Freescale Semiconductor
Application Note

Document Number: AN4549
Rev. B, 8/2012

SC3900 FVP Cluster Cache Optimization in the QorIQ Qonverge B4 Series

Includes the L1 I-Cache, L1 D-Cache, and L2 Unified Cache

by Freescale Semiconductor, Inc.

This application note provides guidelines to optimize the use of the L1 instruction cache (I-Cache) and L1 data cache (D-Cache) in the SC3900 FVP Cluster, and L2 unified shared cache in the DSP cluster used in the B4xxx. The guidelines provide SW code development techniques that result in more efficient DSP cache architecture operation. Programmers can use these techniques to modify existing projects for better utilization of the SC3900 core & Cluster in the B4xxx. This document describes the “what,” but programmers may need to consult the core or caches reference manuals for the complete technical descriptions of the “how” (can be found in “SC3900 Subsystem RM”).

To ensure that any given technique is beneficial, it is recommended that trial runs be made with the DTU (Debug and Trace Unit) to profile the performance trade-offs.

Contents

1 Cache memory

The environment for the most efficient data processing model has an unlimited amount of fast memory connected to the system processor with no delays (latency) associated with data or instruction code transfer. The practical and economic alternative to this ideal situation is the use of a memory hierarchy scheme that takes advantage of the relative cost/performance ratios for different memory technologies. The design of such a memory hierarchy depends on the principle of locality. Locality means that after memory is accessed, the same data and its surroundings are likely to be used again shortly.

There are two types of locality: spatial locality (position-based locality) and temporal locality (time-based locality). Spatial locality means that when a certain address is accessed, its surrounding addresses are likely to be accessed shortly. Temporal locality means that when a specific address is accessed, the same address is likely to be accessed again shortly. The principle of locality allows a designer to create a memory hierarchy using memories of different speeds and sizes. Because there is a correlation between faster memory and higher cost, the designer can organize the memory hierarchy into several levels, using the fastest and most expensive memory carefully to support finite areas with a high degree of repeated access and slower, larger areas of memory to support project areas that have less frequent access likelihood. This method optimizes the cost/benefits of each memory type used in a system.

Due to the implications of the locality principle, cache memory is the best way to handle the performance gap between memory and processor. When properly implemented, cache access time can be significantly faster than access time to memory through a bus outside the processor but within an integrated device. Thus, it can reduce the overall access time. A cache also reduces the number of accesses to off-platform memory, which is important for systems with multiple bus masters that share the same memory. An efficient cache reduces external bus cycles and enhances overall system performance.

1.1 SC3900 cluster cache hierarchy

The SC3900 Cluster contains an L1 instruction cache, a read-only L1 data cache with Store Gather Buffer (SGB) and an L2 unified shared cache. The caches are organized in a hierarchy of two levels, as shown in Figure 1. The L1 caches are 8-way set associative, with 128 bytes per line and support unaligned accesses without penalty (unless crossing 4KB boundary). The L2 cache is 16-way set associative, with 64 bytes per line, organized in 4 banks. All the caches are physically mapped. Valid resolution is entire cache line in all the caches. Both L1 caches are Read-Only caches. Writes are temporally stored in the Store Gather Buffer until written to the L2 cache.

The SGB is a data storage that contains 16 entries of 64 bytes each to hold the last written data. All core's write accesses pass through the SGB on the way to memory. Memory writes are merged if possible to a single entry to save memory bandwidth. Accesses pass through the SGB in order (FIFO). Read after write hazards are resolved without penalty, once data is in the SGB (4 cycles after write execution).

The L2 Cache is composed of four independent banks that operate in parallel to provide high data bandwidth. The bus arbiter forwards requests from the cores and accelerator ports to the appropriate bank based on the request address, using a bank hashing function. Bank hashing takes the physical address of a request, and chooses the correct bank for a transaction, using either a basic address interleaving mapping which alternates banks every 64B in linear address space or a special XOR hashing algorithms that can provide a benefit in some cases. In most use-cases, the basic interleaving mapping is sufficient. For more details, please refer to the SC3900 Cluster RM, L2 Cache chapter.

Different memory segments require different cache parameters. To achieve this flexibility, the cache parameters are configured per memory segment descriptor in the memory management unit (MMU). A first core access to an address that is defined in the MMU as cacheable for L1 and L2 causes a line allocation in one of the L1 caches, according to whether the access type is code or data, and a line allocation in the L2 cache. Data thrashes from the L1 cache due to line replacement will not necessarily be thrashed from the L2 cache, creating a longer-term, low-latency storage.

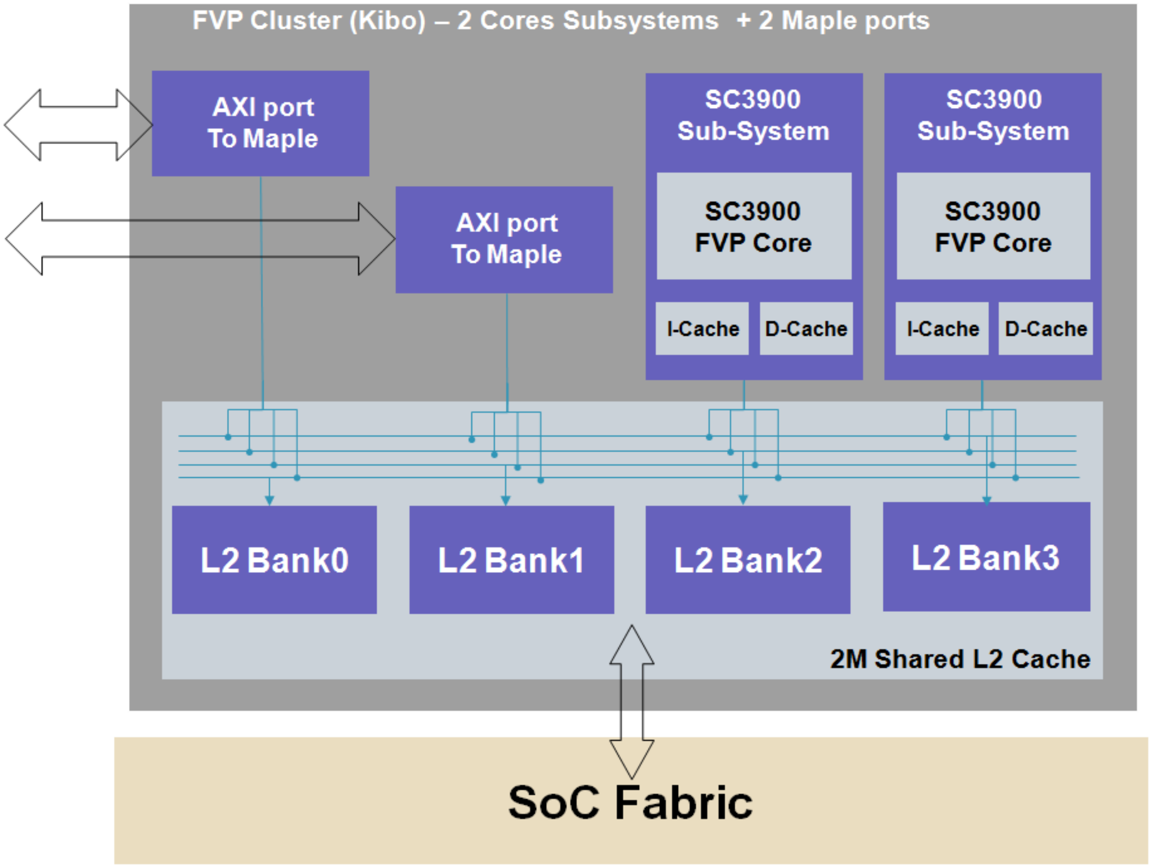


Figure 1. SC3900 FVP cluster cache hierarchy

1.2 Cache misses

The purpose of a cache is to reduce the average memory access time by storing data or code when the processor requests it. If the data is not in the cache when the processor requests it, the request is called a cache miss. For each miss, there is a penalty for fetching a line of data or code 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. The key is to reuse this line as much as possible before it is replaced with another line.

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

1.3 Cache architecture and terminology

Caches store data and code in cache lines organized into ways. The L1 D-Cache and the L1 I-Cache in the SC3900 Cluster each have 8 ways, and they are 32 Kbytes in size (each way is 4 Kbytes). The L2 cache has 16 ways, and is 2Mbytes in size (each way is 128 Kbytes). Each way contains room for several cache lines (up to 32 for L1 caches and up to 2048 cache lines for the L2 cache). Each of the lines inside a way belongs to a different set index.

Figure 2 shows how the L1 D-Cache is organized.

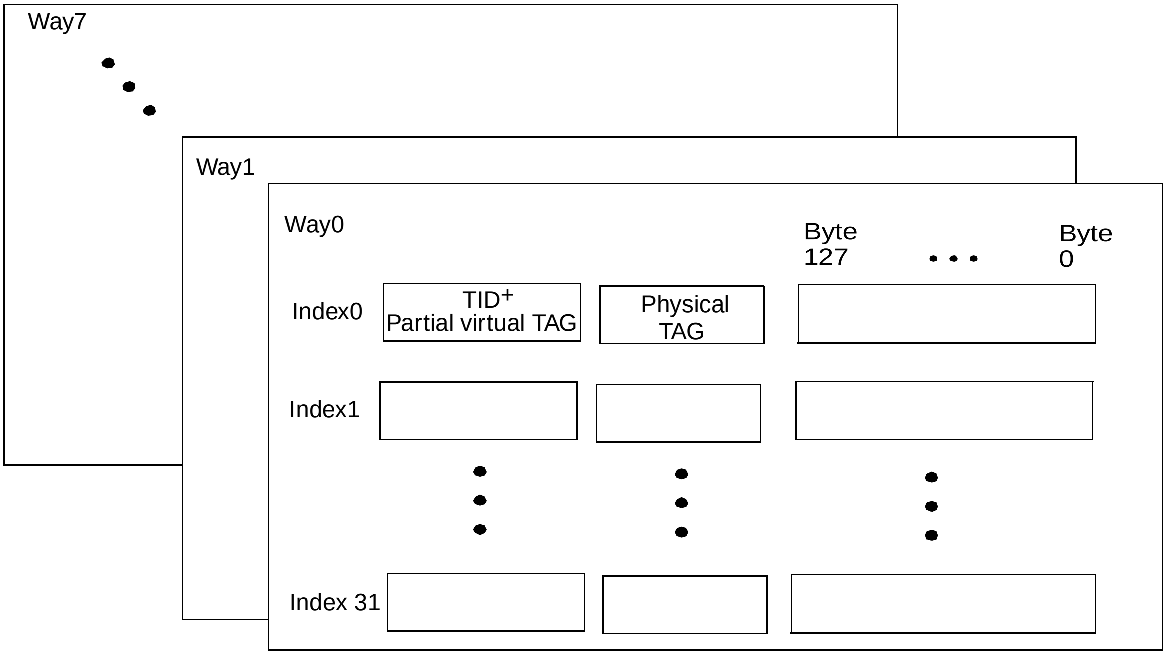


Figure 2. L1 D-Cache organization

1.3.1 Hardware line prefetch

Upon a miss access to the L1 cache, the whole cache line is allocated unconditionally (line prefetch), with 2 requests of 64B each (aligned). The first request contains the miss start address (Critical word first). The second request, containing the rest of the cache line, is fetched when there is no other higher priority miss waiting. Figure 3 shows the line prefetch behavior.

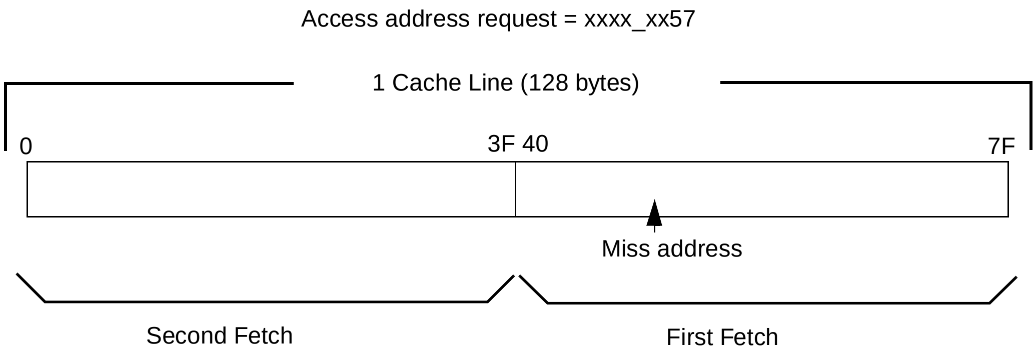


Figure 3. Example of a fetch order on a cache miss

1.3.2 Hardware next line prefetch

L1 cache implements also hardware next line prefetch by automatically generating a tag look up for the following cache line (if next line is not crossing 4 Kbyte boundary). When Next Line Prefetch is enabled for an MMU segment (in the corresponding MMU descriptor) and the next line is a miss, it is loaded to the cache (depends on the MMU setting). The hardware prefetch is performed in the background and does not stall the core. Thus each core read access can be configured to cause automatic (hardware) prefetch of up to two lines (256 bytes). The next line prefetch hardware mechanism can be triggered on any access to a cache line (hit or miss), or just if the core access generated a miss (depends on the MMU descriptor setting). Prefetch accesses has lower priority than core misses and are not always pursued in high miss rate periods, thus ensuring minimal interference.

1.4 Address translation

Address translation is a hardware mechanism that separates the addresses that the program uses to access memory (effective address) from the actual (physical) addresses. Any 32-bit effective address generated by the core is translated to a 36-bit physical address in the background. Up to 64 Gbytes of physical address space is supported.

The following figure depicts the translation operation.

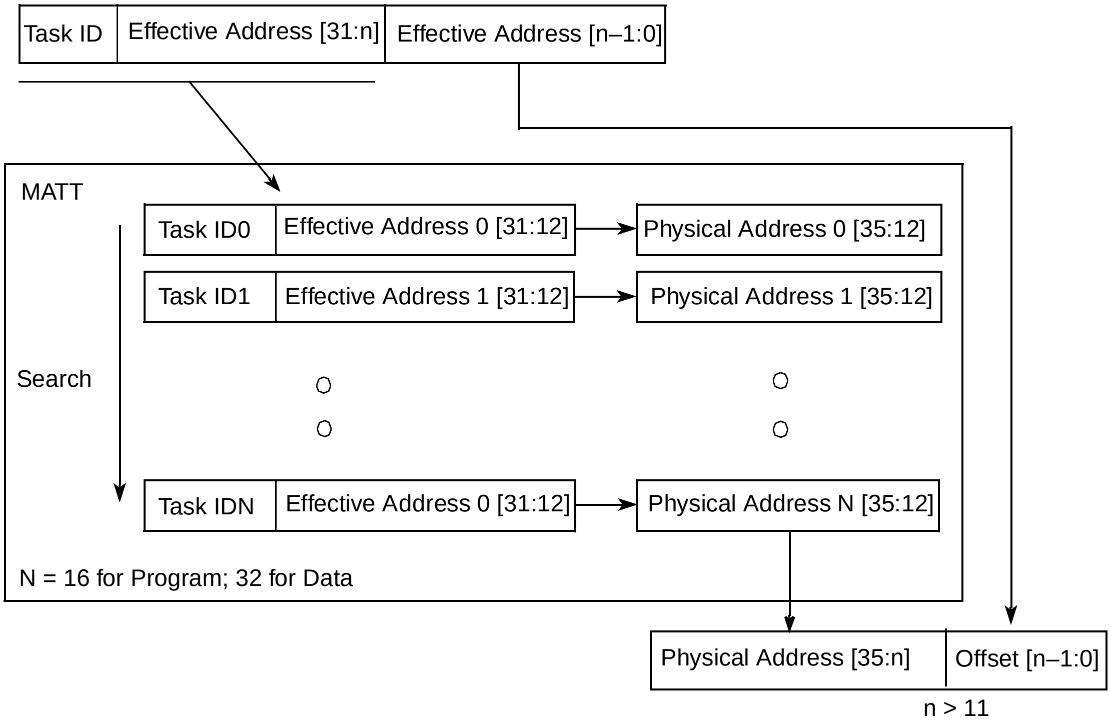

Task ID
Effective Address [31:n]
Effective Address [n–1:0]
MATT
Search
Task ID0
Effective Address 0 [31:12]
Physical Address 0 [35:12]
Task ID1
Effective Address 1 [31:12]
Physical Address 1 [35:12]
Task IDN
Effective Address 0 [31:12]
Physical Address N [35:12]
N = 16 for Program; 32 for Data
Physical Address [35:n]
Offset [n–1:0]
n > 11

Figure 4. Address translation

Translation is performed only on the most significant part (MSP) of the address. The least significant part (LSP) remains unchanged. Translation is defined in the MMU per address range (memory segment). For each access, a set of comparators compares the virtual address with program or data ID and the virtual address of the enabled segments. When there is a hit, the matching translated MSP is driven in its place.

Address translation simplifies software development as follows:

- The program can be written without consideration of the physical memory partition and allocation policy of the system, thereby allowing reuse across systems and cleaner code.
- Relocatable programs can be written without affecting the way the code is written and its cycle performance.
- Use of the lower 64-Kbyte addresses can result in code that is more compactly encoded. With address translation, all tasks can reuse this virtual address space and enjoy the code size reduction.

1.4.1 Task ID and address structure

The SC3900 has a task ID register used to identify the current task. This register includes two separate IDs for program and data, as follows:

- Program task ID (PID)
- Data task ID (DID)

The size of the PID and DID fields in the task ID registers is 8 bits each, which allows up to 255 program and data concurrent tasks. These IDs are used as address space identifiers. PID and DID are part of descriptor entry. This enables descriptors from different tasks to be enabled at the same time without risk of receiving a multi-hit error. The IDs are also partially stored as part of the cache tags. The task ID is used in the caches for performance reasons and not for logical as the physical tag match is checked as well. Tasks that share memory space should have an identical task ID that the cache stores and contrarily, tasks that do not share memory space should have different task IDs. The proposed task ID allocation is not required, but it is recommended for optimal cache operation.

An address used by the software has three representations in different components of the subsystem, as follows:

Effective address — The address represented in the program and issued on the address buses of the core.

Virtual address — The program or data virtual address, appended with the PID or DID, respectively. The virtual address is used inside the caches, in the MMU for descriptor match calculation, and in some communication pathways between the caches and their supporting logic. In general, virtual addresses are hidden from the software and the RTOS.

Physical address — The virtual address after translation, which is the address on the subsystem external buses. The RTOS is aware of the physical addressing of tasks through the programming of the MMU translation tables.

The following figure shows the schematic address structure.

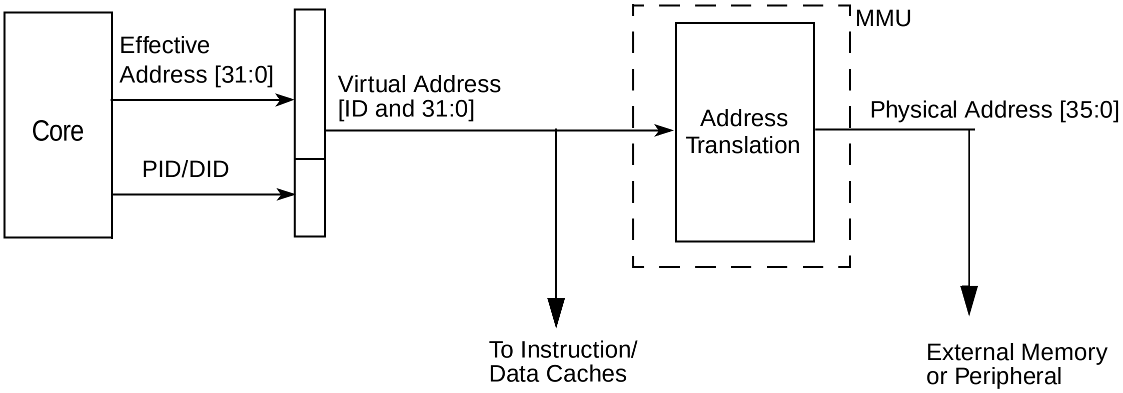


Figure 5. Effective, virtual, and physical addresses

Effective addresses can be reused among tasks. Extending the cache tags with a PID/DID enables this reuse without flushing the cache during each task switch. Thus, the benefits of using the caches are maintained in real-time, multitasked software environments. Each descriptor also has a TaskID field that allows overlapped (effective address) descriptors from different tasks to be enabled at the same time, removing the need of MATT reconfiguration during task switch.

1.4.2 Basic mapping into the cache

In the SC3900 FVP Cluster, the L1 caches are 8-way set associative and the L2 cache is 16-way set associative. For every address, there are 8/16 optional locations, one location in each way. This location is derived from the data or code address as follows:

1. The set index is derived from the address.
2. In L1 caches, the address virtual TAG is compared to the virtual TAG in each way. If it is not equal to any of the set index TAGs, a cache miss is declared (that is, the required data is not in the cache), but if one of the TAGs is equal, it is a virtual TAG hit. Since the caches are physically mapped, the physical TAG match is checked as well. In L2 cache there is only physical TAGs comparison.

   NOTE

   The virtual TAG is shortened by 7MSBs of the virtual address and the 4 MSBs of the task ID in order to optimize power and area. There is a possibility for a virtual TAG hit, but a physical TAG miss due to aliasing on 32MB granularity. This relatively rare scenario results is a full core pipeline flush (13 cycles) and can be avoided totally by thoughtful allocation of the data/program in the virtual memory in such a way that data/program for a certain function doesn't occupy two 32MB windows simultaneously.

3. If the valid bit corresponding to this cache line is set, the line is valid and it is a cache hit (the required data or code is in the cache). Otherwise, it is a cache miss.

1.5 Processing a cache miss

The three caches in the SC3900 FVP Cluster have a mechanism that, upon a cache miss, causes the following to occur:

1. One of the lines in the corresponding set index is evicted according to the replacement algorithm. (In the L2 cache a vacant line will be chosen in higher priority.)
2. In L2 Cache, if the evicted line is “dirty”, it is written back to memory.
3. The new cache line(s) is fetched as described in Section 1.3.1 and in Section 1.3.2.

1.6 Types of cache misses

There are three types of cache misses: conflict miss, capacity miss, and compulsory miss.

To understand conflict and capacity misses, consider a scenario where line A is fetched into the cache, and the set index corresponding to line A is not free in any of the cache ways. Another line, line B, is evicted from this set index. Line B was in the cache because it was accessed before, but it generates a cache miss the next time line B is accessed.

If the cache is not full when line A is fetched into the cache, but the set index corresponding to line A is full, there is a conflict between line A and line B. Therefore, the miss on line B is called a conflict miss. If the cache is full when line A is fetched into the cache, the miss on line B is called a capacity miss.

Compulsory misses, or first reference misses, occur when the data is brought into the cache for the first time. Using a prefetch mechanism can reduce the amount of compulsory misses because data regions are fetched before they are accessed.

1.7 Basic concepts for working with caches

Cache friendly program design should optimize use of the prefetch mechanism by storing the data or code continuously in the order in which it is used. If a cache miss occurs, the next logical accesses are loaded to the cache by the prefetch mechanism. For example, when scanning a matrix, it is better to scan it along its rows (one by one) than to scan it along its columns, as shown in the following comparison.

Version A performs a scan along the rows (see Figure 6):

```
For (row=0; row<64; ++row)
         For (col=0; col<64; ++col)
                  Matrix[row,col]*=2;.
```

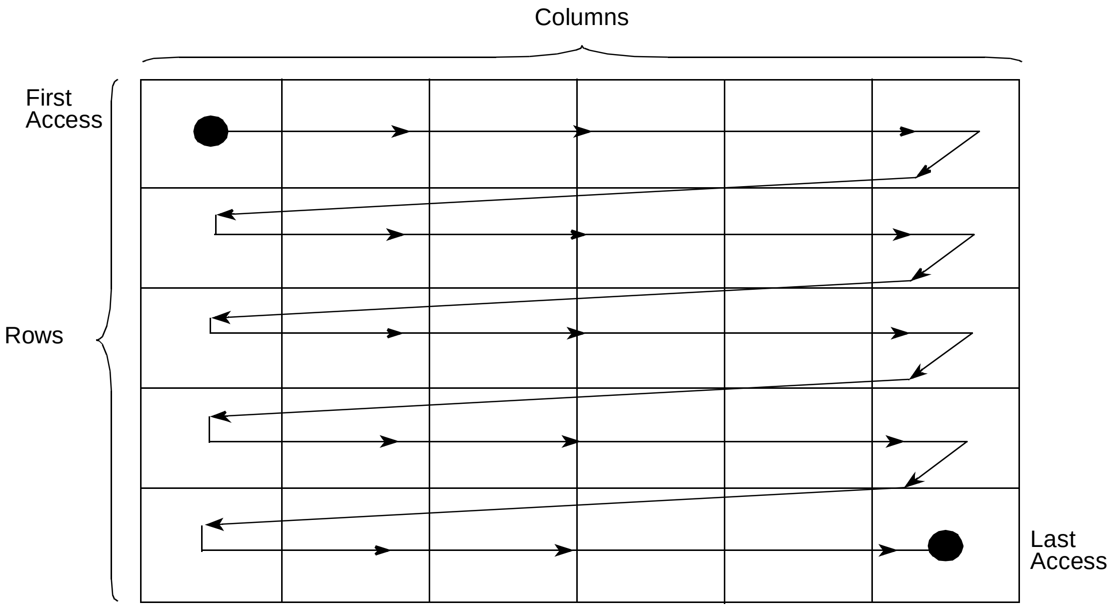


Figure 6. Matrix: Access along the rows (version A)

Version B performs a scan along the columns (see Figure 7):

```
For (col=0; col<64; ++col)
         For (row=0; row<64; ++row)
                  Matrix[row,col]*=2;
```

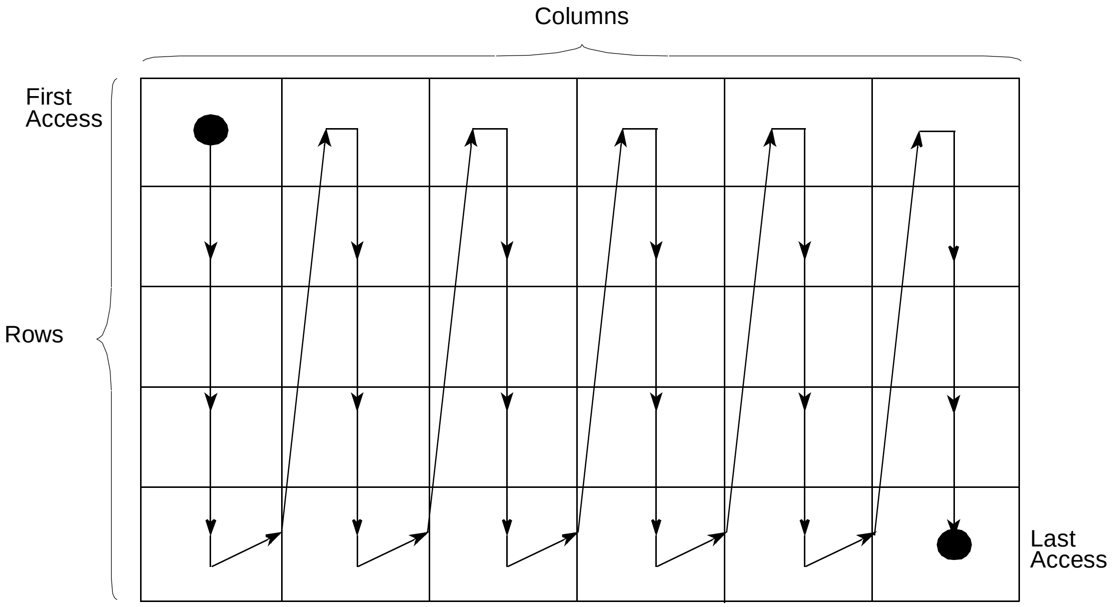


Figure 7. Matrix: Access along the columns (version B)

The cache performance is better in Version A than in Version B. The only cache misses are compulsory misses (one per cache line, or less if next line prefetch is enabled)

1.8 Cache performance

Cache performance is affected by two main factors:

- Number of misses
- Miss penalty. The average number of core stalls caused by a cache miss.

Total cache degradation is the product of these two numbers, as shown in the following equation:

Total Degradation (cycles) = Number of Misses × Miss Penalty ***Eqn. 1***

To improve the overall cache performance, we must reduce both the number of misses and the miss penalties. This is the basic goal of cache optimization.

1.9 Data coherency

Data coherency in the SC3900 FVP Cluster is maintained by hardware. The SC3900 FVP Cluster includes cache management commands to help the user speedup cache de/allocation management. SW Coherency management is outside the scope of this document.

2 Optimization techniques

This section discusses ways to improve the performance of the SC3900 FVP Cluster caches. It describes the elementary cache configuration, primary optimization techniques, and complex secondary optimization techniques with the following two exceptions:

- Code size reduction. Although it is easy to implement, it is included with the secondary optimization techniques because its effect on performance is not always significant.
- Algorithmic-level optimization of the I-Cache. Although it can significantly affect performance, it is included with the secondary optimization techniques because it is not easy to implement.

The first, basic step for achieving good performance is to follow the elementary cache configuration section.

The primary optimization techniques are easy to implement and significantly influence cache performance (improvement can be about 50 percent of the cache degradation). Usually, the primary techniques are sufficient to achieve good cache performance.

If further improvement of the cache performance is required, examine the secondary techniques. The secondary techniques are harder to implement, and the influence on the performance is less significant (improvement can be about 5 percent of the cache degradation).

To achieve the highest level of performance, optimize both the instruction and the data cache. Optimize the D-Cache before optimizing the I-Cache because changes in the D-Cache may require some changes in the code. When the code structure is stable, optimize the I-Cache. Sometimes, there are trade-offs between optimizing the algorithm for D-Cache and optimizing the algorithm for I-Cache. In such a case, minimize the total degradation (I-Cache degradation + D-Cache degradation).

For example, in some applications the data temporal locality contradicts the program temporal locality. This is because if several sequential processing stages are done over one data block followed by same processing stages being done over the next data block, the data temporal locality (data reuse) is very high, but the program temporal locality (code reuse) is very low. However, if one processing stage is performed over all data blocks, followed by the next stage being performed all data blocks, the program temporal locality is very high, but the data temporal locality is very low. Keep this tradeoff in mind and evaluate each application on a case-by-case basis.

2.1 Elementary cache configuration

Elementary configuration requires the following steps:

1. Enable cacheable accesses.
2. Enable hardware next line prefetch on any access.
3. Set the target type to “Memory” (and not “Peripheral”) for the cache attributes to take effect.
4. Choose the write policy (L2 cache only). Default should be “cacheable write-back”.
5. Mark the area as coherent.
6. Set the gathering policy for the SGB.

These steps are simple to implement and they do not require a deep understanding of the code.

2.1.1 Enable cacheable accesses

The accesses policy are non-cacheable by default. The first optimization step is to enable cacheable accesses because the degradation otherwise can be several hundred percents.

To set a cacheable code segment in the memory (the segment can include all of the memory space), perform the following two operations:

1. Enable cacheable instruction accesses (only first time) - Set the MCR[ICE] bit in the MMU.
2. Configure a program segment descriptor in the MMU with the required cache parameters and enable it (M_PSDAx [DC]=1).

To set a cacheable data segment in the memory (the segment can include all of the memory space), perform the following two operations:

1. Enable cacheable data accesses (only first time) - Set the MCR[DCE] bit in the MMU.
2. Configure a data segment descriptor in the MMU with the required cache parameters and enable it (M_DSDAx [DC]=1).

To make a code/data segment in the memory cacheable in L2 cache (the segment can include all of the memory space), perform the following operations:

1. Enable the L2 cache.
2. The configuration of the L1 caches will be automatically applied to the L2 cache.

2.1.2 Enable hardware next line prefetch

Hardware next line prefetch is a feature of the L1 caches. This feature is beneficial when the memory accesses of the application have spatial locality (see Section 1 Cache memory). It may cause invalidation of other lines and thus waste cache space. Hence, this type of prefetch is only beneficial for sections which have very good code locality. In most cases, activating prefetch can reduce the cache degradation by 10 to 50 percent. Enabling hardware prefetch imposes extra load on the buses, which can increase the miss penalty on misses that occur during the prefetch operation. The prefetch might also increase the overall power consumption. In most cases, the pre-fetch has a positive overall performance impact.

2.1.3 Set the target type to “Memory”

The target type is “Peripheral” by default, for safer accesses. In order to let the cache attributes take effect, perform the following operation:

1. Clear the M_DSDCx[DP] bit in the segment descriptor in the MMU. This will Configure the data region with the “Memory” attribute (as opposite to “Peripheral”).

2.1.4 Write policies

Write policies are set only in the L2 cache. The L1 cache policy is write-through, in which core are stored temporary in the SGB, where they are merged until sent to the higher level cache. To improve merging SGB tries to keep write data inside as long as possible, writing it to the external memory only after reaching watermark, memory barrier or dedicated cache instruction.

The L2 cache supports the following policies:

- Non cacheable on read and write (NC)
- Cacheable write back (WB); the accesses are update the cache if they are cache hits. If the write access is miss in the cache, its address is fetched from high-level memory to L2 cache and then write is treated like hit. Modified data is written to higher level memory on L2 cache line thrash.
- Cacheable write through (WT); the accesses are written to higher-level memory and update the cache if they are cache hits. If the write access is miss in the cache, it is only written to higher level memory, no allocation is done, and the cache is unaffected.

2.1.4.1 Select the write policies for L2 cache

For each data segment, Configure the write policy of a data segment in L2 Cache using the Data Write Through field in the corresponding MMU data segment descriptor register A (M_DSDAx [DWT]).

The following table lists recommended associations. This list is a recommendation only. Typical memory sections are associated with a specific write policy, but there are exceptions.

Table 1. Recommended cache policies

Typical Memory Section	Write Policy Recommendation
Scratch memory, Stack	Cacheable write-back Optional • Non-coherent when scratch data is private
Read only input buffer, ROM tables	Cacheable. Optional: • Data that is read once: Mark the reads with “streaming” pragma to avoid cache pollution • For very small, scattered read accesses, consider non cacheable • Non-coherent for true ROM
Input buffer that is overwritten (in place)	Cacheable write-back. Optional: • Mark the reads with “intent to write” pragma to optimize hardware coherency • Or consider Cacheable write-through if the consumer of the output is not in the same cluster to evict the line from L2 upon writing it.
Output buffer to consumer outside cluster	Non cacheable or Cacheable write-through.
Output buffer to consumer in cluster	Cacheable write-back.
Inter core semaphores	Non-Cacheable. Must be marked as “Coherent” in MMU.
Memory mapped registers	“Peripheral”

2.1.5 Mark the memory area as coherent

In order to enable coherency management by hardware, to improve the coherency performance, perform the following operation:

1. Set the M_DSDCx[DCM] bit in the segment descriptor in the MMU. This will Configure the data region to be coherent.

2.1.6 Set the gathering policy for the SGB

The SGB performs as a write cache, as it stores the last writes of the core, and returns the latest data to the core on read after write scenarios. The SGB stores each core write in a 64Byte entry (16 entries in total). Since most writes are smaller than 64B, it is useful to gather several different writes to the same entry, in order to gain performance in two ways. First, gathering few writes in one entry means that written data stays longer in the SGB (entries eviction rate is lower), meaning higher hit rate for future core reads. Second, reducing the number of writes to L2 (one big write instead of several small writes), therefore saving bus slots. Therefore it is clear why gathering accesses is important for performance.

Gathering policy is disabled by default. To enable access gathering in the SGB, perform the following operation:

Enable the accesses gathering in the SGB - Set the MCR[GE] bit in the MMU.

NOTE

> The M_DSDCx[DP] bit in the relevant segment descriptor in the MMU must be cleared for the SGB gathering to take effect. This will Configure the data region with the “Memory” attribute (as opposite to “Peripheral” where gathering is not allowed).

2.2 Primary optimization techniques

Primary techniques include the following:

- Planning the memory layout.
- Configuration of shared memory areas.
- Software initiated background allocation.
- Partitioning of the L2 cache.
- Locking parts of the L2 cache (M2 like memory).

These steps are relatively simple to implement and only require basic understanding of the code.

2.2.1 Memory layout

Using the fastest memory available reduces the number of stall cycles per miss. When the fastest memory is not large enough to contain all the required data and code, place the highest-access data and the most critical code sections in the available space in the fast memory. The SC3900 Cluster allows the programmer to dynamically lock lines of the L2 cache resulting with a M2 like memory. This helps keeping critical code/data very close to the core, without a risk of being thrashed. SW architecture should minimize the amount of data sharing/movement between different clusters to better utilize the L2/M2 shared resources under each cluster.

2.2.2 Software initiated background allocation

The SC3900 FVP Cluster supports software initiated background allocation (aka touch loading) into its caches. The mechanism works for both for L1 and L2 caches. Using this mechanism correctly can reduce the miss ratio of the caches and reduce the miss penalty of L1 caches. The purpose is to initiate the load before the core needs the data/program, and thus when the core makes the access, it will be already hit in the cache. The load should occur early enough to accomplish the purpose, but not too early. If the lines of the load reside too long in the cache without being accessed, they may be replaced.

2.2.2.1 Granular allocation

These instructions are designed for fine granularity data loading into L2 cache, or both L2 and the L1 data cache (not available for program).

To load 64 bytes of data into L2 cache, simply issue a single core instruction: “dfetch.L2”. The L2 cache performs the load in the background, without stalling the core. To load the data to the L1 data cache as well, simply change the suffix to “L12”: “dfetch.L12”.

2.2.2.2 Block allocation (using cache management engine)

When loading larger blocks of data/program into the caches, it is not useful to overload the core with many fetching instructions. Instead, there is additional unit, named CME (cache management engine), to manage these operations in the background. The SC3900 cache system supports a rich set of control functions (for example, pre-fetching operations, coherency operations, and status query operations) that the user can activate on the caches and the CME supports all of those. The CME holds a set of task channels that the user configures. It then monitors the respective address buses (program and data) and drives accesses on them from the active task channels using free access slots that are not used by functional core accesses. The CME is operated with designated core commands, and can be also configured from other cores with registers configuration.

The following pseudo code describes how to generate a simple core command that fetches a data block into both D-L1 and L2 caches.

a) Update Register Rx to contain the start address of the block to be fetched. Address must be 64 byte aligned

b) If required, set the lower bits of Rx (5-0) to contain additional controls, (for example - completion event)

c) Modify Register Ry to contain the block size, divided by 64. Then set bit 11 to “1”

d) Initiate the following core command:
DFETCHB.L12 Rx, Rz BCCAS Ry SYNC.B

Register Rz will be updated with the selected channel index for the task, and the status of the command (command might fail if all channels are busy)

2.2.3 Partitioning of the L2 Cache

The L2 cache partitioning mechanism enables better utilization of the cache space by assigning certain parts of the cache to specific Purposes/Initiators. It is good for keeping memory areas that might be required simultaneously (like input/output buffers, program/data) from thrashing each other from the cache. The L2 cache supports a flexible way partition/allocation control scheme. Each transaction that misses in the cache looks in a table to determine whether or not to allocate and which ways are available for allocation. Table entries are matched by comparing partition IDs that are composed from the static core ID and 2 bits configurable in MMU per descriptor.

2.2.4 Locking parts of the L2 Cache (M2 like memory)

The benefit of placing critical code/data in M2 is that it stays very close to the core, without a risk of being thrashed. The L2 cache can be locked on a per line base (per 64 byte) and generate an M2 like memory behavior. The CME prefetch maintenance instructions have the following variant with L2 lock: prefetch is locked and will not be thrashed until unlock. This feature keeps frequently used data in the L2 cache so that all of its thrashing is not done by big and rarely used data. Because lock is actually a destructive instruction that can reduce the active L2 cache size if done improperly, it is limited to block prefetch maintenance instructions as they cannot be speculated.

To ensure the locking process is done on a specific way/s in the L2 cache and not to arbitrary ways, first step should be to assign a partition in the L2 cache to the area to be locked and only then initiate the lock process. After the CME has finished the locking process the partition can be removed as the locks are set and will remain until unlocked. If not done this way, the usage of partitions will not be efficient as they will include lines that are locked thus significantly reducing the partition efficiency.

When choosing which area should be locked in the L2 cache several aspects need to be taken into account:

- Frequently accessed or program/data
  - Locked space is valuable thus should be assigned carefully for areas that are frequently used
- Latency sensitive program/data
  - Consider processing which is time critical
- Lock data that is modified only by initiators under this cluster
  - Otherwise modifying outside the cluster will cause invalidation by the HW coherency mechanism. The lock will not break but data will be invalid causing a miss when accessed later on.

The following figure Presents a simple L2 Cache allocation example:

- 256KB private (2-way) L2 data cache per core
- 256KB shared (2-way) L2 program cache for both cores
- 1.25MB locked M2 DDR space for shared buffers

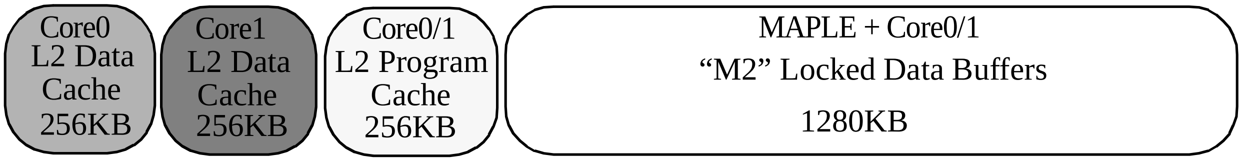


Figure 8. Partitioning Example

2.2.5 L2 Cache Stashing

The SC3900 FVP Cluster L2 Cache supports CoreNet(tm) stashing accesses which allows DMA-based IP in the SoC to push to the L2 cache information that may be requested in the near future by the core, thus significantly reducing latency. Using this mechanism correctly can reduce the miss ratio of the caches and reduce the miss penalty of L1 caches.

Like an SW initiated background allocation, stashing to a cache is a performance hint. The stash operation initiated by a device can improve performance if the stashed data is pre-fetched into the targeted cache prior to when the data is accessed. This avoids the latency of bringing the data into the cache at the time it is needed by the processor. However, since stash operations are hints, depending on conditions within the memory hierarchy and the core, stashes may not always be performed when requested. A module that initiates stashing operations to the core can optimize its usage of stashing if it is configured to understand the amount of buffering dedicated to incoming stashing operations.

Stash accesses are achieved by PAMU translation of regular DMA write accesses. For more information see PAMU chapter in the SoC Reference Manual.

2.3 Secondary optimization techniques

The techniques in this section have a modest influence on performance, and they can be used for further optimization if the primary techniques described in Section 2.2 Primary optimization techniques,” are not sufficient.

2.3.1 Second-order reduction of the number of misses and the miss penalty

There are a few factors that influence the number of misses and the miss penalty, although this influence is of a relatively low degree:

- Availability of the bus

  In the B4xxx devices, the caches share the memory buses with other bus masters. If the bus is not available, access to memory is delayed until the bus is available, thus increasing the miss penalty. To reduce this effect, lower the internal bus utilization to increase the probability of bus availability.

- Percentage of unneeded data lines

  There are scenarios where the programmer has *a priori* information about unneeded data lines, although the data was used recently, for example scratch buffers used for intermediate calculation. These data lines take up space in the cache, and because they were used recently, they are not necessarily evicted from the cache when space is needed. Instead, a needed line may be evicted because it seems to be the least recently used. Accessing this evicted line results in a miss. Minimizing the effect of this phenomenon decreases the number of misses. To accomplish this, evict unneeded data lines from the cache, using CME block invalidate or flush operations.

The following table summarizes the previously listed factors arranged in order of significance, that is, how each factor affects miss penalty or the number of misses, and whether the factor should be increased or decreased to improve the cache performance. Due to the interdependence between the miss penalty and the number of misses, the table mentions the one that is more significantly influenced by the factor.

Table 2. Influence of Specific Factors on Cache Performance

Factor	Effects (Number of Misses or Miss Penalty)	Factor Should Be Decreased or Increased to Improve Performance
Proximity of data to the core	Miss penalty	↑
Continuous data accesses	Number of misses	↑
Availability of the bus	Miss penalty	↑
Percentage of unneeded data lines	Number of misses	↓

2.3.2 Code size reduction

When code size is smaller, more of it can fit into the cache. Reducing the code size may also improve the performance of the instruction cache, but at the expense of the core performance (there is a tradeoff between size and speed). Use code size optimization when the cache performance degradation is more significant than the speed degradation caused by the code size optimization. If you are not sure, check the performance with and without the code size optimization and choose the best one. Freescale recommends that you compile the entire project with code size optimization by using the compiler #pragma opt_level = “OS” (“OS” means Optimization For Size).

2.3.3 Memory reuse and data size reduction

Reduction of data size, of course, decreases memory consumption. This is independent of the cache itself. Because cache performance improves with reduction of the accessed data’s size, there is a double benefit to reducing data size. Smart reuse of the memory area can also achieve further performance improvement, for example by reusing two data structures whose accesses are close in time. When the second data structure is accessed, it is already in the cache, thus resulting in a cache hit.

2.3.3.1 Buffer overwriting

To improve cache performance, memory accesses can reuse a physical memory area, not necessarily with the same specific data. This reuse can be achieved by overwriting an area holding data that is no longer relevant. The technique uses a buffer A that requires continuous access, and whose data, upon being read, is obsolete. Data is taken from buffer A, used for calculations, and the results are placed in a buffer B. In the second stage, buffer B contains the input data, which is being read for calculations, and the results are now placed in buffer A, overwriting the obsolete data there. In a cached architecture, this technique increases the data section reuse and thus can improve the cache performance.

Accesses are not made directly to the input buffer but to the local copy of it. If we use buffer overwriting, each part of the local input buffer that was already used can be made available for calculations or for the results of calculations. The temporal locality is increased, thus increasing data reuse.

For example, in Radix-FFT calculation, there are few “butterfly” stages as can be seen in Figure 9. In each stage, the elements from the input buffer are read and multiplied. The results are then stored in the output buffer, which is the input buffer for the next stage. This is a classic scenario for buffer overwriting, where two buffers can be used for all the butterfly stages. In each odd butterfly stage buffer A is the input buffer, and buffer B is the output buffer, and in each even butterfly stage buffer B is the input buffer, and buffer A is the output buffer.

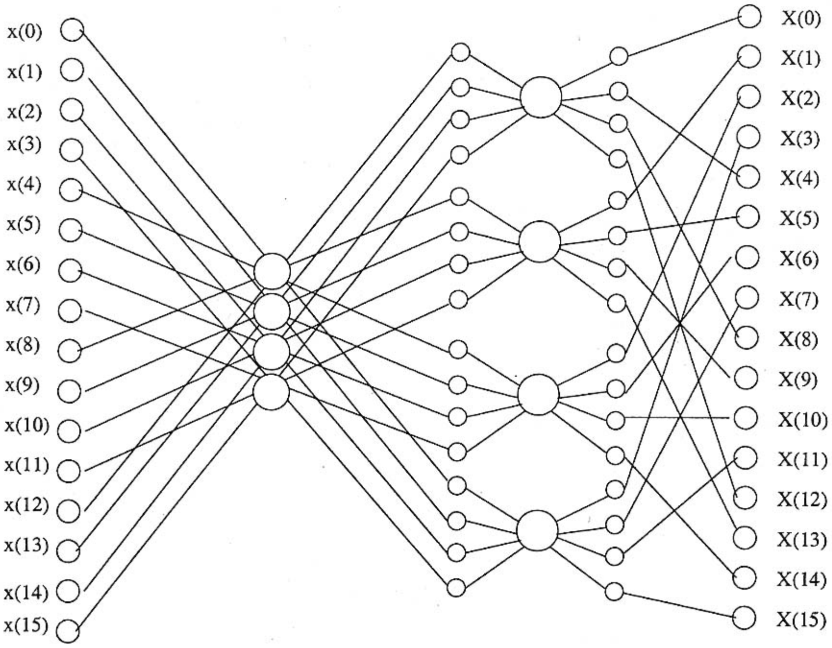


Figure 9. Butterfly stages in Radix-4 16 point FFT

2.3.3.2 Appropriate data types

Always chose efficient data types. For example, if the data is at most 8 bits wide, it should be declared as `char` rather than `int`, thus reducing the data size so the cache can contain more data.

2.3.4 Optimizing cache allocation with sw coherency instructions

The SC3900 Cache maintenance operation (Granular or Block) are designed to solve coherency issues. However, two groups of these commands can also be used for improving data accesses' cache performance, as follows:

- *Flush* (Relevant for L2 Cache)
  - Evicts lines that belong to a specific data range from the cache
  - Writes back to memory the lines that are "Modified"
- *Invalidate* (Relevant for L1 and L2 Caches)
  - Evicts the affected lines from the cache
  - Does not write back lines that are marked as "Modified"

The following guidelines are provided for the use of the above maintenance operations:

- If the programmer knows that a data section is not going to be used for a long time, but the caches does not have that information, it can be forced out from the Caches. This creates space for new data and prevents the eviction of data that is going to be reused in the short term, which indirectly reduces the number of misses.

- The general optimization concept of using the *invalidate* command is the same as *flush*, except that if the evicted data is obsolete, it should not be written to memory. This should be used with care as it can be destructive when applied on a wrong address range (unlike *flush*).
- The use of *invalidate* reduces the miss rate (as does use of *flush*) and also decreases the bus load towards the upper layer memory (e.g. - DDR) by preventing unneeded memory write back accesses.

2.3.5 Avoiding capacity misses in the D-Cache

When data blocks larger than the cache size are processed, capacity misses occur. In addition, sequential access to such a data block can lead to a cache miss on *every line*. This occurs because after reaching an offset from the data block start that is bigger than the cache size, the cache is already full to capacity by the previous parts of the set. Splitting up data sets and processing one subset at a time can eliminate these misses. The following examples demonstrate this technique.

Example 1. Splitting data sets to eliminate capacity misses

Data set of size N and a cache size of N/3, has a program that can read like this:

```
char set[N];
func1( set, N );
func2( set, N );
func3( set, N );
func4( set, N );
```

To eliminate the capacity misses, modify the program as follows:

```
For (i=0; i<4;i++)
{
func1(set + (i*N)/4 , N/4);
func2(set + (i*N)/4 , N/4);
func3(set + (i*N)/4 , N/4);
func4(set + (i*N)/4 , N/4);
}
```

However, it is not always possible to divide functions into independent parts such that each of them accesses only one part of the input.

Example 2. Eliminating capacity misses in MPEG4 motion estimation

Consider a motion estimation process in an MPEG4 encoder with 4CIF frame resolution (704× 576 pixels). Figure 10 shows the motion estimation process.

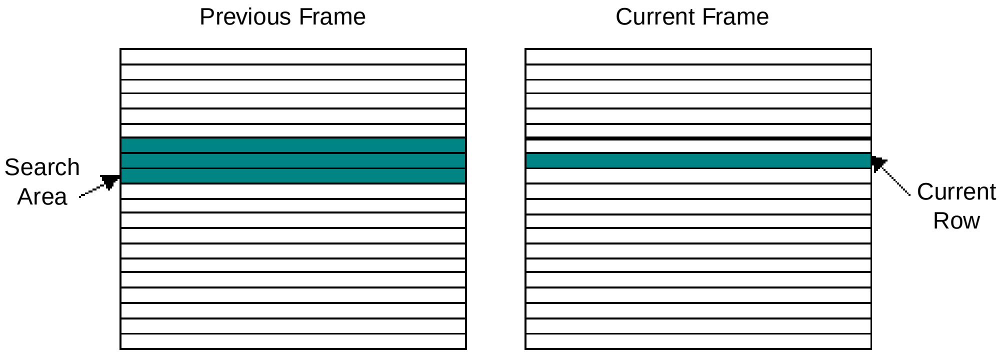


Figure 10. Implementation of raw motion estimation in MPEG4 encoder

- The Y macroblock size is 16 × 16 pixels, which consumes 256 bytes of memory.
- The frame contains 36 rows of macroblocks, each row consisting of 44 macroblocks.
- The encoding is done on the current row of macroblocks.
- For each macroblock, the motion estimation process contains two main parts:
  - — Find the matching macroblock in the previous frame and calculate the motion vector from it.
  - — Calculate the difference between those two macroblocks.

For practical reasons, the search for the matching macroblock is not done in the entire previous frame but only in the same row of macroblocks, the row above, and the row below in the previous frame.

Consider two options for a search algorithm:

- **Option 1**: Calculate all motion vectors for the macroblocks of the row and then calculate the differences for all macroblocks of the row.

```
For ( i=1; i<44; ++i ) {
    Motion_vector[i] = FindMotionVector(current_row[i]);
}
For ( i=1; i<44; ++i ) {
    Y_diff[i] = Difference(current_row[i], Motion_vector[i]);
}
```

- **Option 2**: For every macroblock of the row, calculate the motion vector and the difference.

```
For ( i=1; i<44; ++i ) {
    Motion_vector[i] =  FindMotionVector(current_row[i]);
    Y_diff[i] = Difference(current_row[i], Motion_vector[i]);
}
```

A row of Y macroblocks consists of 44 macroblocks × 256 bytes = 11 Kbyte of data. To find the motion vectors of the current row of macroblocks, four rows of Y macroblocks are accessed (current row of macroblocks + 3 rows of macroblocks from the previous frame), which equals 44 Kbyte. Because the accessed data is larger than the cache, part of it is dumped.

In option 1, part of the row is evicted during the first part of the calculation of the motion vectors. Access to that part of the row of macroblocks in the differences calculation results in a cache miss.

Option 2 solves this problem because all calculations are performed together (close in time) for each macroblock. The `Difference` function does not cause a cache miss because the current macroblock and the matching macroblock are accessed in the motion vector calculation.

2.3.6 Avoiding capacity misses in the I-Cache

If the I-Cache is not large enough to hold all the functions of a loop, the loop may need to be split up to achieve code reuse without evictions. Splitting the loop 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, is larger than the size of the I-Cache. The code loop is split to allow both functions to execute from the I-Cache repeatedly, considerably reducing misses. However, the temporary buffer tmp[] now has to hold all intermediate results from each call to function_1.

Example 3. Combined code size is larger than I-Cache

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

Example 4. Code split to execute from I-Cache

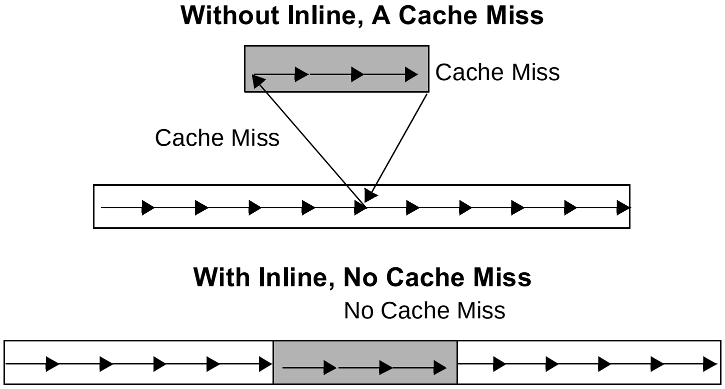
```
for (i=0; i<N; i++)
{
         function_1(in[i], tmp[i]);
}
for (i=0; i<N; i++)
{
         function_2(tmp[i], out[i]);
}
```

2.3.7 Avoiding conflict misses

A very significant advantage of the SC3900 FVP cluster is that its caches are 8 way set associative, so conflict misses are less likely to occur.

2.3.8 Function inlining

If a change of flow, such as BRA and JSR, occurs and the called function is not in the cache, a miss occurs. To minimize the number of misses, function inlining is performed. When a function is inlined, its code is prefetched with the caller function code so that misses at the start of the called function are reduced.


Without Inline, A Cache Miss

With Inline, No Cache Miss

Figure 11. Function Inlining

However, if the inlined function is called from many different places and is inlined in all these places, the code size increases, counteracting the benefit of inlining. It is advisable to inline only functions with the following properties:

- They are called from very few places (up to three places).
- Their code size is very small.

When using the CodeWarrior StarCore compiler, use the following pragmas for functions inlining:

- To inline a function, use `#pragma inline`.
- To inline the function just in a specific call location, use `#pragma inline_call <function>`.

See the CodeWarrior compiler documentation and help for details.

2.3.9 Function alignment

A program miss is served by the ICache by allocating a line of 128Byte and allocating, it with critical 64B word first and then the adjacent 64B. After serving the miss, if enabled, the cache continues with next line pre-tech.

Aligning a function with the beginning address of a cache line decreases the total number of cache lines in which the function resides in, thereby reducing the miss count. However, note that function alignment increases code size because empty spaces are created, which counteracts the benefits of aligning the functions and can even degrade application performance.

Assembly functions are aligned by adding the phrase `.align 128` at the start of each function to be aligned. The following example shows the way to align a C code function using the `.pragma` directive.

```
void foo()
    {
        #pragma align foo 128
        printf("Hello!\n");
    }
```

2.3.10 Adapting memory layout to the prefetch direction

It can be beneficial to adapt the memory layout of data structures to the direction in which they are accessed. For instance, if an array of the FIR filter is accessed backwards, it is better to put it reversed in memory.

Example 5. FIR from an echo canceler

The main loop of this FIR is (first version):

```
s = 0;
for (i = 0; i <= m; i++)
{
     s = L_mac (s, h[m-i], x[i]); //s=s+h[m-i]*x[i]
}
```

As can be seen, the access to the filter **h** is downwards, so if the array **h** is not in the cache before the filter commences its work, the prefetch does not work efficiently since the prefetch direction is forward. A technique to solve this problem, is to put the array **h** reversed in memory. If **h** is reversed in memory, the code should be changed so the access to **h** is correct.

The new code is (second version):

```
s = 0;
 for (i = 0; i <= m; i++)
 {
      s = L_mac (s, h[i], x[i]); //s=s+h[m-i]*x[i]
 }
```

In this case, the prefetch of **h** works efficiently.

2.3.11 Full usage of buffer in a multitask system

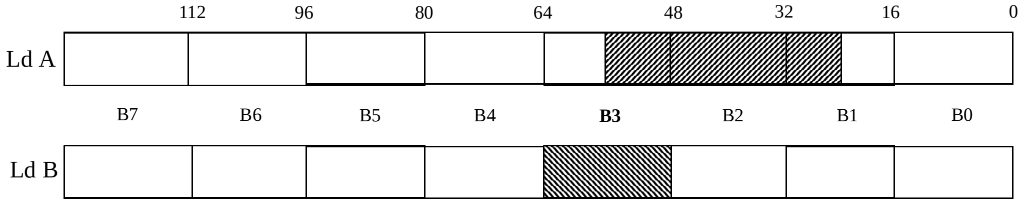
In multitask systems, while each task works on its own buffer, a great deal of buffer data can be in the cache, due to a high frequency of accesses to many areas in the buffer. In this case, finish working on this buffer before a task switch is called. Avoiding this kind of task switch saves refetching the buffer when the task again uses the core.

2.3.12 Improving code linking

In many cases, the code consists of conditional part that usually is not executed (errors or rarely used branches). Use “likely” or “unlikely” keywords to guide the compiler and linker to leverage those cases in order to gain performance, by separating this code to a different memory section. This will improve the ICache utilization by compacting the relevant code (saving space of unused code) and will help the prefetch mechanism work more efficiently. It will also improve core performance by keeping the flow of the usual cases without branching.

2.3.13 Avoiding D-Cache stalls due to double-load contention

The optimization technique described in this section is specific to the hardware architecture of the SC3900 core subsystem and is not necessarily relevant for other architectures. The SC3900 L1 D-Cache can serve 2 Load instructions per cycle, each from 8bit up to 256bit in size. In general when the accesses are hitting the cache there will not be any penalty to the core. To allow simultaneously satisfying both accesses the D-Cache memory is divided internality to 8 banks with a width of 16byte each. A penalty of one cycle will be incurred only if the two load instructions access the same bank. The example below brings a case where such a conflict is incured.



Ld A - 0x####_##24, Size 32byte
Ld B - 0x####_##48, Size 16byte

Results in Contention on Bank3

There are two ways to avoid this scenario:

- Change the location of the buffers in memory so that the offset between their addresses prevent the overlap
- Reorder the read accesses in to prevent the Double-Load in the first place

2.3.14 Fitting memory access order to the prefetch direction

This section describes the optimization of a matrix multiplication function using an example that shows the influence of memory access order on the data cache performance. The matrix multiplication application was chosen because it is an important function for digital signal processing and the basic, straightforward, implementation is not cache-friendly.

2.3.14.1 Basic implementation

Equation 2 describes the mathematics used in matrix multiplication. The straight forward implementation of this equation iterates over each row in Matrix A and each column in Matrix B to generate each element in Matrix C.

Eqn. 2

$$C_{i,j} = \sum_{r} A_{i,r} \cdot B_{r,j}$$

The problem with this implementation is the iteration over the columns in B. The iteration causes the cache to fetch an entire line into the cache (due to prefetching), but the application only uses one of the elements

($B_{i,j}$). For the next element, another entire cache line is fetched, but again, only one element is used. Assuming that there are more than 256 columns in B (the number of lines in the data cache), there is no reuse of the data in the cache and thus the cache efficiency is low. Furthermore, the frequent fetches and prefetches cause unnecessary system traffic and thus burden the entire system.

Example 6. Basic matrix multiplication code implementation

```
void standard_mult()
{
         Word16 i,j,k;
         const Word16 row1 = MATRIX_1_Y;
         const Word16 col2 = MATRIX_2_X;
         const Word16 col1 = MATRIX_1_X;

         for (i = 0; i < MATRIX_1_Y; i++)
         {
                  for (j=0; j<MATRIX_2_X; j++)
                  {
                           for (k = 0; k < MATRIX_1_X; k++)
                           {
                                results_non_optimized[i][j] += matrix_1[i][k] * matrix_2[k][j];
                           }
                  }
         }
}
```

2.3.14.2 Coefficients-vector method

The coefficients-vector method iterates over all lines of B for each line in A. This is depicted in Equation 3.

Eqn. 3

$$C = \begin{bmatrix} a_{1,1} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & \dots \end{bmatrix} + a_{1,2} \cdot \begin{bmatrix} b_{2,1} & b_{2,2} & \dots \end{bmatrix} + \dots \\ a_{2,1} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & \dots \end{bmatrix} + a_{2,2} \cdot \begin{bmatrix} b_{2,1} & b_{2,2} & \dots \end{bmatrix} + \dots \\ \dots \end{bmatrix}$$

Using this method, each row in B is fully used by each element in the current row of A before the next row in B is fetched. This method ensures that the cache remains hot (i.e most of the required data is still in the cache) for matrices A, C, and B.

Example 7. Coefficients-vector code implementation

```
void optimized_mult()
{
         int i, j, k;

         for (i=0 ; i<MATRIX_1_Y ; i++)    // For every row in matrix_1
         {
                  for (j=0 ; j<MATRIX_1_X; j++)    // For every column in matrix_1
                  {
                           Word16 temp = matrix_1[i][j];
                           for (k=0; k<MATRIX_2_X; k++)    // calculate a part of row i in the
                                                           // result.
                           {
                                    results_optimized_2[i][k] += temp * matrix_2[j][k];
                           }
                  }
         }
}
```

3 Revision history

The following table provides a revision history for this application note.

Table 3. Document Revision History

Rev Number	Date	Substantive Change(s)
B	8/2012	Initial release
A	4/2012	Initial draft

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Document Number: AN4549
Rev. B
8/2012