Cache Coherence and Optimization in Embedded Applications

Oliver Sohm

Texas Instruments,
12500 TI Boulevard, MS 8635
Dallas, TX 75243

o-sohm1@ti.com

ABSTRACT

This paper first provides a detailed introduction into the basic operation of caches and presents an overview of the TMS320C64x DSP cache architecture. In order to guarantee correct functioning of embedded applications on a cache-based DSP architecture, programmers have to be aware of potential cache coherence issues. While cache coherence on C6x1x DSPs is automatically maintained for internal memory, it is the programmer's responsibility for accesses to external memory. A simple set of programming guidelines is established that ensures that coherence is maintained correctly. Further, the main purpose of a cache is to reduce the average memory access time. When the CPU requests data that is not held in cache a longer access time is incurred. Therefore, the goal is to keep those address locations allocated in cache that are repeatedly used or likely to be used soon. Since the capacity of caches is limited, memory layout and access pattern have a large impact on which addresses can be kept in cache and which will be evicted. Various optimization techniques are discussed for designing cache-friendly memory layouts to improve cache efficiency and thus application performance.

1. Introduction

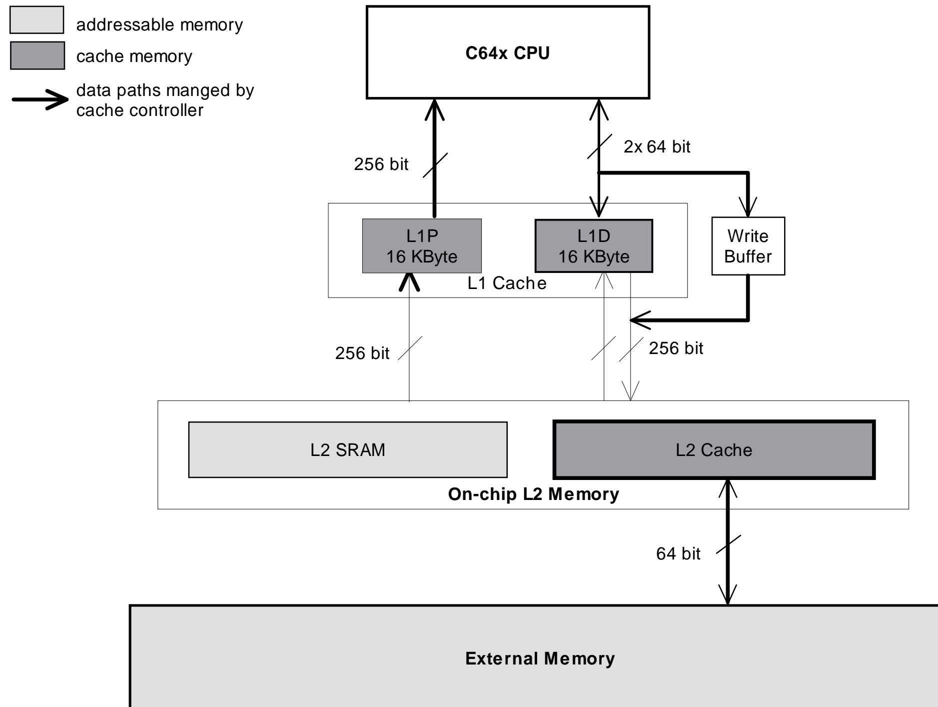
Memory caches have had a place in general purpose computer architectures for a long time. Until recently digital signal processors drew their processing power from efficient low complexity architectures supported by special purpose instructions. However, modern DSP cores such as the C64x push the clock rate to levels where the slower memory system has a negative impact on overall performance. Cache-based memory architectures consist of a small but fast cache memory close to the CPU and a larger but slower main memory. This combination reduces cost, chip area and power consumption while enabling an average memory access time that is dominated by the access time of the fast cache memory. Through the introduction of caches in DSPs the system designer has a powerful device at hand but also faces new challenges posed by optimizing embedded applications for a hierarchical memory architecture.

This tutorial paper first explains the basics of cache operation in Section 2. The relevant cache terminology is introduced and the different types of cache explained on the example of C64x caches. This lays the foundation for the two following sections that will describe cache coherence and discuss various code optimization techniques to improve cache efficiency. Section 3 explains the interaction between CPU and DMA accesses in a cache memory system and describes how a cache coherence protocol works. A basic understanding of these protocols are required since in some application scenarios involving DMA buffers in external memory it is the programmer's responsibility to maintain coherence. Section 4 presents code and memory optimization methods that improve the efficiency of cache and thus reduce the impact of CPU stalls caused by cache activity.

2. Fundamentals of Cache Operation

This section explains in detail the different types of cache architectures and how they work. Generally, one can distinguish between direct-mapped caches and set-associative caches [1]. The caches will be described using C64x L1P (direct-mapped) and L1D (set-associative) as examples, however the concept is the same for C621x/C671x and in fact is similar for all cache-based computer architectures.

Figure 1: C64x cache memory architecture



The C64x and C621x/C671x memory architecture consists of a 2-level internal cache-based memory plus optional external memory. Level 1 cache is split into program (L1P) and data (L1D) cache. On C64x devices each L1 cache is 16 Kbytes and on C621x/C671x devices 4 Kbytes large. Level 1 memory can be accessed by the CPU without stalls. Data write misses are passed through a write buffer directly to Level 2 memory (L2). L2 memory is configurable and can be split into L2 SRAM (addressable on-chip memory) and L2 Cache for caching external memory addresses. On C6416 for instance, the size of L2 is 1 Mbyte and on a C621x/C671x device L2 is 64KBytes. External memory can be several Mbytes large. Figure 1 shows the C64x architecture. All caches and data paths shown are automatically managed by the cache controller.

2.1 Direct-Mapped Caches

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

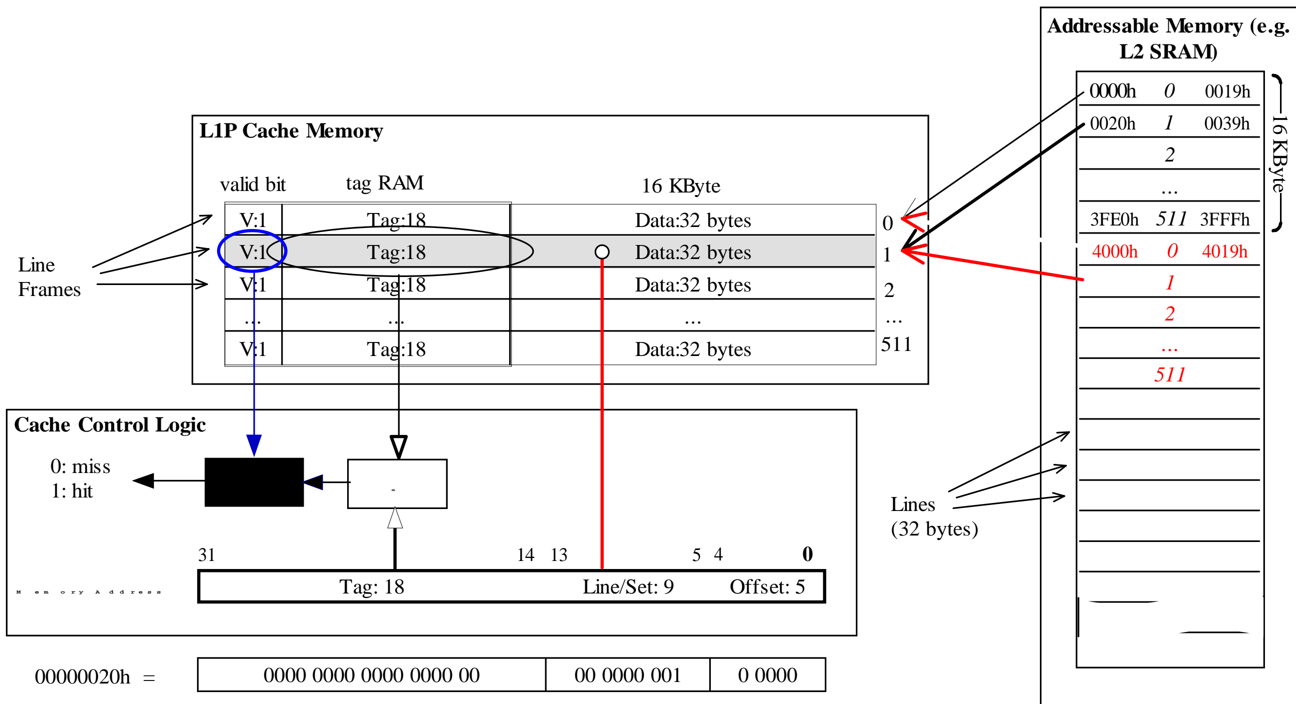
Table 1: L1P characteristics

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

[1] Except for external memory addresses that are marked non-cacheable.

Figure 2 shows the architecture of the C64x L1P which consists of the cache memory and the cache control logic. Additionally, addressable memory (which can be L2 SRAM or external memory) is shown. The cache memory is 16KBytes large and consists of 512 32-byte lines. A line frame consists of a valid bit, the tag and the actual line data. Each line frame always maps to the same fixed addresses in memory. For instance, as shown in Figure 2, addresses 0000h to 0019h (32 bytes) will always be cached in line frame 0 and addresses 3FE0h to 3FFFh will always be cached in line frame 511. Then, since the capacity of the cache has been exhausted, addresses 4000h to 4019h again map to line frame 0, and so forth. Also note that one line contains exactly one instruction fetch packet.

Figure 2: C64x L1P architecture.



2.1.1 Read Misses

Consider a CPU program fetch 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 bit, V. A valid bit of 0 means that the corresponding line frame is invalid, i.e. 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:

- the offset (bits 0-4),
- the set (bits 5-13) and
- the tag (bits 14-31).

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

A miss also means that for the line containing the requested address a line frame will be allocated. The then 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 contains valid cached 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.

2.1.2 Read Hits

Assume now address 0020h is accessed again. The stored tag portion is now be compared against the tag portion of the address requested. This comparison is necessary since multiple lines in memory map to the same set. If we had accessed address 4020h which 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 positive and the valid bit is 1. Thus the controller will register a hit and forward the data in the cache line to the CPU. The access is complete.

2.2 Types of Cache Misses

Before set-associative caches are discussed in the next section 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 re-used the lower the impact of the initial penalty and the shorter the average memory access time will become. The key is to re-use 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.

Evictions are caused by conflicts, i.e. a memory location is accessed that maps to the same set as a memory location that was accessed earlier. This is type of miss is referred to as a *conflict miss,* i.e. a miss that occurred because the line was evicted due to a conflict before it was re-used. It is further distinguished whether the conflict occurred because the capacity of the cache was exhausted or not. If the capacity was exhausted, i.e. 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 re-used. 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. If we have conflict misses that means 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 (i.e. map to the same set) in cache. Alternatively, from a hardware design point of view, 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 basic idea of *set-associative* caches which are described in detail in the next section. 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 point of view, the capacity of the cache can be increased.

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. As opposed to the other two misses above, they cannot be avoided, hence they are compulsory.

2.3 Set-Associative Caches

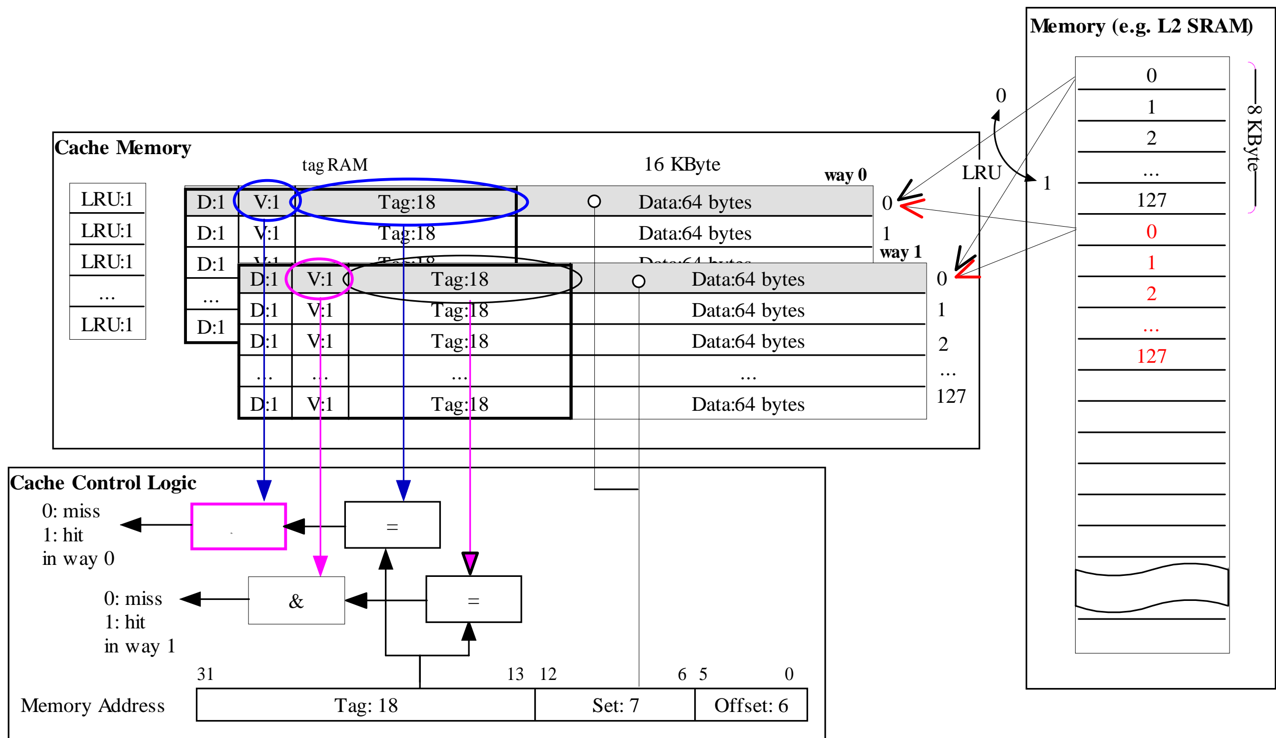
Set-associative caches have multiple so-called *cache ways* to reduce the probability of conflict misses. The C64x L1D is a 2-way set-associative cache with 16Kbytes capacity (8 Kbytes per way) and 64-byte lines. The C621x/C671x L1D is also a 2-way set-associative cache, but with 4Kbytes capacity (2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

Table 2: L1D Characteristics

Characteristic	C64x	C621x/C671x
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	16 Kbytes	4 Kbytes
Line size	64 bytes	32 bytes
Single read miss stall (L2 SRAM)	6 cycles	4 cycles
Single read miss stall (L2 Cache)	8 cycles	4 cycles
Miss pipelining	Yes	No
Multiple consecutive misses (L2 SRAM)	4 + 2 x M cycles	4 cycles
Multiple consecutive misses (L2 Cache)	6 + 2 x M cycles	4 cycles

The difference to a direct-mapped cache is that for a 2-way set-associative cache each set consists of two line frames, one line frame in way 0 and another one in way 1, i.e. 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 set-associative cache.

Figure 3: C64x L1D Architecture.



The set-associative cache architecture shall now be explained in detail by examining how misses and hits are handled for the C64x L1D cache. Its architecture is shown in Figure 3. Hits and misses are determined in the same way as for direct-mapped caches, except that now two tag comparisons are necessary to determine in which way the requested data is kept.

2.3.1 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.

2.3.2 Read Misses

If both ways miss, the data first needs to be fetched from memory and placed in an allocated line frame. 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 (see Figure 3). If it is 0 the line frame in way 0 and if it is 1 the line frame in way 1 is allocated.

The state of the LRU bit changes whenever an access (read or write) 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. Consequently, on a miss the least-recently-used (LRU) line frame in a set will be allocated to the new line evicting the current line. The motivation 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 accessed soon again.

Note again 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.

2.3.3 Write Misses

As L1P, L1D is a *read-allocate* cache, which allocates new data from memory 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).

2.3.4 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 one has to know which line was written to by the CPU. For this purpose every cache line has a *dirty bit* (D) associated with it (see Figure 3). Initially the dirty bit is zero. As soon as the CPU writes to a cached line, the corresponding dirty bit will be 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 (so-called *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 [3,4].

2.4 Level 2 Cache

Until now it was assumed that there is one level of cache memory between the CPU and the 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. Level 2 caches basically operate in the same manner as a level 1 caches; however, they are typically larger in capacity. Level 1 and 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 L2 memory space can be split into an addressable internal memory (L2 SRAM) and a cache (L2 Cache) portion. As opposed to the L1 caches, which 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 cache both L2 SRAM and external memory addresses. L2 Cache characteristics are summarized in Table 3.

Table 3: C6000 L2 Cache Characteristics

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

2.4.1 L2 Cache 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 this line frame contains dirty data it will be first written back to external memory before the new line is fetched. 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.

Note that some external memory addresses may be configured as non-cacheable [2,3]. In this case the requested data is simply forwarded from external memory to the CPU without being stored in any of the caches.

2.4.2 L2 Cache 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 non-cacheable. In this case the data is directly updated in external memory without being stored in cache.

3. Cache Coherence between CPU and DMA Accesses

Generally, cache coherence is relevant in cases where two or more devices (e.g. CPUs, peripherals, DMA controllers) exchange data through a cacheable memory region. A device may have a local copy of a memory location in its own cache. Hence, when data is transferred between devices, it has to be ensured that each device sees the same data as the device it is communicating with.

Suppose the CPU reads a memory location which gets subsequently allocated in cache. Later, a DMA is writing new data to this same memory location which is meant to be read and processed by the CPU. However, since this memory location is kept in cache, the CPU read access will hit in cache and return the old data. A similar problem occurs if the CPU writes to a memory location that is cached, and the data is to be read by a DMA. The data only gets updated in cache but not in memory. When the DMA accesses the memory location it will read an old value instead of the updated one. Whenever the cached data of a memory location is different, cache and memory are said to be incoherent.

C64x and C621x/C671x DSPs automatically maintain cache coherence between CPU and DMA accesses to L2 SRAM through a hardware cache coherence protocol based on snoop commands. The coherence mechanism is activated on a DMA read and write. 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 line containing the address that was written is

invalidated in L1D, so that next time the line is re-allocated from memory when the CPU makes a read request. 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 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.

This coherence mechanism operates fully in the background. The programmer can initiate DMA transfers as if they accessed a flat memory. For instance, a typical DMA double buffering scheme could look like this:

```
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);
}
```

Using DMAs to transfer data to/from internal memory provides the highest application performance. In order to reduce hardware complexity, maintaining coherence for other cases, such as accesses to shared cacheable external memory or self-modifying code, is the responsibility of the programmer. For this purpose the cache controller offers various commands that allow the programmer to manually keep caches coherent.

To maintain coherence between external memory and cache the programmer has to mimic what the cache controller does for L2 SRAM accesses. Whenever an EDMA writes data to an input buffer in L2 SRAM, the cache controller would invalidate the corresponding line in L1 cache. Similarly, here all lines in L1 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 re-allocated these lines from external memory next time the input buffer is read. And accordingly, all lines in L1 and L2 Cache that map to the external memory DMA output buffer have to be written back before the DMA transfer starts. The TI chip support library (CSL) [5] provides a set of routines that allow the programmer to initiate writeback, invalidate and writeback-invalidate operations. They are describe next.

Before the DMA write transfer starts a writeback-invalidate (or alternatively an invalidate on C64x) has to be performed. The start address of the buffer in external memory and number of bytes needs to be specified: CACHE_wbInvL2(InBuffB, BUFSIZE, CACHE_WAIT). For C64x an invalidate only operation is also supported which 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 writeback operation (C621x/C671x and C64x): 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 which still may reside in L1D and L2 Cache.

The following pseudo-code shows exactly in which order the cache coherence calls and the DMA transfers should occur:

```
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);
}
```

Table 4 shows an overview of available L2 cache coherence operations for C621x/C671x and C64x. Note that these operations have no effect if L2 Cache is disabled. The table 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, a snoop-invalidate command to L1D (and invalidate command to L1P if applicable) is issued to make L2 and L1 coherent. Then the appropriate operation is performed on L2 Cache.

Table 4: L2 Cache Coherence Operations.

Scope	Coherence Operation	CSL Command	Operation on L2 Cache	L1D Snoop Commands	L1P Snoop Commands
Block	Invalidate L2 (C64x 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

The following guidelines should be followed for using cache coherence operations. Again, user issued cache coherence operations are only required if CPU and EDMA share a cacheable region of external memory. This means if the CPU reads data written by the DMA and vice versa. Therefore, the safest rule is to issue a *Writeback-Invalidate All L2* operation prior to any DMA transfer to or from external memory. However, the disadvantage is that possibly more cache lines are operated on than it is necessary, causing a larger than necessary cycle overhead. First, it is only required to operate on those cache lines in memory that actually contain the shared buffer. The block cache operations can be used for this purpose. Further, it can be distinguished between the following three scenarios:

Scenario	Use ...
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 is to be read back by the CPU	Writeback-Invalidate before DMA starts

It is important to note that although the CSL routines accept byte aligned start addresses and byte counts, the cache controller always operates on *whole* lines. In fact the cache controller operates on all lines that are "touched" by the specified address range. Therefore, arrays in external memory that are subjected to coherence operations must be

a multiple of cache lines large and

aligned at a cache line boundary.

If this is not ensured, the correct functioning of the application is not guaranteed. Accordingly, the arrays for the above example should be declared as follows:

```
#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];
```

Note that the maximum byte count that can be specified is 2^{18}-1. If the external memory buffer to be operated on is larger, multiple cache operations have to be issued.

4. Cache Optimization

This section now discusses several code and memory optimization methods that improve the efficiency of cache. The focus will be 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 will be used efficiently too. Typically, there is not much benefit in optimizing for L2 Cache only. It is recommended to use L2 Cache for the general purpose processing parts of the application that have largely unpredictable memory access patterns (general control flow, etc.). For time critical signal processing algorithms data should be streamed directly into L2 SRAM using EDMA, and memory accesses should be optimized for L1 cache.

To effectively apply optimizations one has to know the sources of cache-related CPU pipeline stalls. The most common cache stall conditions on C64x devices are described below. More detailed information can be found in [2,3,4].

L1D Read Miss: 6 cycles per miss to perform a line fill from L2 SRAM and 8 cycles per miss to perform a line fill from L2 Cache. L1D read miss stalls can be lengthened by L2 read miss, L2 access conflict, L2 bank conflict and write buffer draining[2]. Consecutive and parallel misses will be overlapped, provided none of the above stall lengthening condition occurs and the two parallel/consecutive misses are not to the same set.

L1D Write Buffer Full: The 4×64-bit write 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 an L2 write miss, L2 access conflict and L2 bank conflict.

L1P Read Miss: 8 cycles per miss to perform a line fill from L2 SRAM or L2 Cache. Can be lengthened by L2 read miss, L2 bank conflict. 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). A snoop access to the L1D tag RAM may the stall the CPU. If the snoop hits in L1D, new CPU requests to L1D are stalled until the

[2] 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 read followed by a write.

snoop operation is complete. If the snoop and the CPU L1D request occur simultaneously, the CPU request is given higher priority.

The C64x cache architecture pipelines read misses allowing parallel and consecutive read miss stall cycles to be overlapped. While for instance a single miss to L2 SRAM causes a 6 cycle stall, multiple parallel and consecutive misses take only 2 cycles once pipelining is set up: $4 + 2 \times M$ cycles for M misses. This mechanism is described in detail in [3]. Note that 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 above. Note also 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, one has 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* [4] which can be used to pre-allocate an array in L1D with causing a minimum number of read miss stall cycles.

There are two important ways to reduce the cache overhead:

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

2) Reduce the number of cache misses (in L1P, L1D and L2 Cache): This can be achieved by

   a) Maximizing cache line re-use:

      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 re-used as often as possible. Either the same data can be re-read or, since L1D is not write allocate, new data written to the already cached locations so that subsequent reads will hit.

   b) Avoiding eviction of a line as long as it is being re-used: An eviction occurs whenever the available number of cache ways has been exceeded.

      i) Thus evictions can be prevented if data is allocated in memory such that the number of cache ways is not exceeded when it is accessed.

      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.

Methods for reducing the number of cache misses and number of stalls per miss will be discussed in detail in this section. A good strategy for optimizing cache performance is to proceed in a top-down fashion, starting on application level, moving to procedural level, and if necessary considering optimizations on algorithmic level. The optimization methods for 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 section reflects the order in which one may want to address the optimizations.

4.1 Application Level Considerations

It may be beneficial to distinguish between DSP-style processing and general purpose processing in an application. Since control and data flow of DSP processing are usually well understood, its code better lends itself to a more careful optimization than general purpose code. Signal processing code and data may therefore benefit from being allocated in L2 SRAM. This reduces cache overhead and gives the programmer more control over memory accesses since only Level 1 cache is involved whose behavior is easier to predict. This allows the programmer 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.

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

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, then it will be cached by L2, then cached by L1D before it finally reaches the CPU.

Cache coherence is automatically maintained by the cache controller, no user action is required. If the buffers are located in external memory the programmer has to take care to maintain coherence by manually issuing L2 cache coherence operations.

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 and the size of the buffers increased accordingly.

For rapid-prototyping applications where implementing DMA double buffering schemes are considered to 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 3, 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.

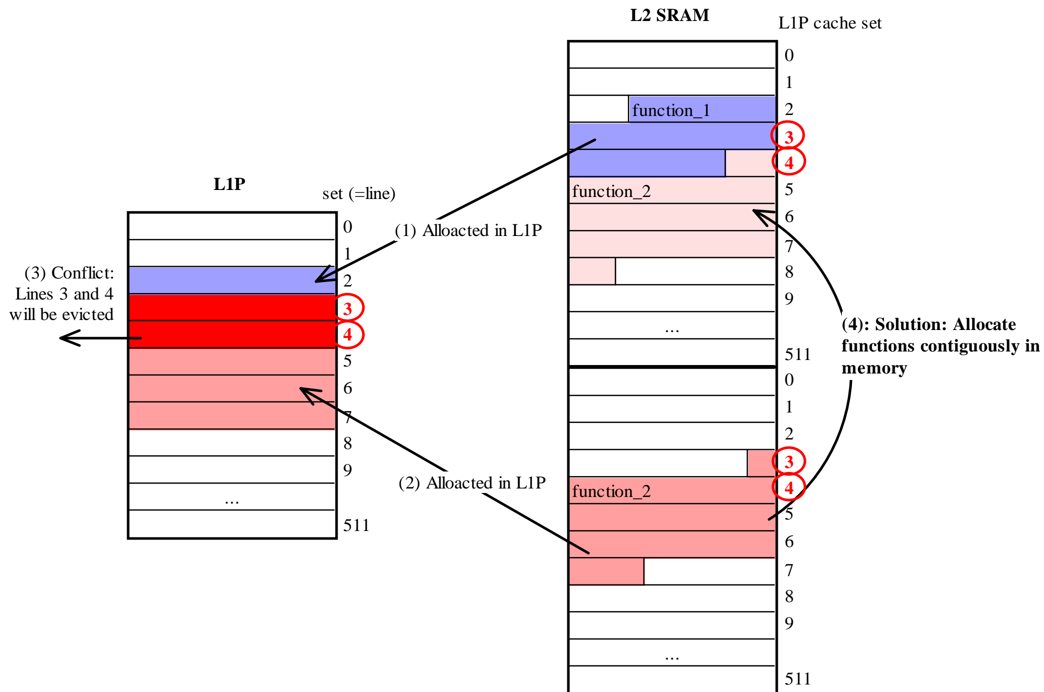
General purpose processing is typically dominated by straight-line code, control flow and conditional branching. This code typically does not exhibit much parallelism, and execution depends on many modes and conditions and thus tends to be largely unpredictable. That is, data memory accesses are mostly 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 whole the application, it is may be best, to allocate general purpose code and associated data in external memory and let L2 Cache 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.

4.2 Contiguous Allocation of Code

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 within the same cache line will evict one another. This section describes how these evictions can be avoided by allocating code contiguously in memory.

Assume two functions, function_1 and function_2, have been placed by the linker such that they overlap in L1P as depicted in Figure 4. When function_1 is called the first time, it will be allocated in L1P causing three compulsory misses (1). A following call to function_2 will cause its code to be allocated in L1P resulting in 5 compulsory misses (2). This also will evict parts of the code of function_1, line 3 and 4, since these lines overlap in L1P (3). When function_1 is called again in the next iteration, two conflict misses occur which bring these lines back into L1P. They are again evicted when function_2 is called. This pattern repeats for all following iterations. Every function call will suffer two conflict misses, totaling four L1P misses per iteration. The conflict misses can be completely avoided by allocating the code of the two functions into non-conflicting sets. The most straightforward way this can be achieved is to place the code of the two functions contiguously in memory (4).

Figure 4: Avoiding L1P Evictions



Contiguous allocation of code on C6x DSPs can be achieved as follows. 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 follows:

```
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
...
}
```

The linker will link all sections in exactly the order specified. 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.

Those functions should be considered for re-ordering 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 re-use without evictions. This may increase the memory requirements for temporary buffers to hold output data.

4.3 Contiguous Allocation of Data

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 addresses that map to the same set would evict one another if they are not contained in the same cache line. However, in the 2-way set-associative L1D, two conflicting lines can be kept in cache without causing evictions. Only if

another, third memory location is allocated which 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).

Optimization methods similar to the ones described for L1P in the previous section can be applied to data arrays. However, the difference between code and data is that L1D is a 2-way set-associative cache, whereas 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 routine dotprod which computes the dot product of two input vectors:

```
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, i.e. in1 × w1, in2 × w2, in1 × w2, and in2 × w1. We could use the following call sequence of dotprod to achieve this:

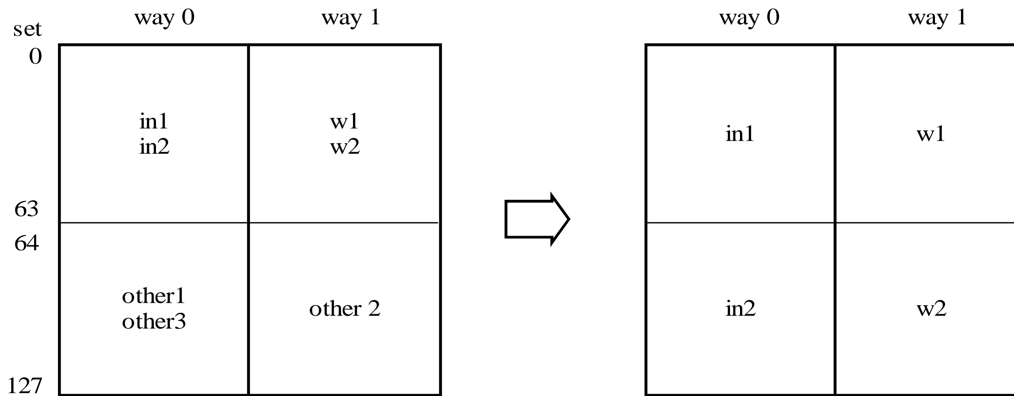
```
r1 = dotprod(in1, w1, N);
r2 = dotprod(in2, w2, N);
r3 = dotprod(in1, w2, N);
r4 = dotprod(in2, w1, N);
```

Further assume that each array is one fourth of the total L1D capacity, such that all 4 arrays fit into L1D. However, assume that we have given no consideration to memory layout and declared the arrays as follows:

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

Where other1, other2 and other3 are some arrays used by other routines in the same application. It shall be assumed here 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 (the size of one way is 8 Kbytes for C64x) map to the same set. In this case this means that, in1, in2, w1 and w2 all map to the same sets in L1D. One possible layout for L1D is shown on the left in Figure 5. Note, that this is only one possible configuration of many. The exact configuration will depend on the start address of the first array, in1, and the state of the LRU bit (which decides in which way the line is allocated). However, all configurations are equivalent in terms of cache performance.

Figure 5: Mapping of Arrays to L1D Sets for Dot Product Example



The first call to dotprod will bring in1 and w1 into L1D, as shown in the left diagram in Figure 5. This will cause 128 compulsory misses. The second call will cause in1 and w1 to be evicted and replaced with in2 and w2, which causes another 128 misses. The third call re-uses w2, but replaces in2 with in1 resulting in 64 misses. Finally, the last call again causes 128 misses, because in1 and w2 are replaced with in2 and w1. Table 5 shows the actual stall cycles for C64x. We expected $3\times128+64=448$ 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).

Of the total number of misses, 256 are compulsory (first reference) and 192 are conflict misses. To eliminate the conflict misses, we 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 in the right diagram in Figure 5. Note that due to the memory allocation rules of the linker it cannot always be guaranteed that consecutive definitions of arrays are allocated contiguously in the same section (e.g. 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 [8], 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 optimized cycle counts are shown in Table 5. Now data is heavily re-used from cache using the new memory configuration. As expected we see about 256 misses. The C64x L1D stall cycles were reduced from 2,730 to 1,560, a 43% reduction.

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

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

Exploiting miss pipelining on C64x devices can further reduce the cache miss stalls. The touch loop can be used to pre-allocate all arrays, in1, in2, w1 and w2, into 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 for using the touch routine are shown in Table 5. The L1D stalls cycles have further reduced from 1,560 to 544 cycles, a total reduction of 80%. Note that as expected the number of misses is the same as before but the average stall penalty per miss is lower.

Table 5: C64x Stall Cycles for Dot Product Example Before Optimization

	Original	**Optimized**	**Optimized with touch**
Execute Cycles	2,152	2,152	2,296
L1D Stalls (Read Misses)	2,730 (455)	1,560 (260)	544 (260)
L1P Stalls	56	56	64
Total	4,400	3,278	2,904

4.4 Array Padding

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:

```
int w_dotprod(const short *restrict w, const short *restrict x, const short *restrict h, int nx)
{
   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<nx; 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 will be accessed and cause three read misses to the same set. The third read miss will evict 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] will cause a read miss, evicting the line of x. Next, x[1] is accessed which just was 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 re-used, every single read access in the routine will cause a read miss. The contents of the L1D set at the times when an access is made is illustrated in Table 6. It can be seen that whenever an array element is attempted to be read, it is not contained in L1D.

Table 6: Contents of an L1D Set at the Time when an Array is Accessed (Weighted Dot Product Example)

Way 0	Way 1	LRU	Read Access to
		0	w[0]
w		1	x[0]
w	x	0	h[0]
h	x	1	w[1]
h	w	0	x[1]
x	w	1	h[1]

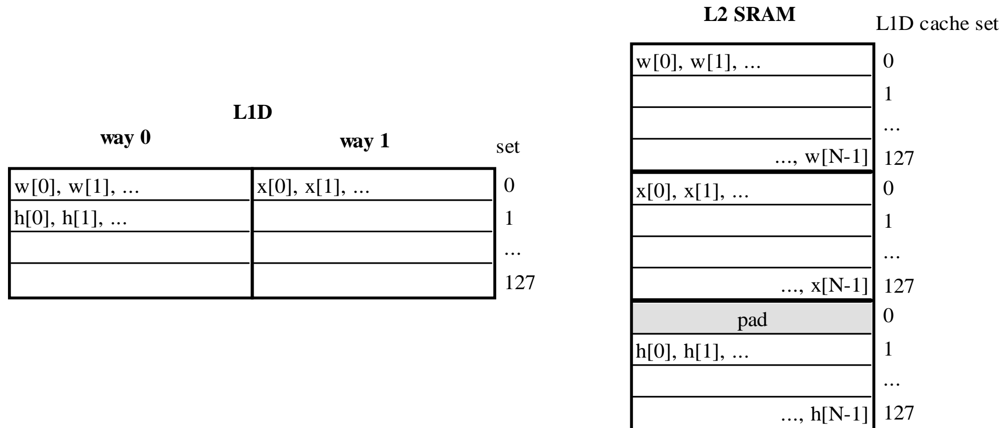
The read miss stalls caused are shown in Table 7. In this case the array size was chosen to be 4,096, i.e. each array is 8,192 Kbytes which is half the L1D capacity and places the start of each array at the same set. We expect to see 3 × 4096 read misses, one for each element access. However, the number of measured read misses is much smaller, 2817 misses only. 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 re-use. Still, the number of read misses without thrashing should be 4096 elements × 2 bytes each × 3 arrays / 64 bytes line size = 384 read misses.

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 illustrated in Figure 6. 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 then since all data in the line has been used and will not be accessed again.

Figure 6: Memory Layout and Contents of L1D after the First Two Iterations



The cycle counts for the padded memory layout is shown in Table 7. L1D conflict misses were completely eliminated, hence the number of L1D read misses matches now the expected ones.

Table 7: Stall Cycles for Weighted Dot Product Example

	Original	**Padded**
Execute Cycles	3,098	3,098
L1D Stalls (Read Misses)	20,485 (2,817)	2,433 (385)
L1P Stalls	33	40
Total	23,616	5,571

4.5 Blocking

If the data set is too large to fit into cache and lines are evicted due to capacity misses before they can be re-used, data can be split up into smaller portions that fit into cache and operated on one at a time. This technique is referred to as blocking or tiling [1]. Consider the dot product routine which 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 C64x L1D capacity (N=16,384). 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 re-use 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 again to avoid eviction of a cache line before it is re-used. We would like to re-use the array w. The first line of w will be the first one to be evicted when the cache capacity is exhausted. In this example the cache capacity is exhausted after N/4 outputs have been computed, since this required N/4 × 2 arrays = N/2 array elements to be allocated in L1D. If we stop processing in1 at this point, and start processing in2, we can re-use 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 re-structured 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);
}
```

The cycle counts are shown in Table 8. The number of elements per vector N was set to 16,384. For the original code we expect to see about 16,384 elements × 2 bytes per element × 2 arrays / 64 bytes per cache line = 1,024 read misses to occur per function call, i.e. 4,096 read misses which closes matches the 4,100 misses measured. The size of each array is 32 Kbytes, twice the capacity of L1D. The total amount of data that will be 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 re-allocated w again.

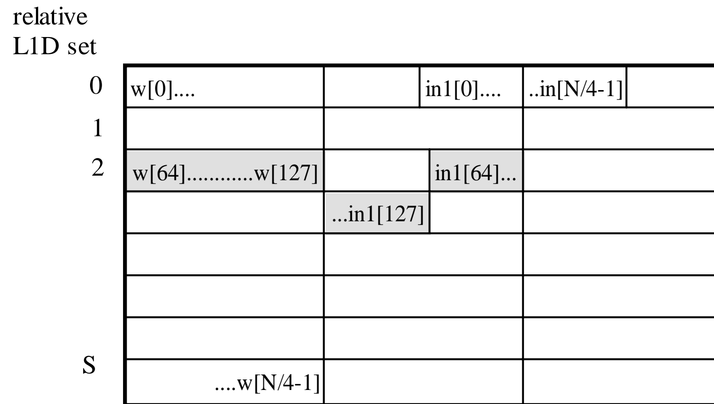
Through blocking we expect to save the capacity misses for the array w, which are 16,384 elements × 2 bytes × 3 arrays / 64 bytes per line = 1,536, i.e. we expect 4,096 - 1,536 = 2,560 read misses. The actual cycle counts are shown in Table 8. Read miss stalls were reduced by 23% from 24,704 to 15,528.

We can further reduce the number of read miss stalls by exploiting miss pipelining. The touch loop is used to allocate the required subset of 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[3]. This LRU behavior cannot be guaranteed if the access order changes. Example: If after dotprod array w is LRU and array in1 is MRU (most recently used), w was accessed before in1. If the next dotprod accesses w first again, the access will hit and the line of w becomes MRU and is protected from eviction. However, if now the touch loop is used, in1 is accessed before w1. Accesses to in1 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 in1 at the same set and within this set placing the start of w ahead of in1 as illustrated in Figure 7. Consider processing of elements 64 to 127 as highlighted in Figure 7. When element 127 of both w and in1 is accessed, in1[127] is one line ahead, leaving the line of w in set 2 most recently used. Consequent all lines of w will become MRU and will not be evicted.

Figure 7: Memory Layout for Dotprod with Touch Example



In this example arrays w and in1 should anyway be aligned to different memory banks to avoid bank conflicts. If arrays w and in1 are aligned to the same set first and in1 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)
```

[3] 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, first way 0 the way 1.

```
short in1[N];
short in2[N];
short in3[N];
short in4[N];
```

The touch loop is called 5 times per iteration. For M read misses each touch loop requires (2.5 × M + 16) cycles inclusive read miss stalls, i.e. in this case 2.5 × N/4 × sizeof(short) / 64 + 16 = 336 cycles. The routine dotprod itself then will not suffer any read miss stalls. The results with blocking and touch are shown in Table 8. This brought the total cycle count down to 24,100. Compared to the original cycle count of 41,253, this is a reduction of 42%.

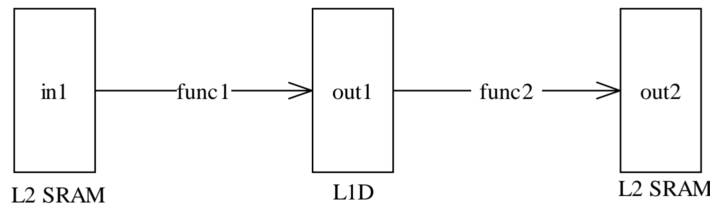
Table 8: Benchmark Cycles for Dot Product Example

	Original	**With Blocking**	**With Blocking and Touch**
Execute Cycles	16,494	16,768	18,444
L1D Stalls (Read Misses)	24,704 (4,100)	15,528 (2,588)	5,404 (2,596)
L1P Stalls	55	228	252
Total	41,253	32,540	24,100

4.6 Function Chaining

Often the results of one algorithm feed as input into the next algorithm forming a chain. If this processing chain is executed repeatedly and the intermediate buffers holding the results can be kept in cache across iterations, only the first algorithm of the chain will suffer read misses to allocate the data from memory. All following algorithms will not suffer any read misses (except for the first iteration during which the buffer is not yet allocated in cache). Also, all algorithms except the last one in the chain which writes its output through the write buffer to memory, will not see any write buffer related stalls (write buffer full or write buffer drain). The concept is illustrated in Figure 8.

Figure 8: Processing Chain With 2 Functions



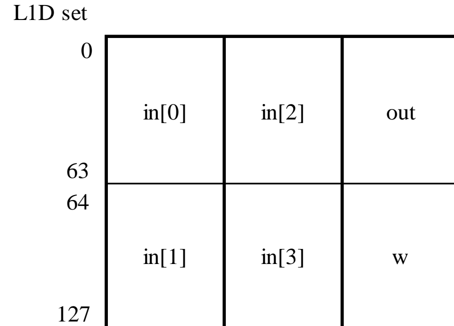
Consider the following example code of a 4-channel filter system consisting of a 64-tap FIR filter followed by a dot product:

```
#define NX  NR+NH-1
short in1[4][NX]; /* input samples           */
short out1  [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(in1[i], h[i], out1, NR, NH);
  out2 = dotprod(out1, w, NR);
}
```

The FIR filter in the first iteration will allocate in1 and h in L1D and write out1 to L2 SRAM. Subsequently out1 and w will be allocated in L1D by the dot product routine. For the next iteration then, the FIR routine will write its results to L1D rather than L2 SRAM and the function dotprod does not incur any read misses. However, the buffers have to be allocated such that out1 and w will not be evicted from cache. A possible memory layout for NX=2,047 (NR=NX-NH+1=1984) is shown in Figure 9. Note that we can neglect the coefficient array h here since it occupies only 4 channels × 64 taps × 16 bits / 64 bytes per cache line = 8 cache lines in total. It can be seen that within one iteration no more than two arrays map to the same sets,

i.e. no conflict misses will occur. Capacity misses will not occur either since the total size of the data set accessed within one iteration fits into L1D. If this had not been the case the arrays would have had to be split up using the blocking technique.

Figure 9: Memory Layout for Processing Chain



The benchmark cycle counts are presented in Table 9. The FIR filter is taken from the TI C64x DSPLIB [7] (DSP_fir_r8) and takes NR × NH / 4 + 17 cycles to execute, i.e. 31,761 cycles. The FIR filter is accessing NX + NH data elements, i.e. 2,111 16-bit elements or 4,222 bytes spanning 66 lines. Accordingly we see around 66 L1D read misses for FIR. The dot product routine is accessing 2 × NR × 16 bits = 7,936 bytes spanning 124 cache lines. As expected miss stalls for this routine occur only during the first iteration (except possible spurious stack accesses). Further, the two functions were allocated contiguously in memory so that L1P misses occur only the first time.

Table 9: Benchmark Cycles for FIR/Dot Product Example

	1st Iteration		2nd-4th Iteration		Total
	fir	dotprod	fir	dotprod	
Execute Cycles	31,766	520	31,766	520	64,572
L1D Stalls (Read Misses)	396 (67)	719 (124)	396 (66)	6 (1)	2,327 (390)
L1P Stalls	58	53	0	0	111
Total	32,220	1,292	32,160	526	66,198

Note that if the cache overhead (= cache stalls / execute cycles) of the FIR filter and the dot product is determined in isolation the FIR filter has a mere 1.4% cache overhead, whereas the dot product has an 148% overhead. Even if it is taken into account that all following calls have no overhead at all, the average overhead would still be 37%. However, the combined cache overhead of the *processing chain* is only 1.9%. Thus the cache overhead of individual functions can be very misleading and is not indicative of the combined cache overhead of an entire application. When a function is benchmarked one has to consider the function in the context of the whole application.

5. Conclusion

This paper explained in detail common types of caches, their characteristics and their operation. This understanding is required for successfully using cache coherence operations and applying cache optimization techniques. Subsequently the issue of cache coherence and the cache coherence protocol of C6x1x DSPs were explained. Simple guidelines were established for manual issuance of coherence operations. This is primarily necessary for maintaining coherence between CPU and DMA accesses to external memory.

The section on cache optimization discussed the benefits of distinguishing between signal processing and general purpose code. The use of DMA buffering schemes allows fast access to data and provides opportunities to efficiently apply cache optimization methods. On the other hand, general purpose code can be handled well by a multi-level cache which at the same time provides ease of programming. Finally, several optimization methods were presented to eliminate conflict and capacity misses, and minimize the impact of compulsory misses. It was shown that proper design of processing chains can eliminate all cache-related overhead, except for the compulsory misses of the first function. This also highlighted the fact that benchmarks

of individual algorithms in the presence of cache can be misleading. Performance of algorithms should therefore be always evaluated in the context of an entire application.

6. References

[1] Hennessy, J.L. and Patterson, D.A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, San Francisco CA, 1996.

[2] Texas Instruments. TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide (SPRU609), 2002.

[3] Texas Instruments. TMS320C64x DSP Two-Level Internal Memory Reference Guide (SPRU610), 2002.

[4] Texas Instruments. TMS320C6000 Cache User's Guide, 2003.

[5] Texas Instruments. Chip Support Library (SPRC090).

[6] Texas Instruments. TMS320C64x Image and Video Processing Library (SPRC094).

[7] Texas Instruments. TMS320C64x Digital Signal Processing Library (SPRC092).

[8] Texas Instruments. TMS320C6000 Optimizing C Compiler User's Guide (SPRU187)