memory first (L2 is write allocate). The number of stall cycles depends on the ratio of CPU and EMIF clock rate and EDMA latencies.
  - L2 Access Conflict: L2 can only service one request at a time. L1P has priority over L1D requests to L2. If a simultaneous access by the write buffer and L1P to L2 memory occurs, L1P is given priority.
  - L2 Bank Conflict: Since an L2 access requires 2 cycles to complete, accesses to the same bank on consecutive cycles cause a stall. For instance, one additional stall cycle is caused if a write buffer access occurs on the next cycle after another write buffer access, L1D line allocation, L1P line allocation, or EDMA access to the same bank. Note, simultaneous accesses to the same bank fall under L2 Access Conflict case.
- L1P Read Miss: 8 cycles per miss to perform a line allocation from L2 SRAM or L2 cache. Can be lengthened by:
  - L2 Cache Read Miss: The program data has to be fetched from external memory first. The number of stall cycles depends on the ratio of CPU and EMIF clock rate and EDMA latencies.
  - L2 Bank Conflict: Since an L2 access requires 2 cycles to complete, accesses to the same bank on consecutive cycles cause a stall. For instance, one additional stall cycle is caused if a L1P line allocation access occurs on the next cycle after a write buffer, L1D line allocation, or EDMA access to the same bank.

  Consecutive misses will be overlapped, provided none of the above stall lengthening condition occurs and the two consecutive misses are not to the same set.
- Snoops: Stalls may occur due to snooping (used by EDMA accesses or cache coherence operations). Every time a snoop accesses the L1D tag RAM, the CPU is stalled (even if there is no simultaneous access request to L1D tag RAM by the CPU). If the snoop hits L1D, CPU L1D requests are stalled until the writeback-invalidate operation is complete. If the snoop and the CPU L1D request occur simultaneously, the CPU request is given higher priority.

Figure 3–2 shows the C64x memory architectures detailing all important characteristics, stall conditions and associated stall cycles.

Figure 3–2. C64x Cache Memory Architecture



Instruction Fetch
CPU
Data Paths A+B
Read
Write
256 bit
No stalls
2 x 64 bit
No stalls
2 x 64 bit
No stalls
2 x 64 bit
Write Buffer stalls when full until free entry available
Write Buffer is fully drained before a L1D Read Miss is serviced, extending read miss stalls.
sets
1 32 byte line
2
3
...
512
Program cache(L1P): Direct Mapped, 16 KBytes
way 0
way 1
sets
1 64 byte line
1 64 byte line
128
Data Cache(L1D): 2-way, LRU, 8 KBytes each way, 8 x 32 bit banks
1 64 bit
2
3
4
Write Buffer drains at 1 cycle / entry (for L2 SRAM/ or L2 cache hit). Data of 2 parallel or consecutive stores may be merged
Level 1 Cache: Read Allocate, Miss-Pipelined
victim buffer
64 bit
256 bit
1 fetches / L1P line
8 cycles stall, multiple consecutive stalls overlap
256 bit
2 fetches / L1D line
SRAM: 6 cycles/stall, multiple consecutive stalls 4 + 2 x M
Cache: 8 cycles/stall, multiple consecutive stalls 6+ 2 x M
victim / snoop path
256 bit
0000 0000h
000F FFFFh
SRAM: 1024/992/960/896/768 KBytes
(always starts at address 0000 0000h)
way 0
way 1
way 2
way 3
1 128-byte Line
64/128/256/512
Cache: 0/32/64/128/256 KBytes, 4-way, LRU, Read/Write Allocate
Byte
1.8
9.16
17.24
25.32
33.40
41.48
49.56
57.64
Byte
65.72
73.80
81.88
89.96
97.104
105.112
113.120
121.128
Level 2 Memory bank organization (8 x 64 bit banks)
Level 2 Memory: 1024 KBytes

3.1.4 C64x Pipelining of Read Misses

The C64x cache architecture pipelines read misses and allows parallel and consecutive read miss stall cycles to be overlapped. While a single miss to L2 SRAM causes a 6 cycle stall, multiple parallel and consecutive misses consume only 2 cycles once pipelining is set up:

- for L2 SRAM, 4 + 2 × M cycles
- for L2 Cache, 6 + 2 × M cycles

  where M is the number of misses.

This mechanism is described in *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610). Miss pipelining will be disrupted, if two misses occur to the same set or if the L1D stall is lengthened by any of the conditions listed in section 3.1.3. Note that when accessing memory sequentially, misses are *not* overlapped since on a miss one full cache line is allocated and the accesses to the next memory locations in the cache line 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 3–3. The assembly routine is shown in Figure 3–4.

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 (2.5 × *N* + 16) cycles to allocate *N* lines. This includes the execution of the code *and* miss penalties.

Figure 3–3. Memory Access Pattern of Touch Loop

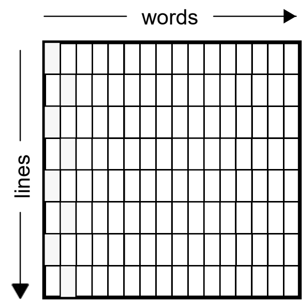


Figure 3–4. Touch Assembly Routine

```
* ========================================================================== *
*   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 cacheline 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.                                *
*                                                                            *
```

Figure 3–4. Touch Assembly Routine (Continued)

```
*   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.                *
* --------------------------------------------------------------------- *
*             Copyright (c) 2001 Texas Instruments, Incorporated.       *
*                            All Rights Reserved.                       *
* ===================================================================== *

        .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
```

Figure 3–4. Touch Assembly Routine (Continued)

```
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 (c) 2001 Texas Instruments, Incorporated.            *
*                            All Rights Reserved.                            *
* ========================================================================== *
```

3.1.5 Optimization Techniques Overview

The focus here 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.). L2 SRAM should be used for time-critical signal processing algorithms. Data can be directly streamed into L2 SRAM using EDMA, and memory accesses can 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 discussed 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.

3.2 Application-Level Optimizations

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

3.2.1 Choosing the Right L2 Cache Size

Choosing the right L2 cache size is particularly important for C621x/C671x devices, since the cache size also determines the associativity. A 16-Kbyte cache is direct-mapped, a 32-Kbyte cache is 2-way, a 48-Kbyte cache is 3-way, and a 64-Kbyte cache is 4-way set–associative. As a general rule, you should always try to use at least 32 Kbytes of L2 cache to be able to get a 2-way set-associativity. L2 cache should only be enabled if the code and/or data is too large to fit into L2 SRAM and has to be allocated in external memory.

3.2.2 Using DMA or L2 Cache

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

1) L2 SRAM is closer to the CPU; therefore, latency is reduced. If the buffers are located in external memory, data is first written from the peripheral to external memory by the DMA, cached by L2, then cached by L1D, before reaching the CPU.

2) Cache coherence is automatically maintained by the cache controller, no user action is required. 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 2.2, *Coherence,* explains in detail how to manage cache coherence.

3) The additional coherence operations may add to the latency. 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 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 2.2, *Coherence*, 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.

3.2.3 Signal Processing versus General-Purpose Processing Code

It may 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, its code better lends itself to a more careful optimization than general-purpose code. General-purpose processing is typically dominated by straight-line code, control flow and conditional branching. This code typically does not exhibit much parallelism and execution depends on many modes and 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 case 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. On C6211/C6711 devices, this has the additional benefit of a higher set-associativity. On C64x devices, the associativity of L2 cache is always 4 regardless of the size of the cache that can be configured between 32 Kbytes and 256 Kbytes. High set-associativity is crucial for general-purpose code, since it can hold multiple memory locations that would otherwise conflict in cache and would cause evictions. Due to the randomness of memory accesses, conflicts are highly likely.

DSP code and data may benefit from being allocated in L2 SRAM. This 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 CPU is accessing data, and/or to alter data structures to allow for more cache-friendly memory access patterns.

3.3 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 C62x and C64x DSP Library (DSPLIB) and is described in more detail in the Chapter 4, *Examples.*

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 3.3.2) 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 discussed in sections 3.3.4 and 3.3.5.

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 3.3.6.

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 3.3.7.

Avoiding stalls that are caused directly or indirectly by the write buffer are described in section 3.3.8.

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 3.3.3. A more comprehensive example that demonstrates this important concept is provided in section 4.3, *Processing Chain With DMA Buffering*.

3.3.1 Method of Analysis

In the following sections the effectiveness of optimization methods is demonstrated through examples. The performance is assessed by measuring:

- Execute Cycles that are raw CPU execute cycles
- L1D Stall Cycles that are predominantly caused by L1D Read Miss and L1D Write Buffer Full occurrences, and
- L1P Stall Cycles that are caused by L1P Misses. Note that as long as the execute pipeline stages are being supplied with execute packets, a fetch packet read miss does not stall the CPU.

The total cycles are the sum of the execute cycles, L1D stall cycles, and L1P stall cycles. The total cycle count was obtained by running the code on the hardware target. The number of misses and miss stall cycles were obtained either from the emulator or simulator depending on the available capabilities.

The number of read misses can be estimated by dividing the size of the array that is accessed by the line size. Typically, the number of misses is slightly higher due to stack accesses between function calls. The number of read miss stall cycles can then be estimated by multiplying the number of read misses with the number of stall cycles per miss. Since we assume that all code and data is allocated in L2 SRAM, the typical stall cycles per L1D read miss are 4 cycles for C621x/C671x devices and 6 cycles for C64x devices. The typical stall cycles per L1P miss are 5 cycles for C621x/C671x devices. For C64x devices, L1P miss stall cycles are difficult to estimate since misses are pipelined and the stall cycles depend on the average number of execute packets per fetch packet. As explained in section 3.1, the typical number of stall cycles per miss can increase due to other conditions.

In the performance tables, L1D stall cycles includes L1D read miss stall cycles, write buffer full stall cycles, and all stall cycles caused by the other conditions.

Note:

In this section, individual functions are benchmarked for the purpose of demonstrating cache optimization methods. These benchmarks are not intended to be used for performance estimation of an entire application. The cache overhead of individual functions can be misleading and is not indicative of the total cache overhead of an entire application. When a function is benchmarked, you have to consider the function *within* the context of the entire application. See section 3.3.2 for an example.

3.3.2 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 two. 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.

Consider Example 3–1 and Example 3–2. Converting the array x[ ] from *int* to *short* reduced the execution cycles from 5197 to 2245 on a C64x device. Table 3–1 shows the corresponding cycle count breakdown.

Example 3–1. Integer Data

```
int x[ ];
for (i=0; i<n; i++)
   r[i] = x[i] - c;
```

Example 3–2. Short Data

```
short x[ ];
for (i=0; i<n; i++)
   r[i] = x[i] - c;
```

Table 3–1. Cycle Count Breakdown for Example 3–1 and Example 3–2

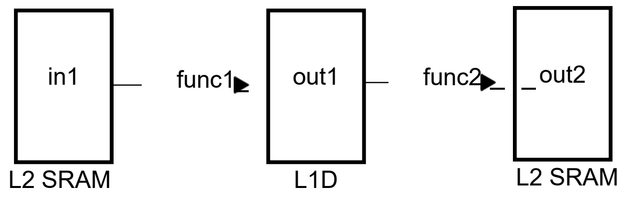
	Integer Data Type (Cycles)	**Short Data Type (Cycles)**
Execute Cycles	3117	1071
L1D Stall Cycles	2063	1152
L1P Stall Cycles	17	22
Total Cycles	5197	2245

The optimizations, enabled by using *short* data type, brought roughly a 2.3 times speed up, and the cache miss stalls were about halved.

3.3.3 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 3–5.

Figure 3–5. Processing Chain With 2 Functions



Consider Figure 3–6, 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 write 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.

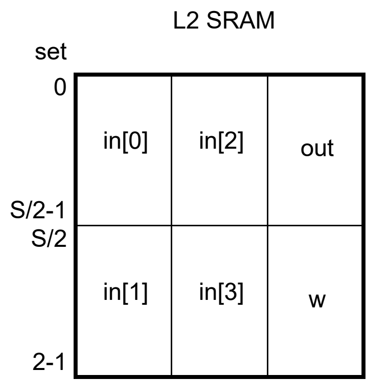
Figure 3–6. Channel FIR/Dot Product Processing Chain Routine

```
#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);
}
```

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 discussed in section 3.3.5, 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.

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 3–7 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 3–7. Memory Layout for Channel FIR/Dot Product Processing Chain Routine



3.3.3.1 C621x/C671x Device Example

For C621x/C671x devices, the number of taps, NH, was chosen to be 32 and the number of outputs, NR, 480. The number of input samples required is then NX = NR + NH – 1 = 480 + 32 – 1 = 511. The cycle counts are listed in Table 3–2. The FIR filter is reading NX + NH data elements, 543 16-bit elements or 1086 bytes spanning 34 lines. Accordingly, we see around 34 compulsory read misses for the FIR filter (the actual numbers may be slightly higher due to stack accesses). Additionally, we see 7 write buffer full occurrences that add to the read miss stalls. Therefore, for the first iteration the L1D stalls are higher than the expected 37 × 4 stalls cycles per miss = 148 read miss stall cycles. The dotprod routine is accessing 2 × NR × 16 bits = 1920 bytes spanning 60 cache lines. As expected, misses occur only during the first iteration. For all following iterations, w[ ] and out[ ] are in L1D. The FIR filter is taken from the C62x DSPLIB (DSP_fir_r8) and takes nr × nh/2 + 28 cycles to execute, 7708 cycles.

Table 3–2. Misses and Cycle Counts for FIR/Dot Product Example (C621x/C671x Devices)

	1st Iteration		**2nd Iteration**		**3rd Iteration**		**4th Iteration**		
NR = 480, NH = 32, S = 64	**fir**	**dot-prod**	**fir**	**dot-prod**	**fir**	**dot-prod**	**fir**	**dot-prod**	**Total**
Execute Cycles	7708	253	7708	253	7708	253	7708	253	31 844
L1D Stall Cycles	173	214	150	8	146	18	137	0	846
L1D Read Misses	37	60	35	2	33	4	34	0	205
L1D Write Buffer Full	7	0	0	0	0	0	0	0	7
L1P Stall Cycles	50	30	0	0	0	0	0	0	80
L1P Misses	10	6	0	0	0	0	0	0	16
Total Cycles	7931	497	7858	261	7854	271	7845	253	32 770

3.3.3.2 C64x Device Example

For C64x devices, the number of taps, NH, was chosen to be 64 and the number of outputs, NR, 1984. The number of input samples required is then NX = NR + NH – 1 = 1984 + 64 – 1 = 2047. The cycle counts are listed in Table 3–3. The FIR filter is reading NX + NH data elements, 2111 16-bit elements or 4222 bytes spanning 66 lines. Accordingly, we see around 66 misses for FIR. The dotprod routine is accessing 2 × NR × 16 bits = 7936 bytes spanning 124 cache lines. As expected, misses occur only during the first iteration. For all following iterations, w[ ] and out[ ] are in L1D. The FIR filter is taken from the C64x DSPLIB (DSP_fir_r8) and takes nr × nh/4 + 17 cycles to execute, 31 761 cycles.

Table 3–3. Misses and Cycle Counts for FIR/Dot Product Example (C64x Devices)

	1st Iteration		2nd Iteration		3rd Iteration		4th Iteration		
NR = 1984, NH = 64, S = 128	**fir**	**dot-prod**	**fir**	**dot-prod**	**fir**	**dot-prod**	**fir**	**dot-prod**	**Total**
Execute Cycles	31 766	520	31 766	520	31 766	520	31 766	520	129 144
L1D Stall Cycles	396	719	396	1	408	16	388	0	2327
L1D Read Misses	67	124	66	6	68	3	65	0	390
L1D Write Buffer Full	0	0	0	0	0	0	0	0	0
L1P Stall Cycles	58	53	0	0	0	0	0	0	111
L1P Misses	12	11	0	0	0	0	0	0	23
Total Cycles	32 220	1292	32 160	526	32 174	536	32 154	520	131 582

3.3.3.3 Interpretation of Cache Overhead Benchmarks

Note that if the cache overhead of the FIR filter and the dot product is determined in isolation, for C621x/C671x devices, the FIR filter has a low 2.2 percent overhead; whereas, the dot product has an 85 percent overhead. Even if it is taken into account that only the first call of dotprod has 85 percent overhead and all following calls have no overhead at all, the average overhead would still be 24 percent. However, the total cache overhead of the *processing chain* is only 2.7 percent. This is a good example that shows the cache overhead of an individual function can be misleading and is not indicative of the total cache overhead of an entire application. When a function is benchmarked, you have to consider the function in the context of the entire application.

Generally, cache overhead is a function of how much data is accessed relative to the amount of processing that is performed on the data. In case of the 32-tap FIR filter, relative little data is accessed while processing is quite expensive (for example, 543 input samples take 7708 cycles to process, that is 14 cycles per data element). The opposite is true for a dot product. A large amount of data is accessed, but only little processing is done on that data (for example, 960 input data take 253 cycles to process, that is 0.26 cycles per data element). The higher the ratio of processing cycles to data elements, the lower the cache overhead.

3.3.4 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 and 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 3–8. 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 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.

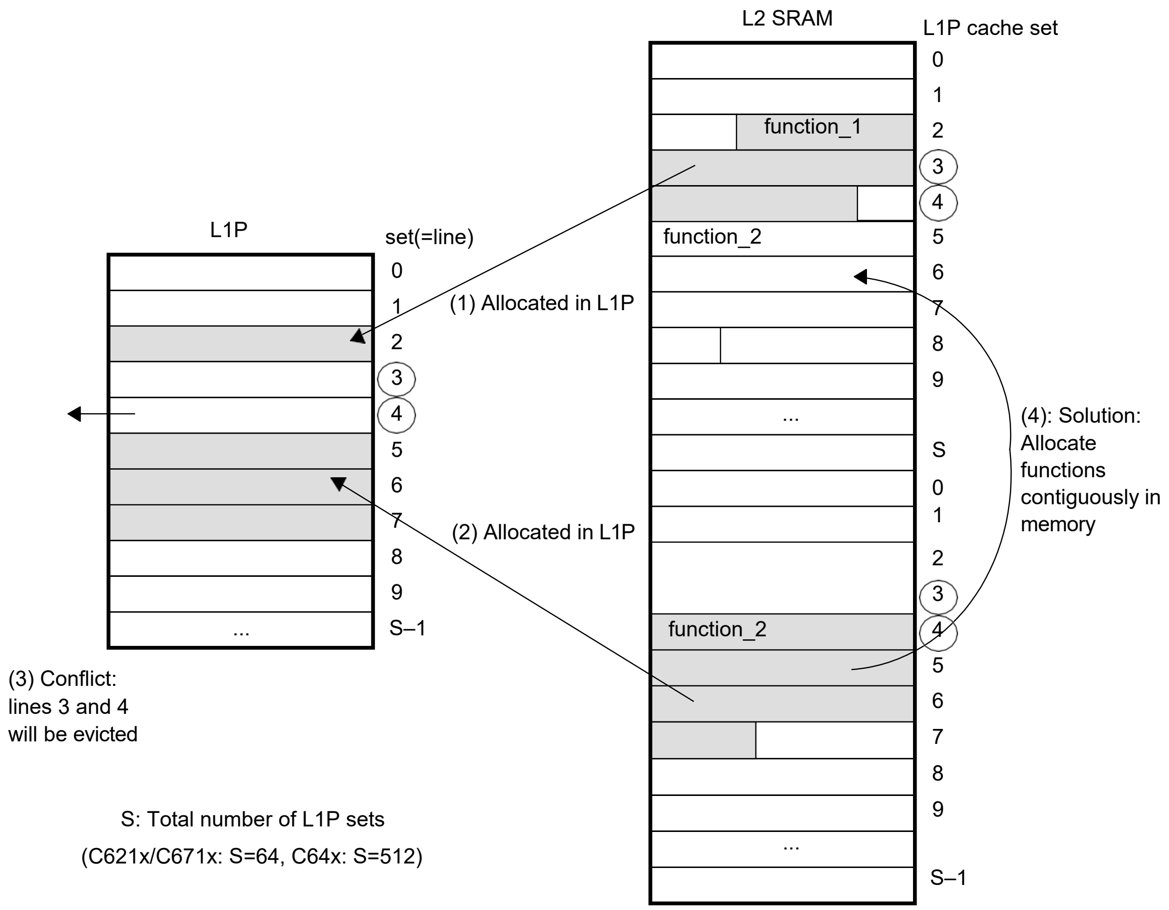
These type of misses are called conflict misses. They can be completely avoided by allocating the code of the two functions into nonconflicting sets. The most straightforward way this can be achieved is to place the code of the two functions contiguously in memory (4).

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 Example

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

Figure 3–8. Avoiding L1P Evictions


L2 SRAM
L1P cache set
function_1
function_2
function_2
L1P
set(=line)
(1) Allocated in L1P
(2) Allocated in L1P
(3) Conflict: lines 3 and 4 will be evicted
(4): Solution: Allocate functions contiguously in memory
S: Total number of L1P sets
(C621x/C671x: S=64, C64x: S=512)
0
1
2
3
4
5
6
7
8
9
S
S–1
...

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
{
       vecs:         o = 00000000h   l = 00000200h
       SRAM:         o = 00000200h   l = 0000FE00h
       CE0:          o = 80000000h   l = 01000000h
}

SECTIONS
{
       .vectors            >      vecs
       .cinit              >      SRAM
       .text:_function_1   >      SRAM
       .text:_function_2   >      SRAM
       .text               >      SRAM
       .stack              >      SRAM
       .bss                >      SRAM
       .const              >      SRAM
       .data               >      SRAM
       .far                >      SRAM
       .switch             >      SRAM
       .sysmem             >      SRAM
       .tables             >      SRAM
       .cio                >      SRAM
       .external           >      CE0
}
```

The linker will link all sections in exactly the order specified in the linker command file. 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 shell option can result in overall code size growth because any section containing code will be aligned to a 32-byte boundary to support the C6000 DSP branching mechanism.

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
{
       .vectors      >      vecs
       .cinit        >      SRAM
       .funct1       >      SRAM
       .funct2       >      SRAM
       .text         >      SRAM
       .stack        >      SRAM
...
}
```

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]);
}
```

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]);
}
```

3.3.5 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 and 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 set 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 according to 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 3.3.4 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;
}
```

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:

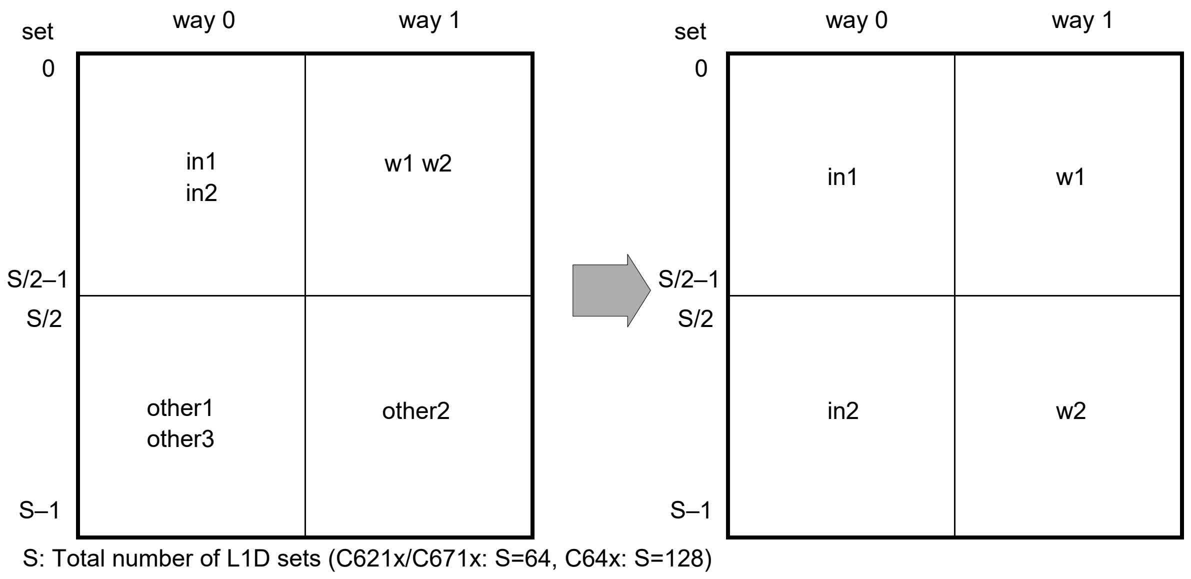
```
r1 = dotprod(in1, w1, N);
r2 = dotprod(in2, w2, N);
r3 = dotprod(in1, w2, N);
r4 = dotprod(in2, w1, N);
```

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 (2 Kbytes for C621x/C671x devices, and 8 Kbytes for C64x devices) 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 3–9. 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 3–9. 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 3–9. 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. Table 3–4 (page 3-32) shows the stall cycles for C621x/C671x devices and Table 3–6 (page 3-33) shows the stall cycles for C64x devices.

We expected *S* read misses in L1D, but the actual number is slightly higher. Additional misses occur if the arrays are not aligned at a cache line size boundary or due to stack access (benchmarks include function call overhead).

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 3–9. 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, it is recommended to assign the arrays 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. The new cycle counts for a C621x/C671x device are shown in Table 3–5 (page 3-33) and for a C64x device are shown in Table 3–7 (page 3-34). Now the data is reused from cache using the new memory configuration. The C64x L1D stall cycles were reduced from 2730 to 1560, a 43 percent reduction.

Note that for the C64x devices, 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 of miss pipelining on C64x devices can further reduce the cache miss stalls. The touch loop discussed in section 3.1.4 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);
```

The cycle counts are shown in Table 3–8 (page 3-34). The L1D stalls cycles have further reduced from 1560 to 544 cycles, a total reduction of 80 percent.

Table 3–4. Misses and Cycle Counts for Dot Product Example Before Optimization (C621x/C671x Devices)

N = 512, S = 64	**1st Call (in1,w1)**	**2nd Call (in2,w2)**	**3rd Call (in1,w2)**	**4th Call (in2,w1)**	**Total**
Execute Cycles	272	272	272	272	1088
L1D Stall Cycles	229	228	163	228	848
L1D Read Misses	64	64	32	64	224
L1D Write Buffer Full	0	0	0	0	0
L1P Stall Cycles	20	0	0	0	20
L1P Misses	4	0	0	0	4
Total Cycles	521	500	435	500	1956

Table 3–5. Misses and Cycle Counts for Dot Product Example After Optimization (C621x/C671x Devices)

N = 512, S = 64	**1st Call (in1,w1)**	**2nd Call (in2,w2)**	**3rd Call (in1,w2)**	**4th Call (in2,w1)**	**Total**
Execute Cycles	272	272	272	272	1088
L1D Stall Cycles	229	228	0	0	457
L1D Read Misses	64	64	0	0	128
L1D Write Buffer Full	0	0	0	0	0
L1P Stall Cycles	20	0	0	0	20
L1P Misses	4	0	0	0	4
Total Cycles	521	500	272	272	1565

Table 3–6. Misses and Cycle Counts for Dot Product Example Before Optimization (C64x Devices)

N = 2048, S = 128	**1st Call (in1,w1)**	**2nd Call (in2,w2)**	**3rd Call (in1,w2)**	**4th Call (in2,w1)**	**Total**
Execute Cycles	538	538	538	538	2152
L1D Stall Cycles	780	780	390	780	2730
L1D Read Misses	130	130	65	130	455
L1D Write Buffer Full	0	0	0	0	0
L1P Stall Cycles	35	8	5	8	56
L1P Misses	11	2	1	2	16
Total Cycles	1353	1326	395	1326	4400

Table 3–7. Misses and Cycle Counts for Dot Product Example After Optimization (C64x Devices)

N = 2048, S = 128	1st Call (in1,w1)	2nd Call (in2,w2)	3rd Call (in1,w2)	4th Call (in2,w1)	Total
Execute Cycles	538	538	538	538	2152
L1D Stall Cycles	780	768	6	6	1560
L1D Read Misses	130	128	1	1	260
L1D Write Buffer Full	0	0	0	0	0
L1P Stall Cycles	35	8	5	8	56
L1P Misses	11	2	1	1	16
Total Cycles	1358	1314	549	552	3278

Table 3–8. Misses and Cycle Counts for Dot Product Example With Touch Loop (C64x Devices)

N = 2048, S = 128	Touch	1st Call (in1,w1)	2nd Call (in2,w2)	3rd Call (in1,w2)	4th Call (in2,w1)	Total
Execute Cycles	144	538	538	538	538	2296
L1D Stall Cycles	515	0	22	0	7	544
L1D Read Misses	256	0	3	0	1	260
L1D Write Buffer Full	0	0	0	0	0	0
L1P Stalls	14	42	0	8	0	64
L1P Misses	3	12	0	2	0	17
Total Cycles	673	580	560	546	545	2904

3.3.6 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 (C621x/C671x devices) and Example 3–8 (C64x devices).

Example 3–7. Weighted Dot Product (C621x/C671x Devices)

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

       _nassert((int)w % 4 == 0);
       _nassert((int)x % 4 == 0);
       _nassert((int)h % 4 == 0);

       #pragma MUST_ITERATE(16,,2)
       for (i=0; i<N; i++)
         sum += w[i] * x[i] * h[i];

       return sum;
}
```

Example 3–8. Weighted Dot Product (C64x Devices)

```
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;
}
```

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. 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. The contents of the L1D set, at the time when an access is made, is listed in Table 3–9. It can be seen that whenever an array element is attempted to be read, it is not contained in L1D.

Table 3–9. Contents of an L1D Set at the Time When an Array is Accessed (Weighted Dot Product Example)

Read Access To	Way 0l	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

For C621x/C671x devices, the read miss stalls caused are shown in Table 3–10. In this case, N was chosen to be 1024, so each array is 2048 Kbytes, which is half the L1D capacity, and places the start of each array at the same set. We expect to see 3 × 1024 read misses, one for each element access. However, the number of measured read misses is much smaller, only 1089 misses. This is due to the fact the compiler uses 32-bit wide accesses to read two 16-bit array elements simultaneously and schedules two of those accesses in parallel. This reduces the overall number of memory accesses and introduces some amount of line reuse. Still, the number of read misses without thrashing should be 1024 elements × 2 bytes each × 3 arrays/32 bytes line size = 192 read misses.

Table 3–10. Misses and Cycle Counts for Weighted Dot Product Example (C621x/C671x Devices)

N = 1024	Original	Optimized
Execute Cycles	1556	1556
L1D Stall Cycles	4297	712
L1D Read Misses	1089	193
L1D Write Buffer Full	0	0
L1P Stall Cycles	25	20
L1P Misses	5	4
Total Cycles	5878	2288

For C64x devices, the read miss stalls caused are shown in Table 3–11. In this case, N was chosen to be 4096, so each array is 8192 Kbytes, which is half the L1D capacity, and places the start of each array at the same set. We expect to see 3 × 4096 read misses, one for each element access. However, the number of measured read misses is much smaller, only 2817 misses. This is due to the fact the compiler uses 64-bit wide accesses to read four 16-bit array elements simultaneously and schedules two of those accesses in parallel. This reduces the overall number of memory accesses and introduces some amount of line reuse. Still, the number of read misses without thrashing should be 4096 elements × 2 bytes each × 3 arrays/64 bytes line size = 384 read misses.

Table 3–11. Misses and Cycle Counts for Weighted Dot Product Example (C64x Devices)

N = 4096	**Original**	**Optimized**
Execute Cycles	3098	3098
L1D Stalls Cycles	20 485	2433
L1D Read Misses	2817	385
L1D Write Buffer Full	0	0
L1P Stalls Cycles	33	40
L1P Misses	13	13
Total Cycles	23 616	5571

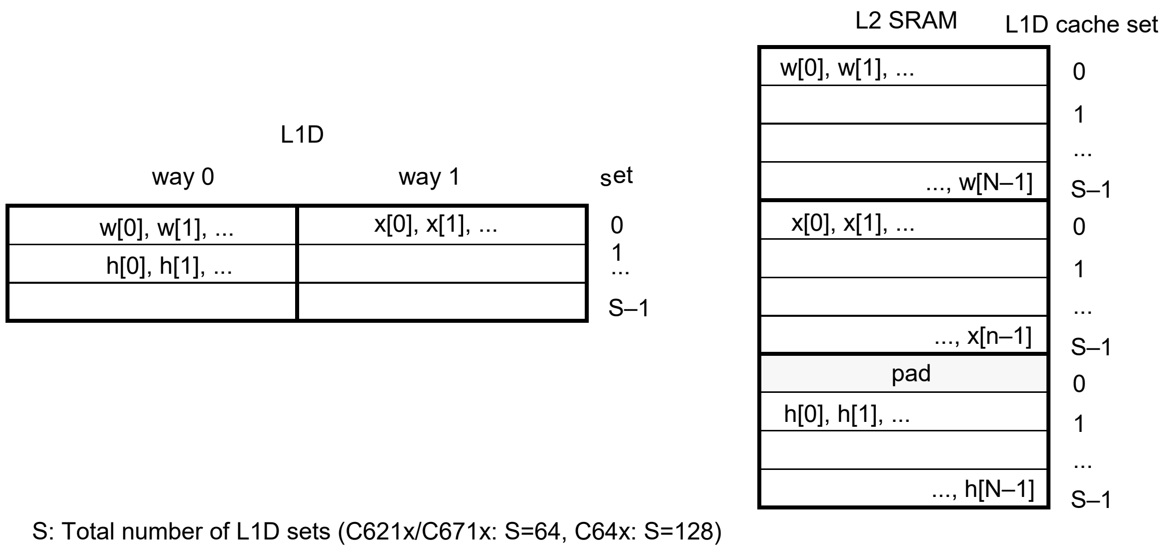
These conflict misses can be completely eliminated by allocating the data set contiguously in memory and pad arrays as to force an interleaved mapping to cache sets. For instance:

```
#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];
```

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 3–10. 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.

The cycle counts for the modified memory layout are listed in Table 3–10 and Table 3–11. L1D conflict misses were completely eliminated; hence, the number of L1D read misses now matches the expected ones.

Figure 3–10. Memory Layout and Contents of L1D After the First Two Iterations



3.3.7 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 in Example 3–6 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);
```

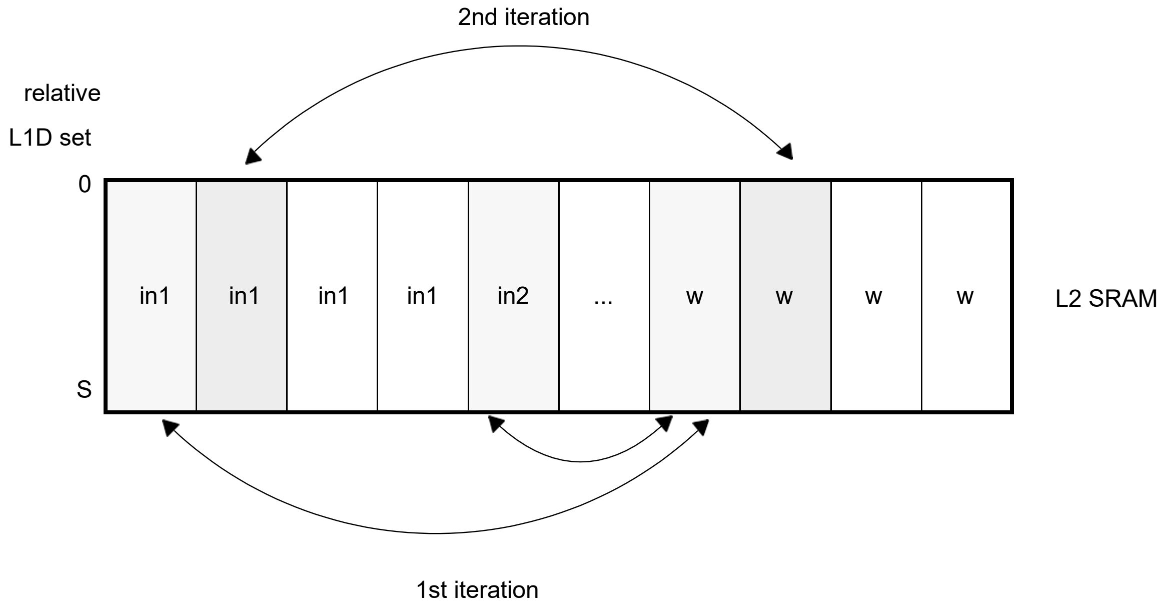
Assume that each array is twice the L1D capacity. We expect compulsory misses for in1[ ] and w[ ] for the first call. For the remaining calls, we expect compulsory misses for in2[ ], in3[ ], and in4[ ], but would like to reuse w[ ] from cache. However, after each call, the beginning of w[ ] has already 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 3–11. 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 $\times$ 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 next 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 3–11.Memory Layout for Dotprod Example



3.3.7.1 C621x/C671x Cycle Counts

For C621x/C671x devices, the cycle counts are shown in Table 3–12. The number of elements per vector N was set to 4096. We expect to see about 4096 elements × 2 bytes per element × 2 arrays/32 bytes per cache line = 512 read misses to occur. The size of each array is 8 Kbytes, twice the capacity of L1D. The total amount of data that is allocated in L1D for each call is 16 Kbytes, but only 4 Kbytes can be retained at a time. Since the two arrays are accessed in an interleaved fashion, by the time the first call to the routine has completed, only the last quarter of the arrays in1[ ] and w[ ] will reside in L1D. The following call then has to reallocate w[ ] again.

Through blocking we expect to save the capacity misses for the array w[ ], which are 4096 elements × 2 bytes × 3 arrays/32 bytes per line = 768. The actual cycle counts are shown in Table 3–12.

Table 3–12. Misses and Cycle Counts for Dot Product Example (C621x/C671x Devices)

N = 4096, S = 64	**Original**	**Optimized**
Execute Cycles	8269	8269
L1D Stall Cycles	7177	5639
L1D Read Misses	2048	1280
L1D Write Buffer Full	0	0
L1P Stall Cycles	50	50
L1P Misses	10	10
Total Cycles	15 496	14 225

3.3.7.2 C64x Cycle Counts

For C64x devices, the cycle counts are shown in Table 3–13 (page 3-45). The number of elements per vector N was set to 16 384. We expect to see about 16 384 elements × 2 bytes per element × 2 arrays/64 bytes per cache line = 1024 read misses to occur. The size of each array is 32 Kbytes, twice the capacity of L1D. The total amount of data that is allocated in L1D for each call is 64 Kbytes, but only 16 Kbytes can be retained at a time. Since the two arrays are accessed in an interleaved fashion, by the time the first call to the routine has completed, only the last quarter of the arrays in1[ ] and w[ ] will reside in L1D. The following call then has to reallocate w[ ] again.

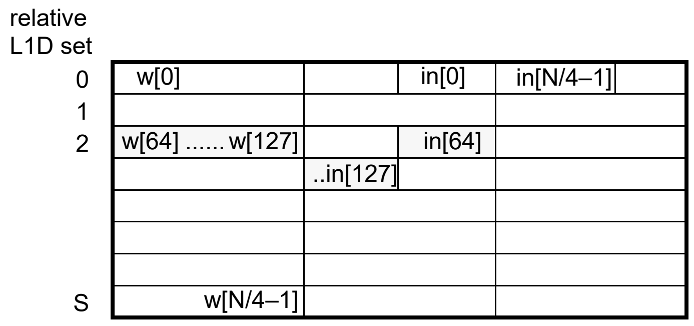
Through blocking we expect to save the capacity misses for the array w[ ], which are 16 384 elements × 2 bytes × 3 arrays/64 bytes per line = 1536. The actual cycle counts are shown in Table 3–14 (page 3-46). Read miss stalls were reduced by 23 percent from 24 704 to 15 528.

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(in2+o, N/4 * sizeof(short));
       dotprod(in2+o, w+o, N/4);

       touch(in3+o, N/4 * sizeof(short));
       dotprod(in3+o, w+o, N/4);

       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 can be achieved by aligning w[ ] and in[ ] at the same set, and within this set placing the start of w[ ] ahead of in[ ] as shown in Figure 3–12. Consider processing of elements 64 to 127. When element 127 of both w[ ] and in[ ] is accessed, in[127] is one line ahead, leaving the line of w[ ] in set 2 most recently used. Consequently, all lines of w[ ] will become MRU and will not be evicted.

Figure 3–12. Memory Layout for Dotprod With Touch Example



In this example, arrays w[ ] and in[ ] should be aligned to different memory banks to avoid bank conflicts. If arrays w[ ] and in[ ] are aligned to the same set first, and in[ ] is then aligned to bank 2 the desired memory layout is achieved:

```
#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 AND ensure w[ ] is MRU */
#pragma DATA_MEM_BANK(in1, 2)
short in1[N];
short in2[N];
short in3[N];
short in4[N];
```

The touch loop is called five times per iteration. For *M* read misses, each touch loop requires (2.5 × *M* + 16) cycles, in this case: 2.5 × N/4 × sizeof(short)/64 + 16 = 336 cycles. The dotprod routine then will not suffer any read miss stalls. The results are shown in Table 3–15 (page 3-46). This brings the total cycle count down to 24 100. Compared to the original cycle count of 41 253, this is a reduction of 42 percent.

Table 3–13. Misses and Cycle Counts for Dot Product Example (C64x Devices)

N = 16 384, S = 128	1st Call (in1,w)	2nd Call (in2,w)	3rd Call (in3,w)	4th Call (in4,w)	Total
Execute Cycles	4120	4124	4125	4125	16 494
L1D Stall Cycles	6176	6176	6176	6176	24 704
L1D Read Misses	1025	1025	1025	1025	4100
L1D Write Buffer Full	0	0	0	0	0
L1P Stall Cycles	46	7	2	0	55
L1P Misses	11	1	2	0	14
Total Cycles	10 342	10 307	10 303	10 301	41 253

Table 3–14. Misses and Cycle Counts for Dot Product Example After Blocking (C64x Devices)

N = 16 384, S = 128	1st Call (in1,w)	2nd Call (in2,w)	3rd Call (in3,w)	4th Call (in4,w)	Total Per Iteration	Total
Execute Cycles	1048	1048	1048	1048	4192	16 768
L1D Stall Cycles	1542	780	780	780	3882	15 528
L1D Read Misses	257	130	130	130	647	2588
L1D Write Buffer Full	0	0	0	0	0	0
L1P Stall Cycles	46	2	0	9	57	228
L1P Misses	10	2	0	2	14	56
Total Cycles	2636	1830	1830	1839	8135	32 540

Table 3–15. Misses and Cycle Counts for Dot Product Example With Blocking and Touch (C64x Devices)

N = 16 384, S = 128	Touch	1st Call (in1,w)	Touch	2nd Call (in2,w)	Touch	3rd Call (in3,w)	Touch	4th Call (in4,w)	Total Per Itera-tion	Total
Execute Cycles	165	1048	83	1048	83	1048	82	1054	4611	18 444
L1D Stall Cycles	518	6	259	12	259	12	265	20	1351	5404
L1D Read Misses	256	1	128	2	128	2	129	3	649	2596
L1D Write Buffer Full	0	0	0	0	0	0	0	0	0	0
L1P Stall Cycles	15	32	1	3	1	2	1	8	63	252
L1P Misses	5	10	1	2	1	1	1	2	23	92
Total Cycles	698	1086	343	1063	343	1062	348	1082	6025	24 100

3.3.8 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. If the write buffer is full and a write miss occurs, the CPU 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 section 3.1, the *TMS320C621x/ C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609), and the *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610).

Write buffer related stalls can be simply avoided by allocating the output buffer in L1D. Writes will then hit in L1D rather than being passed on to the write buffer. Consider the constant-vector add routine in Example 3–9.

Example 3–9. 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;
}
```

Assume the scenario shown in Example 3–10. 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.

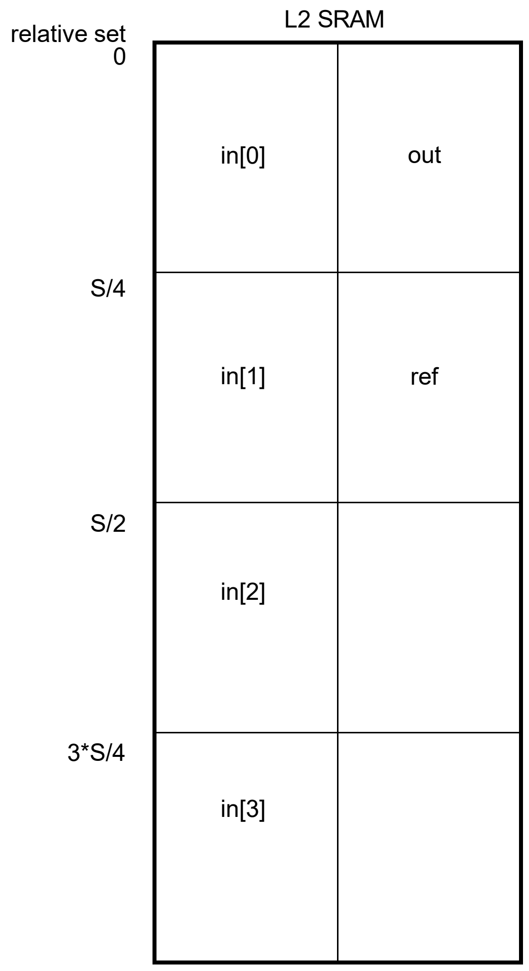
Example 3–10. Vecaddc/Dotprod Code

```
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);
}
```

3.3.8.1 C621x/C671x Device Example

The size of each array shall be 256 elements such that one array occupies one-eighth of L1D as shown in Figure 3–13. An optimized C version of the vecaddc routine was used that computes two results per cycle in the inner loop, that is, it takes N/2 cycles to execute plus some cycles for set-up code. Thus, we expect to see 128 execute cycles for vecaddc. The routine accesses 256 elements, 512 bytes spanning 16 cache lines. We expect to see 16 L1D read misses causing 16 $\times$ 4 stalls = 64 stall cycles. Additionally, there will be write buffer related stalls. First, one STW instruction is issued every cycle in the kernel. When the output array is not in L1D (for the first iteration in Example 3–10), the write buffer fills at a rate of one entry per cycle. However, since the write buffer drains only at a rate of one entry every 2 cycles, this will cause write buffer full stalls. Second, every time a read miss occurs, the write buffer will be drained completely to maintain proper program ordering.

Figure 3–13. Memory Layout for Vecaddc/Dotprod Example



The interaction of write buffer related stalls and read misses is listed in Table 3–16 (page 3-51). Consider the loop prolog and kernel shown in Figure 3–14 (page 3-52). Every cycle, 4 bytes are read from the input array. Therefore, after 8 execute cycles, 4 bytes $\times$ 8 cycles = 32 bytes are consumed that equals one cache line. The data words stored by STW shall be denoted A, B, C, ..., etc. In the first execution cycle of the prolog, a read miss occurs that costs 4 cycles. All subsequent 7 LDW's hit in L1D. The write buffer starts filling up in execute cycle 7 (the predicate for STW on cycle 6 is false). On execute cycle 9, the next read miss occurs. This time the write buffer contains data and needs to be drained that takes 3 cycles. One more cycle for entry B is required that already was at the top of the queue for one cycle, and two cycles for entry C. Then the write buffer starts filling again. On execute cycle 16, the write buffer was full when another write miss occurred for data word K. The write buffer takes another cycle to drain G, which frees up one entry for K. On execute cycle 17, the write buffer is full when the read miss occurs. This causes 7 additional write buffer drain stalls (2 cycles for I, J, and K; and 1 cycle for H). Again, on execute cycle 24, we take one stall cycle due to write buffer full. The pattern from execute cycle 17 to 24 now repeats. In summary, we expect to see the following number of L1D stall cycles: 4 + (4 + 2 + 1) + ((4 + 7 + 1) $\times$ 14) = 179.

The dotprod routine sees 32 read misses since it accesses 512 elements. Since the load instructions occur in parallel, the misses will be overlapped and we expect to see 32 misses/2 $\times$ 7 cycles for two parallel misses = 112 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. Read miss stalls will take 16 read misses $\times$ 4 stalls per miss = 64 stall cycles. Also, the dotprod routine will not incur any stalls since both out[ ] and ref[ ] arrays are held in L1D. The actual cycle counts are shown in Table 3–17 (page 3-53).

Table 3–16. Interaction of Read Miss and Write Buffer Activity for the First Call of Vecaddc (n = 0..13) (C21x/C671x Devices)

Execute Cycle	Read Activity	Write Buffer Contents
1	4 miss stalls	
2	hit	
3	hit	
4	hit	
5	hit	
6	hit	
7	hit	A
8	hit	A, B
9	4 miss stalls, 2 write buffer drain stalls	C
10	hit	C, D
11	hit	D, E
12	hit	D, E, F
13	hit	E, F, G
14	hit	E, F, G, H
15	hit	F, G, H, I
16	hit	1 write buffer full stall, G, H ,I, J
17 + 8*n	4 miss stalls, 7 write buffer drain stalls	K
18 + 8*n	hit	K, L
19 + 8*n	hit	L, M
20 + 8*n	hit	L, M, N
21 + 8*n	hit	M, N, O
22 + 8*n	hit	M, N, O, P
23 + 8*n	hit	N, O, P, Q
24 + 8*n	hit	1 write buffer full stall, O, P, Q, R

Figure 3–14. C621x/C671x Assembly Code for Prolog and Kernel of Routine vecaddc

```
;*----------------------------------------------------------------------------*
L1:           ; PIPED LOOP PROLOG

              LDW    .D1T1  *A0++,A3            ; (P) |10|
||            B      .S1    L2                  ; (P) |10|

              SUB    .L1X   B5,8,A1
||            B      .S1    L2                  ; (P) @|10|
||            LDW    .D1T1  *A0++,A3            ; (P) @|10|

              LDW    .D1T1  *A0++,A3            ; (P) @@|10|
|| [ A1]      SUB    .L1    A1,1,A1             ; (P) @@@|10|
|| [ A1]      B      .S1    L2                  ; (P) @@|10|

              MV     .S2X   A6,B5
|| [ A1]      B      .S1    L2                  ; (P) @@@|10|
|| [ A1]      SUB    .L1    A1,1,A1             ; (P) @@@@|10|
||            LDW    .D1T1  *A0++,A3            ; (P) @@@|10|

              MVK    .S2    0x1,B0              ; init prolog collapse predicate
||            ADD    .L2X   B4,A3,B4
||            LDW    .D1T1  *A0++,A3            ; (P) @@@@|10|
|| [ A1]      SUB    .L1    A1,1,A1             ; (P) @@@@@|10|
|| [ A1]      B      .S1    L2                  ; (P) @@@@|10|

;*----------------------------------------------------------------------------*
L2:           ; PIPED LOOP KERNEL

   [ B0]      SUB    .L2    B0,1,B0             ;
|| [!B0]      STW    .D2T2  B6,*B5++            ; |10|
||            ADD2   .S2X   A3,B4,B6            ; @|10|
|| [ A1]      B      .S1    L2                  ; @@@@@|10|
|| [ A1]      SUB    .L1    A1,1,A1             ; @@@@@@|10|
||            LDW    .D1T1  *A0++,A3            ; @@@@@@|10|

;*----------------------------------------------------------------------------*
```

Table 3–17. Misses and Cycle Counts for Vecaddc/Dot Product Example (C621x/C671x Devices)

	1st Iteration		2nd – 4th Iteration		
N = 256, S = 64	**vecaddc**	**dotprod**	**vecaddc**	**dotprod**	**Total**
Execute Cycles	147	146	147	146	1172
L1D Stall Cycles	183	116	64	0	491
L1D Read Misses	17	32	16	0	97
L1D Write Buffer Full	65	0	0	0	65
L1P Stall Cycles	25	30	0	0	55
L1P Misses	5	6	0	0	11
Total Cycles	354	287	211	146	1707

3.3.8.2 C64x Device Example

The size of each array shall be 1024 elements such that one array occupies one-eighth of L1D as shown in Figure 3–13. 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 256 execute cycles for vecaddc. The routine accesses 1024 elements, 2048 bytes spanning 32 cache lines. We expect to see 32 L1D read misses causing 32 × 6 stalls = 192 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 3–10), the write buffer fills at an average rate of one entry per cycle. 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 the high draining rate and support for write merging, the C64x write buffer does not generally suffer write buffer full stalls.

The interaction of write buffer related stalls and read misses is listed in Table 3–18. Consider the loop prolog and kernel shown in Figure 3–15 (page 3-55). 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 that equals one cache line. The data words stored by STDW 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 6 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 B 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: $6 + ((6 + 1) \times 31) = 223$.

The dotprod routine sees 64 read misses since it accesses 2048 elements. We expect to see 64 misses $\times$ 6 cycles = 384 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. Read miss stalls will take 32 read misses $\times$ 6 stalls per miss = 192 stall cycles. Also, the dotprod routine will not incur any stalls since both out[ ] and ref[ ] arrays are held in L1D. The actual cycle counts are shown in Table 3–19 (page 3-56).

Table 3–18. Interaction of Read Miss and Write Buffer Activity for the First Call of Vecaddc (n = 0..30) (C64x Devices)

Execute Cycle	Read Activity	Write Buffer Contents
1	6 miss stalls	
2		
3	hit	
4		
5	hit	
6		
7	hit	
8		A, B
9 + 8*n	6 miss stalls, 1 write buffer drain stall	
10 + 8*n		D, E
11 + 8*n	hit	E
12 + 8*n		F, G
13 + 8*n	hit	G
14 + 8*n		H, I
15 + 8*n	hit	I
16 + 8*n		J, K

Figure 3–15. 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|

;**----------------------------------------------------------------------------*
```

Table 3–19. Misses and Cycle Counts for Vecaddc/Dot Product Example (C64x Devices)

	1st Iteration		2nd – 4th Iteration		
N = 1024, S = 128	**vecaddc**	**dotprod**	**vecaddc**	**dotprod**	**Total**
Execute Cycles	272	280	272	280	2208
L1D Stall Cycles	342	384	192	0	800
L1D Read Misses	33	64	32	0	98
L1D Write Buffer Full	0	0	0	0	0
L1P Stall Cycles	19	25	0	0	0
L1P Misses	15	11	0	0	0
Total Cycles	633	689	464	280	3554

Chapter 4

Examples

This chapter provides examples that demonstrate the optimization techniques that were discussed in Chapter 3. The source code along with Code Composer Studio™ (CCS) projects files for each example are provided in the accompanying file archive.

The cache-optimized FFT from the Texas Instruments DSP Library (DSPLIB) is explained. It is described how it can be used to reduce the cache overhead for FFT data that exceeds the cache capacity. Additionally, a modified version of this FFT routine is presented that allows you to reduce the cache overhead even further.

The blocking method described in section 3.3.7, *Avoiding Capacity Misses*, is applied to a FIR filter with a large number of taps.

Finally, a typical processing chain configuration within a DMA double-buffering framework is described. It is explained how to choose memory layout that enables data reuse from cache and thus eliminating all cache misses except the compulsory misses for the input data. An in-depth analysis of the cache behavior is provided.

Topic **Page**

4.1 Cache-Optimized FFT

The C62x and C64x DSPLIB provide several optimized FFT routines. This example uses the *fft16x16r* routine that has been modified to allow for higher cache efficiency. For additional information on this routine including twiddle factor generation and appropriate input data scaling, refer to the *DSPLIB Programmer's Reference Guides* (SPRU565 for C64x devices and SPRU402 for C62x devices). The routine can be called in a single-pass or a multi-pass fashion. As single pass, the routine behaves like other DSPLIB FFT routines in terms of cache efficiency. If the total data size accessed by the routine fits in L1D, the single pass use of *fft16x16r* is most efficient. The total data size accessed for an N-point FFT is N × 2 complex parts × 2 bytes for the input data plus the same amount for the twiddle factors, that is, 8 × N bytes. Therefore, if 8 × N is smaller or equal to the capacity, then no special considerations need to be given to cache and the FFT can be called in the single-pass mode. Examples: the L1D capacity of a C621x/C671x device is 4 Kbytes. If N : : ; 512 (= 4 Kbytes/8), the single-pass FFT gives the best performance. For C64x devices, the L1D capacity is 16 Kbytes, thus if N : : ; 2048, the single-pass FFT is the best choice. The multi-pass version should be used whenever N > L1D capacity/8.

The fft16x16r routine has the following API:

```
void fft16x16r
(
       int                 N,
       short               *x,
       short               *w,
       unsigned char       *brev,
       short               *y,
       int                 n_min,
       int                 offset,
       int                 nmax
);
```

The function arguments are:

N	Length of FFT in complex samples, power of 2 ≤ 16 384
x[2*N]	Complex input data
w[2*N]	Complex twiddle factors
brev[64]	Bit reverse table (C62x device only)
y[2*N]	Complex data output
n_min	Smallest FFT butterfly used in computation. Used for decomposing FFT into sub-FFTs
offset	Index to sub-FFT from start of main FFT (in complex samples)
nmax	Size of main FFT in complex samples

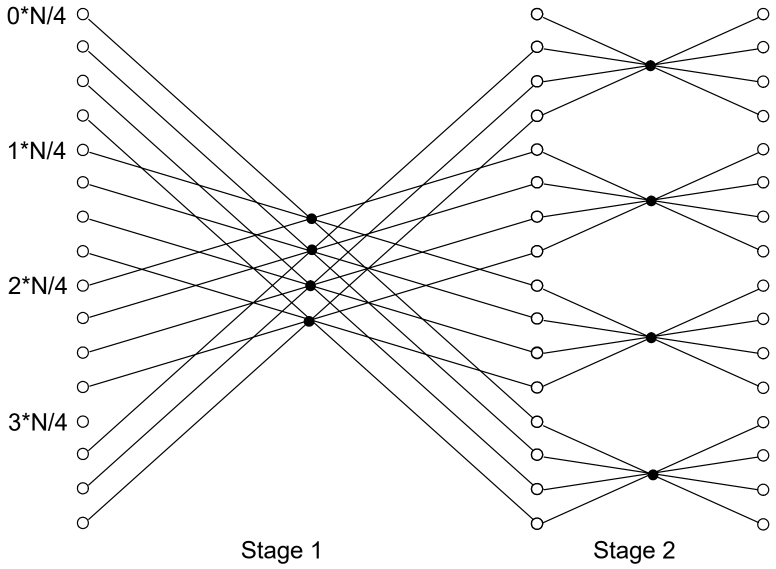
4.1.1 Structure of the Algorithm

The FFT routine uses a decimation-in-frequency algorithm that is able to perform radix-2 and radix-4 FFTs. In the first part of the algorithms, $\lceil \log_4(N-1) \rceil$ radix-4 stages are performed. The second part then performs either another radix-4 stage or a radix-2 stage in case N is a power of 2. Since for the final stage the twiddle factors are ± 1 and $\pm j$, no multiplications are required. The second part also performs a digit or bit-reversal. The computation of the first part is in-place, while the second part is out-of-place. In-place computation has the advantage that only compulsory read misses occur when the input and twiddle factors are read for the first time. The output of each stage is written back to the input array, which is kept in L1D, thus eliminating all write buffer related stalls and read misses for the following stage (provided all data fits into cache). Only the output of the last stage is passed through the write buffer to L2. Due to the high data output rate this results in write buffer full stalls.

4.1.2 Specially Ordered Twiddle Factors

All fftNxN type routines in the DSPLIB use a redundant set of twiddle factors where each radix-4 stage (Figure 4–1) has its own set of twiddle factors. The first stage comprises $3/4 \times N$ twiddle factors, the second stage $3/16 \times N$, the third stage $3/64 \times N$, etc. There are $\lceil \log_4 N \rceil$ stages and twiddle factor sets. All twiddle factors required for the entire FFT are contained in the first set; however, for the following stages, the twiddle factors required for a butterfly are no longer located contiguously in memory that would prevent the use of LDDW (for C64x devices) to load two complex twiddle factors. Another benefit of having individual twiddle factor sets per stage is that now the FFT can be decomposed into multiple smaller FFTs that fit into cache.

Figure 4–1. 16-Point Radix-4 FFT



4.1.3 FFT Data Fits in L1D

If all FFT data, input samples and twiddle factors fit into L1D, the routine only suffers compulsory misses.

4.1.3.1 C621x/C671x Device Example

All FFT data fits in L1D for N : : ; 512. A 512-point FFT causes 128 misses (= 8 bytes × 512 points/32 bytes per cache line) and results in 512 stall cycles (= 128 misses × 4 stalls per miss).

To assess potential write buffer full stalls in the last stage of the FFT, we observe that the kernel loop consists of 8 cycles and 4 STWs. The write buffer drains at a rate of 2 cycles per entry. Therefore, considering the average rate, we should not see any write buffer full stalls. However, if there had been bursts of writes, write buffer full conditions could have still occurred even though the average drain rate was not exceeded. In this case, the stores are scheduled such that the entries are allowed to drain before the write buffer becomes full. The write buffer contains no more than two entries at a time.

L1P stalls can be estimated through the code size, which is 1344 bytes. Thus, the number of L1P misses are 21 (= 1344 bytes/64 bytes per cache line) and the number of stalls are 105 (= 21 misses × 5 cycles per miss).

The cycle count formula for the C62x device is $2.5 \times N \times 1\log_4 Nl - N/2 + 164$.

The execute cycle counts are tabulated in Table 4–1 over the allowed range of N.

Thus, the expected total cycle count for N = 512 for C621x/C671x devices is 6308 + 512 + 105 = 6925.

Table 4–1. Execute Cycle Counts for a Single-Pass fft16x16r Routine

N	C621x/C671x Cycles	C64x Cycles
8	200	71
16	236	91
32	388	196
64	612	316
128	1380	741
256	2596	1381
512	6308	3326
1024	12 452	6526
2048	29 860	15 511
4096	59 556	30 871
8192	139 428	71 856
16 384	278 692	143 536

Figure 4–2 shows how the routine has to be called in a single-pass mode. The twiddle factors can be generated with the twiddle factor generator program provided with DSPLIB. The array brev has to be initialized with the values given in the DSPLIB Reference Guide. The argument n_min is set equal to the radix of the FFT (4 if N is a power of 4, and 2 if N is a power of 2). The arguments offset and nmax are only required for multi-pass mode, and for single-pass mode should always be 0 and N, respectively.

The benchmark results in Table 4–2 show 139 L1D read misses instead of the expected 128. The additional misses are due to accesses to the bit-reverse index table (up to 3 cache lines) and stack accesses (about 5 cache lines) within the routine. The write buffer full stalls are caused by a series of 20 stores for saving register contents on the stack.

Figure 4–2. Single-Pass 512-Point FFT Routine (C621x/C671x Devices)

```
#define N     512
#define RADIX 2

short x[2*N];        /* input samples                                        */
short w[2*N];       /* twiddle factors created by twiddle factor generator    */
short y[2*N];        /* output data                                          */
unsigned char brev[64];/* bit reverse index table, see DSPLIB Reference Guide */

fft16x16r(N, x, w, brev, y, RADIX, 0, N);
```

Table 4–2. Execute Cycle Counts for Single-Pass 512-Point FFT (C621x/C671x Devices)

N = 512	Cycles
Execute Cycles	6308
L1D Stall Cycles	600
L1D Read Misses	139
L1D Read Miss Stall Cycles	556
L1D Write Buffer Full Stall Cycles	15
L1P Stall Cycles	123
L1P Misses	22
Total Cycles	7016

4.1.3.2 C64x Device Example

All FFT data fits in L1D for $N \leq 2048$. A 2048-point FFT causes 256 misses (= 8 bytes × 2048 points/64 bytes per cache line), and results in 1536 stall cycles (= 256 misses × 6 stalls per miss).

To assess potential write buffer full stalls in the last stage of the FFT, we observe that the kernel loop consists of 5 cycles and 4 STWs. The write buffer drains at a rate of 1 cycle per entry. Therefore, considering the average rate, we should not see any write buffer full stalls. However, since the outputs are written in a bit-reversed order rather than contiguously, L2 bank conflicts may occur. Since an L2 access requires 2 cycles to complete, accesses to the same bank on consecutive cycles cause a stall (see section 3.1.3, *C64x Stall Conditions*). These bank conflicts show up as write buffer stalls, since they effectively slow down the drain rate to one entry every 2 cycles (insufficient for the number of STWs per cycle).

L1P cycles are difficult to estimate due to miss pipelining whose effectiveness depends on the number of instructions per execute packet.

The cycle count formula is $\lceil \log_4 N - 1 \rceil \times (5 \times N/4 + 25) + 5 \times N/4 + 26$. The execute cycle counts are tabulated in Table 4–1 over the allowed range of N. Thus, the expected total cycle count for N = 2048 for C64x devices is 15 511 + 1536 + L1D Write Buffer Full Stall Cycles + L1P Stall Cycles.

Figure 4–3 shows how the routine has to be called in a single-pass mode. The twiddle factors can be generated with the twiddle factor generator program provided with DSPLIB. The array brev is only required for the C62x version of the routine, and the pointer argument should be passed NULL. The argument n_min is set equal to the radix of the FFT (4 if N is a power of 4, and 2 if N is a power of 2). The arguments offset and nmax are only required for multi-pass mode, and for single-pass mode should always be 0 and N, respectively.

Figure 4–3. Single-Pass 2048-Point FFT Routine (C64x Devices)

```
#define N     2048
#define RADIX 2

short x[2*N];       /* input samples                                       */
short w[2*N];       /* twiddle factors created by twiddle factor  generator   */
short y[2*N];       /* output data                                         */

fft16x16r(N, x, w, NULL, y, RADIX, 0, N);
```

The penalty from the compulsory misses can be reduced by utilizing the touch loop. Assuming that x[ ] and w[ ] are allocated contiguously in memory, you can use one call to touch() to preallocate both arrays in L1D:

```
touch(x, 4*N*sizeof(short));
fft16x16r(N, x, w, NULL, y, RADIX, 0, N);
```

The cycle counts are shown in Table 4–3. Although the number of L1D read miss stall cycles were reduced to about one third, the overall cycle count with the touch loop is only about 3 percent lower.

Table 4–3. Execute Cycle Counts for Single-Pass 2048-Point FFT (C64x Devices)

		With Touch		
N = 2048	**Without Touch**	**Touch**	**FFT**	**Total**
Execute Cycles	15 520	143	15 520	15 663
L1D Stall Cycles	2724	516	1497	2013
L1D Read Miss Stall Cycles	1548	516	18	534
L1D Read Misses	258	256	3	259
L1D Write Buffer Full Stall Cycles	1176	0	1479	1479
L1P Stall Cycles	169	14	177	191
L1P Misses	30	5	29	34
Total Cycles	18 413	673	17 194	17 867

4.1.4 FFT Data is Larger Than L1D Cache

If the FFT data exceeds the capacity of L1D, not all data can be held in L1D. For each stage, input data and twiddle factors are read and the butterflies computed. The input data is overwritten with the results that are then read again by the following stage. When the capacity is exhausted, new read misses will evict already computed results (and twiddle factors) and the following stage suffers capacity misses. These capacity misses can be partially avoided if the computation of the FFT is split up into smaller FFTs that fit into cache.

Consider the radix-4 16-point FFT shown in Figure 4–1. After computing the first stage, all four butterflies of stage 2 are independent and can be computed individually. Generally after each radix-4 FFT stage, an N-point input data set splits into 4 separate N/4-point data sets whose butterflies are completely independent from one another. This tree structure of the FFT makes decomposition possible. Assume that the data for an N-point FFT exceeds L1D, but the N/4-point data fits. Instead of computing the entire FFT stage after stage, we stop after the first stage. We then obtained 4 N/4-point data sets. To complete the N-point FFT, we then compute 4 individual N/4-point FFTs on the output data of the first stage. Since each of the N/4-point FFT fits into cache, misses only occur for the first stage of each of the N/4-point FFT. If the FFT had been computed conventionally, each stage would have suffered capacity misses.

4.1.4.1 C621x/C671x Device Example

Assume a 2048-point FFT is computed. The data size is $8 \times 2048 = 16\ 384$ bytes; four times the size of L1D. After the first stage, we obtain four data sets with 512 points each. The data for a 512-point FFT fits into L1D. Hence, we can decompose the FFT into the first stage of a 2048-point FFT plus four 512-point FFTs that compute the remaining stages. The required calling sequence of the FFT routine is shown in Figure 4–4. The argument nmin is N/4 that specifies the routine to exit after the stage whose butterfly input samples are a stride of N/4 complex samples apart (after the first stage). For the sub-FFTs, nmin becomes RADIX that means the FFT is computed to the end. The first argument now changes to N/4 to indicate that we compute N/4-point FFTs. The argument offset indicates the start of the sub-FFT in the input data set. The pointers to the input array have to be offset, accordingly. Since each FFT stage has its own set of twiddle factors, the twiddle factors for the first set ($3 \times$ N/4 complex twiddle factors) are skipped and set to point to the ones for the second stage. All sub-FFTs start at the second stage; the pointer into the twiddle factor array is the same for all sub-FFT calls.

Figure 4–4. Cache-Optimized 2048-Point FFT Routine (C621x/C671x Devices)

```
#define N     2048
#define RADIX 2

short x[2*N];       /* input samples                                          */
short w[2*N];       /* twiddle factors created by twiddle factor generator    */
short brev[64];     /* bit reverse index table, see DSPLIB Reference Guide    */
short y[2*N];       /* output data                                            */

/* Stage 1 of N-point FFT:                                                    */
fft16x16r(N,   &x[      0], &w[      0], brev, &y[0],   N/4,     0, N);

/* Four N/4-point FFTs:                                                       */
fft16x16r(N/4, &x[    0], &w[2*3*N/4], brev, &y[0], RADIX, 0, N);
fft16x16r(N/4, &x[2*1*N/4], &w[2*3*N/4], brev, &y[0], RADIX, 1*N/4, N);
fft16x16r(N/4, &x[2*2*N/4], &w[2*3*N/4], brev, &y[0], RADIX, 2*N/4, N);
fft16x16r(N/4, &x[2*3*N/4], &w[2*3*N/4], brev, &y[0], RADIX, 3*N/4, N);
```

For the first call, we can expect compulsory and conflict misses. For all following calls, there are only compulsory misses due to the first time access of data and twiddle factors. Since the sub-FFTs process a 512-point input data set, we can expect the same number of cache stalls as for Figure 4–2 (see Table 4–2). The cycle counts measured are shown in Table 4–4.

Table 4–4. Execute Cycle Counts for Multi-Pass 2048-Point FFT (C621/C671x Devices)

N = 2048	1st Call	2nd Call	3rd–5th Call	Total
Execute Cycles	5138	6308	6308	30 370
L1D Stall Cycles	10 302	633	565	12 630
L1D Read Miss Stall Cycles	10 287	612	550	12 549
L1D Read Misses	2572	153	137	3136
L1D Write Buffer Full Stall Cycles	15	15	15	75
L1P Stall Cycles	103	10	10	143
L1P Misses	21	2	2	29
Total Cycles	15 543	6946	6883	43 138

The total execute cycle count for a single-pass FFT is 29 860. This is very close to the actual measured 30 370 cycles. It is slightly higher due to the call overhead for calling the function five times. Most of the L1D stalls are caused, as expected, by the first call (first FFT radix-4 stage). Once the FFT fits into L1D, we mostly see the compulsory misses for the data.

For a 4096–point FFT, two stages have to be computed to obtain 16 256-point FFTs that fit into cache. The code would then change as shown in Figure 4–5.

Figure 4–5. Cache-Optimized 4096-Point FFT Routine (C621x/C671x Devices)

```
#define N 4096
#define RADIX 4

short x[2*N];        /* input samples */
short w[2*N];        /* twiddle factors created by twiddle factor generator */
short brev[64];      /* bit reverse index table, see DSPLIB Reference Guide */
short y[2*N];        /* output data */

/* Stage 1 and 2 of N–point FFT: */
fft16x16r(N, &x[0], &w[0], brev, &y[0], N/16, 0, N);

/* 16 N/16–point FFTs: */
for(i=0;i<16;i++)
 fft16x16r(N/16, &x[2*i*N/16], &w[2*(3*N/4+3*N/16)], brev, &y[0], RADIX, i*N/16,  N);
```

4.1.4.2 C64x Device Example

Assume an 8192-point FFT is computed. The data size is 8×8192 = 65 536 bytes; four times the size of L1D. After the first stage, we obtain four data sets with 2048 points each. The data for a 2048-point FFT fits into L1D. Hence, we can decompose the FFT into the first stage of an 8192-point FFT plus four 2048-point FFTs. The required calling sequence of the FFT routine is shown in Figure 4–6. The argument nmin is N/4 that specifies the routine to exit after the stage whose butterfly input samples are a stride of N/4 complex samples apart (after the first stage). For the sub-FFTs, nmin becomes RADIX that means that the FFT is computed to the end. The first argument now changes to N/4, to indicate that we compute N/4-point FFTs. The argument offset indicates the start of the sub-FFT in the input data set. The pointers to the input array have to be offset, accordingly. Since each FFT stage has its own set of twiddle factors, the twiddle factors for the first set ($3 \times$ N/4 complex twiddle factors) are skipped and set to point to the ones for the second stage. All sub-FFTs start at the second stage; the pointer into the twiddle factor array is the same for all sub–FFT calls.

Figure 4–6. Cache-Optimized 8192-Point FFT Routine (C64x Devices)

```
#define N     8192
#define RADIX 2

short x[2*N];       /* input samples                                          */
short w[2*N];       /* twiddle factors created by twiddle factor generator    */
short y[2*N];       /* output data                                            */

/* Stage 1 of N-point FFT:                                                    */
fft16x16r(N,   &x[      0], &w[      0], NULL, &y[0],   N/4,     0, N);

/* Four N/4-point FFTs:                                                       */
fft16x16r(N/4, &x[     0], &w[2*3*N/4], NULL, &y[0], RADIX,     0, N);
fft16x16r(N/4, &x[2*1*N/4], &w[2*3*N/4], NULL, &y[0], RADIX, 1*N/4, N);
fft16x16r(N/4, &x[2*2*N/4], &w[2*3*N/4], NULL, &y[0], RADIX, 2*N/4, N);
fft16x16r(N/4, &x[2*3*N/4], &w[2*3*N/4], NULL, &y[0], RADIX, 3*N/4, N);
```

For the first call, we can expect compulsory and conflict misses. For all following calls, there are only compulsory misses due to the first time access of data and twiddle factors. The total execute cycle count according to the single-pass cycle count formula is 71 856. The cycle breakdown for the 2048-point sub-FFTs is expected to be similar to the single-pass 2048-point FFT (see Table 4–3, page 4-8). The cycle counts measured are shown in Table 4–5. Note that an 8192-point single-pass FFT takes 156 881 cycles; whereas, the multi-pass FFT takes only 128 956 cycles, an 18 percent improvement.

The stall overhead relative to the ideal formula cycle count is 79 percent. Using the touch loop here is not very effective. Due to the high number of conflict misses in the first stage, the miss pipelining of the compulsory read misses results in only small savings. For the following stages, L1D contains a large amount of dirty data. Whenever a dirty cache line has to be evicted, miss pipelining is disrupted. Hence, the touch loop again shows little effect.

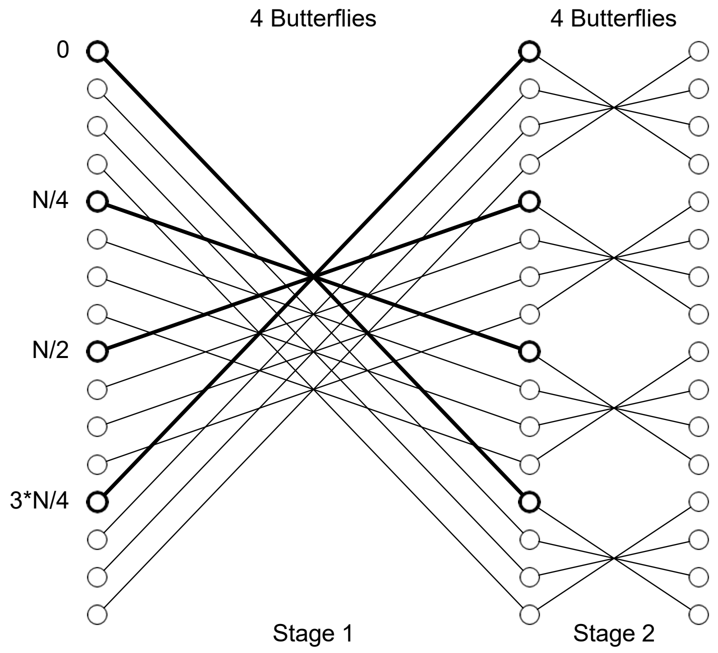
Table 4–5. Execute Cycle Counts for Multi-Pass 8192-Point FFT (C64x Devices)

N = 8192	**1st Call**	**2nd Call**	**3rd–5th Call**	**Total**
Execute Cycles	10 293	15 520	15 520	72 373
L1D Stall Cycles	43 993	3522	2962	56 401
L1D Read Miss Stall Cycles	43 993	1638	1386	49 789
L1D Read Misses	7333	273	231	8299
L1D Write Buffer Full Stall Cycles	0	1884	1576	6612
L1P Stall Cycles	130	31	7	182
L1P Misses	27	5	2	38
Total Cycles	54 416	19 073	18 489	128 956

4.1.5 Avoiding L1D Thrashing on C64x Device

The examples in section 4.1.4 showed that the majority of cache misses occur in the initial FFT stages that exceed the cache capacity. The majority of these misses are conflict misses caused by thrashing. To see how these conflict misses are caused, consider the computation of the first two butterflies of the stage 1 in the 16-point radix-4 FFT shown in Figure 4–7.

Figure 4–7. Radix-4 Butterfly Access Pattern



The first butterfly accesses the input data elements x[0], x[N/4], x[N/2] and x[3*N/4]. The second butterfly accesses the elements x[2], x[N/4 + 2], x[N/2 + 2] and x[3*N/4 + 2]. Assume N/4 complex input samples consume one cache way, that is, half of the L1D capacity (if N = 8192 for C64x device). In this case, all four addresses of the elements of the two butterflies map to the same set. Initially, elements x[0] and x[N/4] are allocated in L1D in different ways. However, the access of x[N/2] and x[3*N/4] evicts these two elements. Since x[0] and x[2] share the same line, the access to x[2] misses again, as do all remaining accesses for the second butterfly. Consequently, every access to an input sample misses due to conflicts, making no use of line reuse. Instead of one miss per line, we see one miss per data access. The accesses to twiddle factors may interfere with input data accesses, but this is insignificant. The twiddle factors are ordered such that they accessed linearly for a butterfly, thus avoiding conflicts.

These conflict misses can be avoided if each of the four input elements of a butterfly maps to a different set. To achieve this, the input data is split into four sets consisting of N/4 complex samples each and a gap of one cache line is inserted after each set as shown in Figure 4–8. This new memory layout requires that the FFT routine is modified such that an offset is added to the input data array indices as shown in Table 4–6 (note that x[ ] is defined as short and the increment of i is 2 since we access complex samples). The modified assembly routine can be found in the corresponding example CCS project.

Figure 4–8. New Memory Layout for FFT Input Data

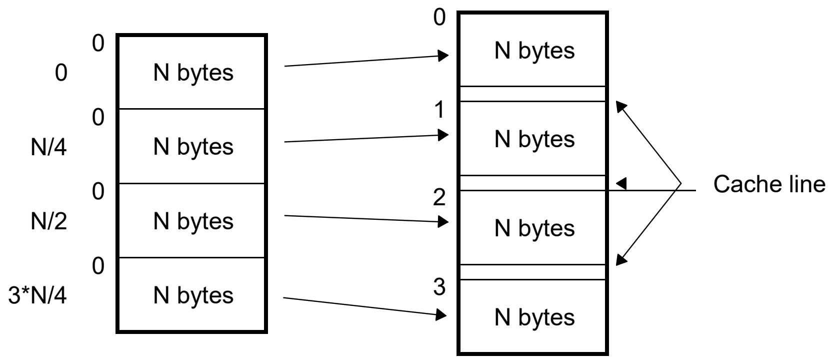


Table 4–6. New Butterfly Access Pattern

Normal Access Pattern	New Modified Access Pattern
x[i]	x[i]
x[N/4 + i]	x[N/4+ i + L1D_LINESIZE/2]
x[N/2 + i]	x[N/2+ i + L1D_LINESIZE]
x[3*N/4 + i]	x[3*N/4 + i + 3*L1D_LINESIZE/2]

The new access pattern is used for all FFT stages that exceed the cache capacity. Once the FFT has been decomposed to the stage where an entire FFT data set fits into cache, the normal access pattern is used. Before the modified FFT routine can be called, the four input data sets have to be moved apart. This can be done with the DSPLIB routine DSP_blk_move. The size of the input array has to be increased by 3 cache lines. Also, since the output data of the first stage is split up, we have to offset the start of each sub-FFT, accordingly. The argument list of the FFT routine was modified to accept a flag gap that indicates if the input data was split up (gap = 1) or is contiguous (gap = 0). The exact details of the procedure is shown in Figure 4–9 (N = 8192).

Figure 4–9. 8192-Point FFT with Cache-Optimized Access Pattern (C64x Devices)

```
#define CACHE_L1D_LINESIZE 64
#define N      8192
#define RADIX  2
#define GAP    1
#define NO_GAP 0

short x  [2*N];                               /* input samples            */
short xc [2*N+ 3*CACHE_L1D_LINESIZE/2];       /* input samples split up   */
short w  [2*N];     /* twiddle factors created by twiddle factor generator */
short y  [2*N];                               /* output data              */

/* Copy and split the input data */
for(i=0; i<4; i++)
{
      touch   (&x[2*i*N/4], 2*N/4*sizeof(short));
      DSP_blk_move(&x[2*i*N/4], &xc[2*i*N/4 + i*CACHE_L1D_LINESIZE/2], 2*N/4);
}

/* Compute first FFT stage */
fft16x16rc(N, &x_s[0], &w[0], GAP, y, N/4, 0, N);

/* Compute remaining FFT stages */
for(i=0; i<4; i++)
  fft16x16rc(N/4,&xc[2*i*N/4+ i*CACHE_L1D_LINESIZE/2], &w[2*3*N/4], NO_GAP, y,
               RADIX, 3*N/4, N);
```

The savings in the number of conflict misses far outweighs the additional overhead introduced by splitting up the input data. Splitting the data can also be performed in the background, if a DMA is used. The CSL DAT_copy routine can be used for this purpose:

```
/* Copy and split the input data */
for(i=0; i<4; i++)
  id = DAT_copy(&xi[2*i*N/4], &x[2*i*N/4 + i*CACHE_L1D_LINESIZE/2],
                 2*N/4*sizeof(short));

/* Perform other tasks */
...
DAT_wait(id);

/* Compute first FFT stage */
...
```

The cycle count results are shown in Table 4–7. Note that the number of L1D stalls is greatly reduced from 43 993 to only 6101 stall cycles. The overhead due to stalls and the block move operation is 34 percent. If background DMAs are used instead of block moves, the overhead drops to 26 percent.

Table 4–7. Execute Cycle Counts for Multi-Pass FFT with Cache-Optimized Access Pattern (C64x Devices)

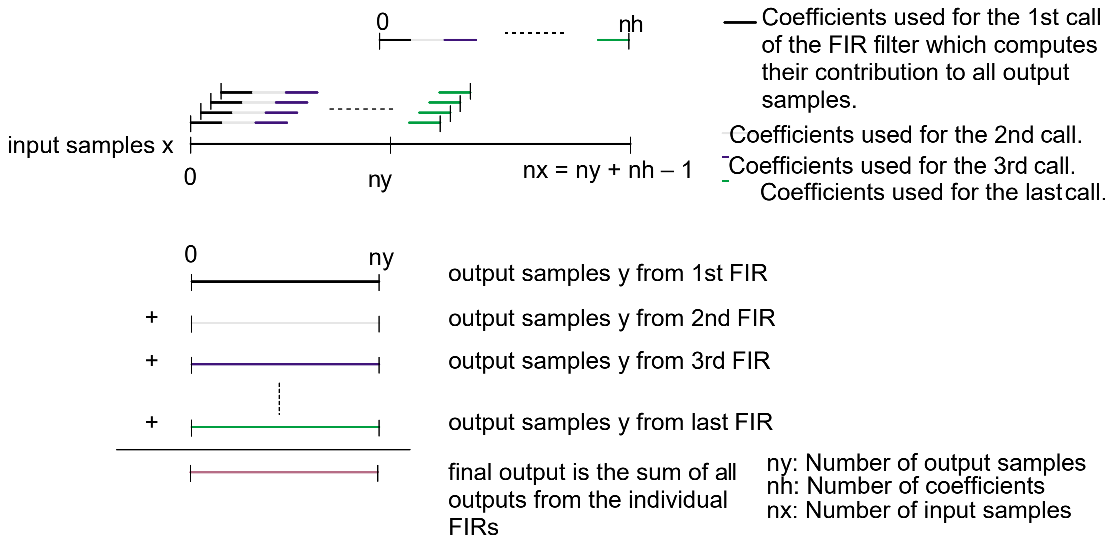
	blk_move	1st Pass	Total With blk_move	Total Without blk_move
Execute Cycles	4620	10 294	77 090	72 470
L1D Stall Cycles	1100	6101	19 098	17 998
L1D Read Miss Stall Cycles	1100	6101	12 990	11 890
L1D Read Misses	514	903	2334	1820
L1D Write Buffer Full Stall Cycles	0	0	6108	6108
L1P Stall Cycles	50	141	227	177
L1P Misses	14	27	49	35
Total Cycles	5771	16 536	96 415	90 644

4.2 FIR With Large Number of Coefficients

Assume a FIR filter is computed with a large number of coefficients that exceed the capacity of one way of L1D. A conventional implementation would result in a high number of capacity misses because after the convolution for one output is completed, the input data and coefficients required for the next output will have been evicted. Thus, data cannot be reused across filter outputs.

In order to reduce the number of capacity misses, the blocking method described in section 3.3.7, *Avoiding Capacity Misses*, can be used to split the FIR filter into multiple smaller filters such that the same coefficients can be reused before they get evicted. A subset of coefficients is kept in L1D and used to compute its contribution to all output samples, then the next subset of coefficients is brought into cache, and so on. While one subset of coefficients is retained in L1D, new inputs are continuously brought into cache (one new input sample for each output in the steady state). This is shown in Figure 4–10. After all subfilters have been calculated, their individual contributions to each output sample are summed up to yield the final output sample.

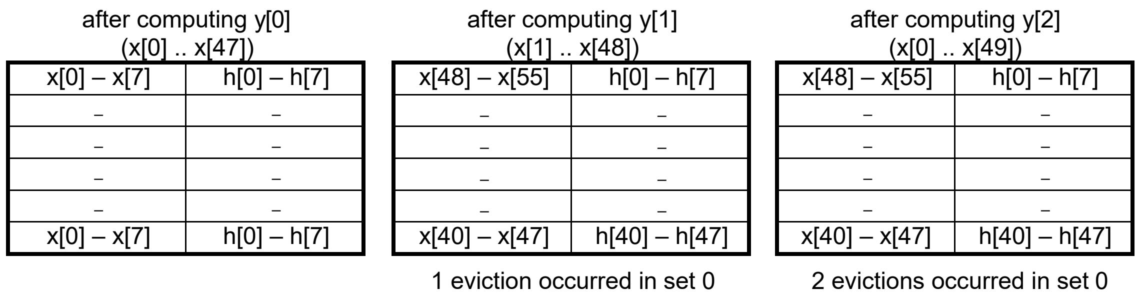
Figure 4–10. Splitting Up a Large FIR Filter


0
nh
Coefficients used for the 1st call of the FIR filter which computes their contribution to all output samples.
Coefficients used for the 2nd call.
Coefficients used for the 3rd call.
Coefficients used for the last call.
input samples x
0
ny
nx = ny + nh – 1
0
ny
output samples y from 1st FIR
+
output samples y from 2nd FIR
+
output samples y from 3rd FIR
+
output samples y from last FIR
final output is the sum of all outputs from the individual FIRs
ny: Number of output samples
nh: Number of coefficients
nx: Number of input samples

While new input samples replace old ones in cache, this does not matter as long as the coefficients are retained. If the coefficients are kept in one way and the input samples in the other (the size of the coefficient subset is not larger than one way of L1D), the coefficients will not be evicted by input samples. The behavior of the LRU replacement scheme automatically ensures that the lines which are not reused are replaced, leaving the ones that are reused in the cache.

What is the optimal number of coefficients per subfilter? Note that for the computation of ny output samples, each coefficient is allocated in cache only once. However, each input sample has to be reallocated for each subfilter computation. Therefore, more coefficients per subfilter results in fewer cache misses. Using half the L1D capacity (equal to one cache way) for coefficients can be expected to be most efficient. However, due to the nature of convolution, there are evictions during the computation of each output, if a full cache way is allocated for the coefficients. Consider the accesses required for each filter output, assuming a hypothetical cache architecture with 6 sets and 8 coefficients per line as shown in Figure 4–11. Computation of the first output y[0] requires input samples x[0] to x[47]. The next output y[1] requires x[1] to x[48]. However, x[48] maps to set 0 and evicts the line containing x[0] to x[7]. For output y[3], which requires x[2] to x[49], the access to x[2] will then miss. At the same time, this evicts x[48] and x[49] that are required later. Consequently, there will be 2 misses for each filter output. Even though 48 samples can be packed into 6 lines, they span 7 lines when they do not start at a cache line boundary. The cache capacity, in terms of number lines accessed rather data size, is exceeded. The repeated eviction and reallocation of the cache line can be avoided, if the number of coefficients is reduced such that they occupy one cache line less. That is, if only 40 instead of 48 coefficients are used in the example, the cache capacity is never exceeded. This is shown in Figure 4–12. The optimum subset size in bytes for the coefficient array, therefore, is the size of one cache way less one line.

Figure 4–11. Cache Contents After Each Filter Output for Maximum Number of Filter Coefficients



after computing y[0] (x[0] .. x[47])

x[0] – x[7]	h[0] – h[7]
–	–
–	–
–	–
–	–
x[0] – x[7]	h[0] – h[7]

after computing y[1] (x[1] .. x[48])

x[48] – x[55]	h[0] – h[7]
–	–
–	–
–	–
–	–
x[40] – x[47]	h[40] – h[47]

1 eviction occurred in set 0

after computing y[2] (x[0] .. x[49])

x[48] – x[55]	h[0] – h[7]
–	–
–	–
–	–
–	–
x[40] – x[47]	h[40] – h[47]

2 evictions occurred in set 0

Figure 4–12. Cache Contents After Each Filter Output for Optimum Number of Filter Coefficients

after computing y[0]
(x[0] .. x[47])

x[0] – x[7]	h[0] – h[7]
–	–
–	–
–	–
x[32] – x[39]	h[32] – h[39]
–	–

after computing y[1]
(x[1] .. x[48])

x[48] – x[55]	h[0] – h[7]
–	–
–	–
–	–
x[32] – x[39]	h[32] – h[39]
x[40] – x[47]	–

1 miss occurred in set 5

after computing y[2]
(x[0] .. x[49])

x[0] – x[7]	h[0] – h[7]
–	–
–	–
–	–
x[32] – x[49]	h[32] – h[39]
x[40] – x[47]	–

no misses occured

4.2.1 C621x/C671x Device Example

In this example, we use the routines DSP_fir_r8 and DSP_fir_r4 from the C62x DSPLIB (refer to the *DSPLIB Programmer's Reference Guide*, SPRU402, for a description of the routine).

Assume that a block of 512 outputs of a 4000-tap FIR filter is to be computed, NH = 4000 and NR = 512. One L1D cache way is 2048 bytes and can hold 1024 16-bit coefficients. We subtract one cache line of coefficients to obtain the optimum number of filter coefficients per sub-FIR, 1024 – 16 = 1008. This leaves three sub-FIRs with 1008 coefficients plus one sub-FIR with 76 coefficients. The code for the multi-pass FIR filter is shown in Figure 4–13.

Figure 4–13. Optimized Multi-Pass FIR Filter Routine (C621x/C671x Devices)

```
#pragma DATA_ALIGN(x, CACHE_L1D_LINESIZE)
#pragma DATA_ALIGN(h, CACHE_L1D_LINESIZE)
short x[NX]   ;      // input
short h[NH]   ;      // coefficients
short r[NR]   ;      // output
short r1[NR]  ;      // intermediate output

DSP_fir_r8(x, h, r, NH_SET, NR);
for (j=1; j<3; j++)
{
       DSP_fir_r8(x + j*NH_SET, h + j*NH_SET, r1, NH_SET, NR);
       for (i=0; i<NR; i++) r[i] += r1[i];
}
DSP_fir_r4(x + j*NH_SET, h + j*NH_SET, r1, NH_LAST_SET, NR);
for (i=0; i<NR; i++) r[i] += r1[i];
```

Table 4–8 shows the measured cycle counts for the optimized multi-pass filter. For comparison, the cycle counts for the unoptimized filter that is performed in a single call to DSP_fir_r8 are also shown. The ideal cycle count for the DSP_fir_r8 is nh × nr/2 + 28 = (4000 × 512)/2 + 28 = 1 024 028 cycles. The number of input samples accessed is 4000 + 512 – 1 = 4511. The number of compulsory L1D read misses is 532 (= 4511 input samples + 4000 coefficients) × 2 bytes/32 bytes per line). The remainder are capacity misses. By splitting up the filter into multiple calls, the number of capacity misses is greatly reduced from 128 265 to only 799. The cache stall overhead is reduced from 53 percent to 0.69 percent.

Table 4–8. Execute Cycle Counts for Single-Pass and Multi-Pass FIR Filter (C621x/C671x Devices)

	Single Pass	**Multi Pass**
Execute Cycles	1 024 074	1 027 450
L1D Stall Cycles	545 264	3415
L1D Read Misses	128 265	799
L1D Write Buffer Full	0	0
L1P Stall Cycles	55	192
L1P Misses	11	33
Total Cycles	1 569 338	1 031 057

4.3 Processing Chain With DMA Buffering

This example illustrates some of the cache optimization techniques described in section 3.3, *Procedural-Level Optimizations*. The example uses the C64x device as a target, but applies conceptually also to C621x/C671x devices. It consists of a simple processing chain within a DMA double-buffering framework. This type of scenario is frequently found in typical DSP applications. In this example, a horizontal wavelet filter (wave_horz) is applied to an image that is located in external memory. Since the filter routine operates on 16-bit data, two additional routines are required that convert the 8-bit image data to 16-bit data, and conversely (pix_expand and pix_sat). All three routines are taken from the Texas Instruments C64x Image and Video Processing Library (IMGLIB).

The data flow is shown in Figure 4–14. The image is transferred from external memory to a buffer in L2 SRAM using DMA. The three processing steps are then applied, and the output transferred back to external memory. While one DMA input buffer is being processed, the second one is being filled in the background. Figure 4–15 shows how the routines of the processing chain are called.

Note that the optimization techniques presented here can also be applied to processing chains outside a DMA double-buffering framework.

Figure 4–14. Data Flow of Processing Chain

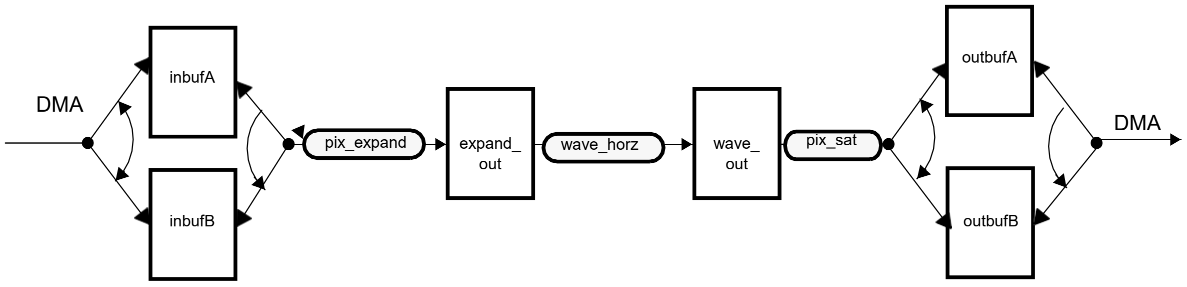


Figure 4–15. Code for Processing Chain

```
pix_expand(ROWS*COLS, inbuf, expand_out);
for(i=0; i<=ROWS; i++)
       wave_horz(&expand_out[i*COLS], qmf, mqmf,
              &wave_out[i*COLS], COLS);
pix_sat(ROWS*COLS, wave_out, outbuf);
```

4.3.1 Data Memory Layout

We will now discuss optimizations that ensure the highest possible data throughput. The library code of the three routines is already optimized to achieve fastest possible execution. However, further consideration has to be given to overhead caused by DMA set up and cache misses. This overhead can be minimized by choosing the right memory layout that takes into account the size of each of the buffers and their relative location in memory.

The size of the DMA buffers determine how many transfers have to be initiated to bring the image into on-chip memory. Every transfer initiation costs some cycles to set up the DMA controller. Therefore, the buffers should be large enough to minimize DMA setup cycles. On the other hand, they should be small enough to leave on-chip memory for other critical data and code. The buffer size also affects the delay for initial and final DMA transfers. In this example, a DMA buffer size of 16 Kbytes is chosen, corresponding to 16 384 pixels. If we assume that the image size is 512 × 512 pixels, then 16 DMA transfers are required.

In a processing chain, the output of one function forms the input of the next function. If the buffers that interface the functions (expand_out and wave_out in Figure 4–14) can be retained in L1D, there will be no cache misses inside the processing chain, thus completely eliminating read miss and write buffer related stalls. The only read misses that then occur are compulsory misses for the first routine, pix_expand, that reads new data from the DMA buffers. The last routine in the chain writes its results through the write buffer to the DMA output buffer in L2 SRAM. Note that the first time the processing chain is executed, read accesses to these interface buffers will miss. However, all following iterations will then access these buffers in L1D.

What is the appropriate memory layout that ensures that the interface buffers are retained in cache? In the example, we have to consider read accesses to four buffers: the DMA input buffers *inbufA* and *inbufB*, and the interface buffers *expand_out* and *wave_out*. Whereas the interface buffers are being reused from cache (within and across iterations of the processing chain), the DMA buffers are continuously filled with new data by the DMA. The data is written to L2 SRAM and, consequently, allocated in L1D. When the DMA writes to a buffer that is held in L1D, the internal cache coherence protocol is triggered that invalidates the lines of the buffer. This ensures that the latest data is read from memory, rather than stale data from the cached copy of the buffer. Eventually, an appropriate memory layout has to fulfill the requirement that both interface buffers fit completely into cache and are not evicted by allocation of lines of the input buffers.

A memory layout that fulfills this requirement is shown in Figure 4–16. Since the input buffer size was chosen to be 16 Kbytes and is larger than one cache way (8 Kbytes), the only method to protect the interface buffers from being evicted is to make sure they allocate in the opposite way of the input buffer. This leaves the interface buffers with a size of 4 Kbytes each. Since the interface buffers will always be most recently used, the LRU replacement scheme automatically ensures that the interface buffers are not evicted when new lines from the input buffers are allocated. New input buffer lines always replace old input buffer lines.

The size of the interface buffers determines how much data can be processed at a time. Since we have 4 Kbytes for results, we can at most expand a block of 2048 8-bit pixels to 16 bits. The two input buffers can be placed anywhere in memory. However, the two interface buffers should be allocated contiguously; otherwise, they may map to conflicting sets in L1D and would be subject to eviction by input buffer lines. Figure 4–17 shows a C code example for the necessary array declarations.

Note that we have neglected the wavelet filter coefficients in this discussion. Since they occupy only 32 bytes (half a cache line), they have very little impact on the overall cache miss overhead.

Figure 4–16. L1D Memory Layout

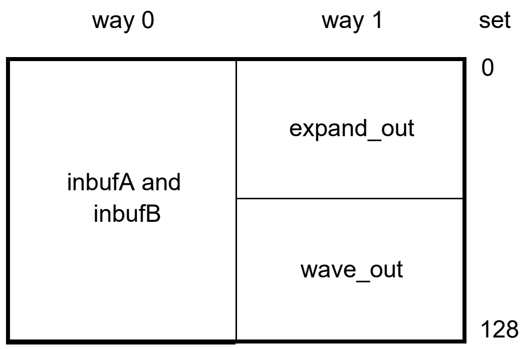


Figure 4–17. Array Declarations

```
#define COLS   512
#define DMABUF 16384
#define TMPBUF (4*COLS) /* 2048 */
#pragma DATA_ALIGN(inbufA,CACHE_L1D_LINESIZE)
#pragma DATA_SECTION(expand_out, ”.tmpbuf”)
#pragma DATA_SECTION(wave_out,   ”.tmpbuf”)
unsigned char inbufA [DMABUF];
unsigned char inbufB [DMABUF];
unsigned char outbufA[DMABUF];
unsigned char outbufB[DMABUF];
short expand_out[TMPBUF];
short wave out  [TMPBUF];
```

4.3.2 Miss Pipelining

The previous discussion showed how to eliminate most of the cache misses. The only cache misses left are now the compulsory misses due to the first time reference of the input data. Their impact can be reduced by using the touch routine, that effectively exploits miss pipelining by allocating two cache lines per cycle. The 6 stall cycles per read miss reduces to an incremental stall penalty of only 2 cycles per miss. Figure 4–18 shows the code of the processing chain after optimization. Also, the touch routine can be used to preallocate the interface buffers prior to the first iteration. This has the additional effect of also eliminating write buffer related stalls during the first iteration.

Figure 4–18. Code for Processing Chain After Optimization

```
#define COLS 512
for(j=0; j<8; j++)
{
       touch(&inbuf[j*TMPBUF], TMPBUF);
       pix_expand(TMPBUF, inbuf, expand_out);
       for(i=0; i<=4; i++)
              wave_horz(&expand_out[i*COLS], qmf,
                     mqmf, &wave_out[i*COLS], COLS);
       pix_sat(ROWS*COLS, wave_out, outbuf);
}
```

4.3.3 Program Memory Layout

Now we consider program cache misses. Since the three routines of the processing chain are repeatedly executed, they should be contiguously allocated in memory to avoid evictions and thus conflict misses. Additionally, the code for setting up DMA transfers (for example, DAT_copy and DAT_wait routines from the Texas Instruments chip support library) and the touch routine should also be taken into account. An example linker command file that achieves this is shown in Figure 4–19.

Figure 4–19. Linker Command File

```
SECTIONS
{
...
       .text:_pix_expand   > L2SRAM
       .text:_wave_horz    > L2SRAM
       .text:_pix_sat      > L2SRAM
       .text:_touch        > L2SRAM
       .text:_DAT_copy     > L2SRAM
       .text:_DAT_wait     > L2SRAM
...
       .tmpbuf             > L2SRAM
}
```

4.3.4 Benchmark Results

Since the image size is 512 × 512 pixels and we process 2048 pixels (4 rows) at a time, the processing chain is executed a total of 128 times. Table 4–9 shows a detailed cycle breakdown into execute cycles, data cache (L1D) read miss stalls, and program cache (L1P) read miss stalls for each routine. The first time the processing chain is executed, the interface buffers and the code have to be allocated into cache (cold cache). Therefore, the cycle counts for the first iteration are shown separately.

Table 4–9. Cache Stall Cycle Count Breakdown for Fully-Optimized Processing Chain

	1st Iteration (Cold Cache)			All Following 127 Iterations			
	Pixel Expand (With Touch)	Wavelet Filter (With Touch)	Pixel Saturation (With Touch)	Pixel Expand (With Touch)	Wavelet Filter	Pixel Saturation	Total
Execute Cycles	436	4322	450	436	4280	401	655 067
L1D Stall Cycles	112	144	134	76	0	0	10 042
L1P Stall Cycles	18	89	23	0	0	0	130
Total Cycles	566	4555	607	512	4280	401	665 239
Cache Overhead (includes touch execute cycles)	39.4%	6.6%	51.8%	26.7%	0%	0%	2.2%

The individual routines are now discussed. The cycle count formula for each routine is listed in Table 4–10. Assume that the first DMA input buffer has been filled, and the routine pix_expand is executed. For 2048 pixels, the routine takes 399 cycles to execute. We also have to add the number of execute cycles for the touch routine, which is 32 cycles. The total execute cycle count therefore is 431 cycles. The routine accesses 2048 pixels spanning 2048/64 bytes per cache line = 32 cache lines. Since we use the touch loop, we expect 6 + (2 × 32) = 70 read miss stalls. However, in the first iteration, L1D may still contain dirty data that needs to be written back to memory that will disrupt miss pipelining. Also, we may see a slightly higher number of misses due to stack activity. Therefore, the actual L1D stall cycle count, 112 cycles, is somewhat higher than the estimate. For all following iterations, the L1D stall cycles are 76, closer to the estimate. Also, no L1P miss stalls occur.

Table 4–10. Cycle Count Formulas

Routine	Formula	Pixels = 2048
pix_expand	(3 × pixels/16)+ 15	399
wave_horz	rows × (2 × columns + 25)	4196
pix_sat	(3 × pixels/16) + 13	397
touch	(pixels + 124)/128 + 16	32 (8-bit pixels) 48 (16-bit pixels)

The wavelet filter takes 4196 cycles to execute plus 48 cycles for the touch routine in the first iteration. The actual cycle count, 4322 in the first iteration, is slightly higher due to the loop overhead. Since the pixels have been expanded to 16 bits, the number of read misses is 64 (= 2048 pixels × 2 bytes/64 bytes per cache line), resulting in 6 + (2 × 64) = 134 stall cycles. The actual stall count is 144, again due to spurious stack accesses. In the following iterations, this routine will not have any L1D or L1P read misses.

The routine pix_sat executes in 397 cycles plus 48 cycles for the touch routine in the first iteration. The measured cycle count, 450, comes very close to this estimate. L1D read miss stalls are about the same as for the wavelet routine since both routines access the same amount of data. Again, for all following iterations this routine will not have any L1D or L1P read misses.

Table 4–9 summarizes all cycle counts. Note that the cache overhead for executing an algorithm the first time is as high as 51.8 percent. However, when the algorithm is called repeatedly within an optimized processing chain, cache related stalls are completely eliminated, except for the compulsory misses of the first routine that reads new data. The total cache overhead of the optimized processing chain is only 2.2 percent.

The DMA setup overhead can be estimated as follows. The number of DMA buffers that have to be transferred is 32 = 256 Kbytes image size/16 Kbytes DMA buffer size × 2 (from and to external memory). Each transfer requires one initiation (DAT_copy) and a loop to wait for completion (DAT_wait). Benchmarking results for DMA setup and wait code are around 130 and 30 cycles, respectively. The total number of cycles spent on DMA management, therefore, is (130 + 30) × 32 = 5120 cycles. However, we can expect additional stall cycles due to conflicts between CPU and DMA accesses. The measured total cycle count without DMA is 666 580 cycles and with DMA 674 582 cycles. This is a difference of 8002 cycles, made up of 5120 cycles for DMA management, as determined above, and 2882 stall cycles.

There will also be some overhead due to the loop control code required for DMA double buffering. The breakdown of all overhead cycle counts is given in Table 4–11. Taking into account both cache and DMA related overhead cycles, the relative total overhead is now 3.6 percent for the fully-optimized version. Note, that the total cycle count omits the memory transfer cycles for the first DMA buffer, since these can usually overlapped with other CPU processing. Table 4–11 also shows cycle counts for implementations without cache optimization and/or DMA for comparison. If we had used DMA but not performed any cache optimizations, the overhead would have increased by about a factor of 10 to 35.2 percent. Further, if we had relied on L2 cache for accesses to the external memory instead of using DMA, but had performed cache optimizations, the overhead would have been 155 percent.

The results show that benchmarking algorithms individually to assess cache stall cycles can be very misleading. Instead, algorithms have to be considered within the context of an entire application. It was also shown that for high throughput signal processing algorithms, DMA combined with cache optimizations achieves the best possible performance.

Table 4–11. Cycle Count Results

	DMA and Cache Optimization	DMA, But No Cache Optimization	L2 Cache With Cache Optimization
L1D Stall Cycles	10 042	219 615	1 008 224
L1P Stall Cycles	130	633	1 489
Touch Execute Cycles	4 192	0	0
DMA Management Cycles	5 120	5 120	0
DMA Relative Stall Cycles	2 882	2 246	0
Loop Control Cycles	1 341	1 341	0
Total Overhead	23 707	228 955	1 009 713
Image Processing Execute Cycles	650 875	650 875	650 875
Relative Total Overhead	3.6%	35.2%	155%

Index