Cache Usage in High-Performance DSP Applications With the TMS320C64x

Jackie Brenner *DSP Applications*

ABSTRACT

The TMS320C64x™, the newest member of the TMS320C6000™ (C6000™) family, is used in high-performance DSP applications. The C64x™ processes information at a rate of 4800 MIPs, while operating at a clock rate of 600 MHz. Processing data at these extremely high rates requires fast memory that is directly connected to the CPU (Central Processing Unit). However, a bandwidth dilemma has occurred with the dramatic increase in processor speed. While processor speed has increased dramatically, memory speed has not. Therefore, the memory to which the CPU is connected often becomes a processing bottleneck.

Cache memories can greatly reduce the CPU to memory processing bottleneck. Caches are small, fast memory that reside between the CPU and slower system memory. The cache provides code and data to the CPU at the speed of the processor, while automatically managing the data movement from the slower main memory which is frequently located off-chip.

Cache memories are still a new concept to many DSP programmers. This application report provides a high-level overview of cache-based system performance. It covers cache fundamentals, provides an overview of the C64x cache architecture, discusses code behavior in caches, and points out techniques for optimizing code for cache-based systems.

Contents

List of Figures

List of Tables

1 Introduction

High-performance DSP systems process digital data quickly. The newest member of the C6000 family, the C64x, processes information at a rate of 4800 MIPs while operating at a clock rate of 600 MHz. Processing data at these extremely high rates requires fast memory that is directly connected to the CPU (Central Processing Unit). However, a bandwidth dilemma has occurred with the dramatic increase in processor speed. While processor speed has increased dramatically, memory speed has not. Advanced process technologies have allowed both the CPU clock speed to increase and more memory to be integrated on-chip, but the access time of the on-chip memory has not decreased proportionally. By nature, SRAMs are large arrays of static storage elements. As you increase the number of storage elements, you have more capacitance on the data lines between these storage elements and the CPU. The more capacitance you have, the slower the switching time. Thus, the speed advances due to process technology are often mitigated by the increased capacitance of large memory arrays. Therefore, the memory to which the CPU is connected often becomes a processing bottleneck.

Cache memories can greatly reduce the CPU to memory processing bottleneck. Caches are small, fast memory that reside between the CPU and slower system memory. The cache provides code and data to the CPU at the speed of the processor while automatically managing the data movement from the slower main memory which is frequently located off-chip.

The C64x employs a two-level cache architecture for on-chip program and data accesses. In this hierarchy, the C64x CPU interfaces directly to a dedicated level-one program (L1P) and data (L1D) caches of 16 Kbytes each. Dedicated L1 caches eliminate conflicts for the memory resources between the program and data busses, thus increasing speed. These L1 caches operate at the same speed as the CPU. The L1 memories are also connected to a second-level memory of on-chip memory called L2. L2 is a 1024Kbyte unified memory block that contains both program and data. This unified L2 memory provides flexible memory allocation between program and data for accesses outside of L1. The L2 memory acts as a bridge between the L1 memory and memory located off-chip. See Figure 1.

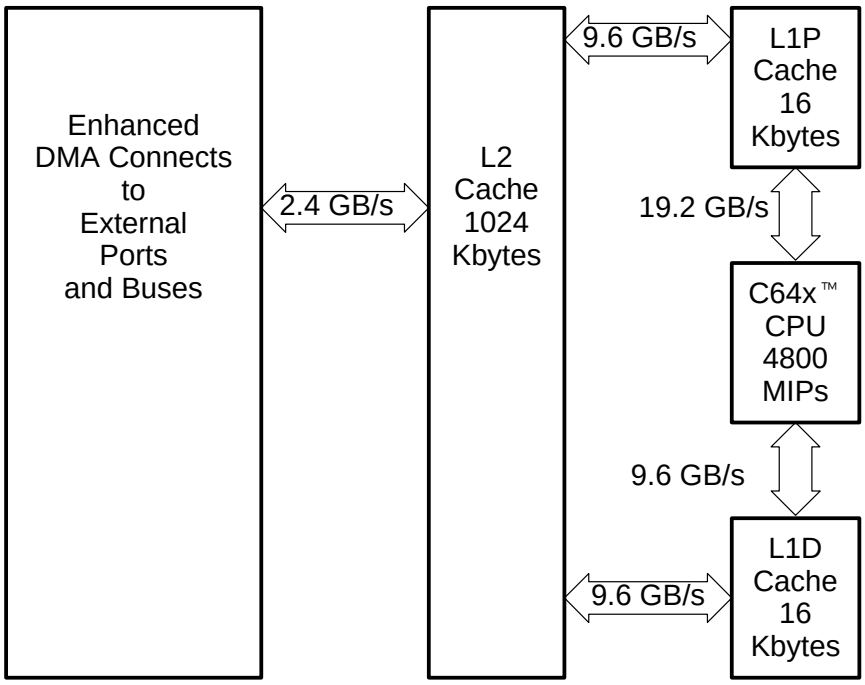

Figure 1. C64x Two-Level Cache Architecture

Cache memories are still a new concept to many DSP programmers. This application report provides a high-level overview of cache-based system performance. It covers cache fundamentals, provides an overview of the C64x cache architecture, discusses code behavior in caches, and points out techniques for optimizing code for cache-based systems. Further detail on the C6000 cache architecture and cache behavior can be found in the *TMS320C6000 Peripherals Reference Guide* (SPRU190) and the *TMS320C6211 Cache Analysis* (SPRA472).

2 Cache Concepts

Caches are based on two concepts: temporal locality (if an item is referenced, it will be referenced again soon) and spatial locality (if an item is referenced, items that are located nearby will also be referenced soon).

Let us look at a piece of C code that illustrates both these concepts:

```
for (i = 0; i < 10; i++)
      {sum += x[i] * y[i];}
```

In this loop, we are multiplying elements in the **x** array by corresponding elements in the **y** array, and we are keeping a running sum of the product. Note that the same piece of code is executed ten times in a row. The data value **sum** is also accessed ten times in a row. This illustrates temporal locality; the same pieces of code and data are accessed several times in succession. However, the individual elements in the **x** and **y** arrays are accessed only once. The data required for each iteration of the loop is the succeeding element in the array. In the first iteration, x[0] and y[0] are accessed; in the next iteration, x[1] and y[1] are accessed, etc. This is an example of spatial locality. Note that while the values within the individual arrays are in close proximity to each other, the two arrays with respect to each other may not be.

Caches effectively utilize temporal locality – the CPU is kept close to the instructions and data it is currently using. Caches also take advantage of spatial locality – instructions and data that are in close proximity to those being used currently by the CPU are also kept close to the CPU. Those instructions and data that have not been used recently are located farther away from the CPU in a separate level of memory. This is not very different from a flat memory architecture. When we want the fastest possible performance in a flat memory model, we keep the code and data we need in on-chip memory. Those instructions and data that are not of immediate importance can be kept in slower, off-chip memory. The main difference between a flat memory model and a cache model is how the memory hierarchy is managed. In the flat memory model, the programmer manages the hierarchy (brings new code and data on-chip when needed) while in a cache system, the cache controller manages the hierarchy.

Since the cache controller greatly simplifies managing the memory hierarchy, a programmer can quickly develop his/her system. This is especially true in large systems where it is very complex to program the data flow using a Direct Memory Access (DMA) approach, or very inefficient to use the CPU to manage the data flow which would be necessary in a flat memory architecture. Thus, cache-based systems provide a large advantage over flat memory systems in terms of data management simplicity and rapid development time.

2.1 Cache Fetch Flow

In a cache-based system, when an access is initiated by the CPU, the cache controller checks to see if the desired value resides in the cache. If the value (be it an instruction or a piece of data) is in the cache, a cache hit occurs and the value is sent to the CPU. If the value the CPU is requesting is not in the cache, a cache miss occurs. On a cache miss, the cache controller requests the value from the next level of memory. For a C64x L1P or L1D miss, the next level of memory looked at is L2. In the case of an L2 miss, the next level memory to be searched is off-chip memory. See Figure 2.

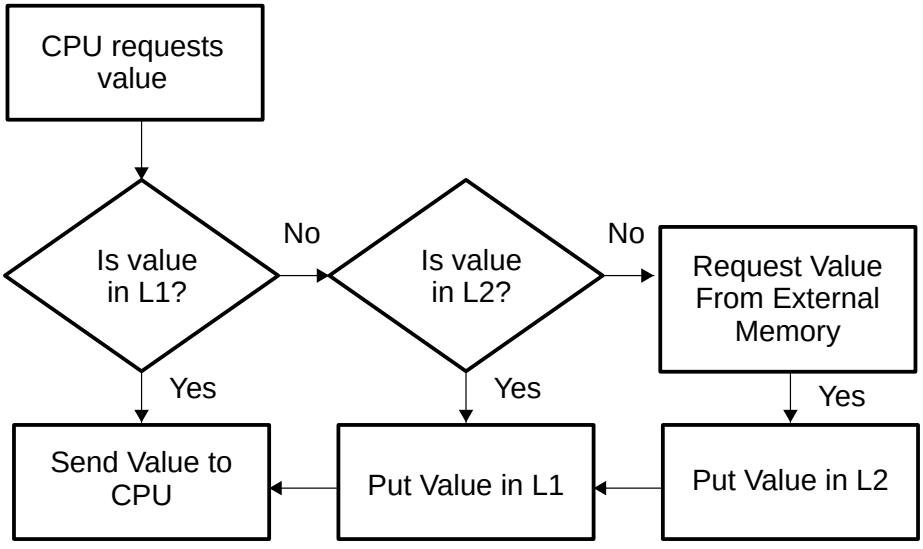

Figure 2. C64x Two-Level Cache Fetch Flow

When a cache miss occurs, the C64x CPU waits until the program/data value becomes available. If the value is not in L1P/L1D but is in L2, the program/data is copied into L1P/L1D and then is sent to the CPU. If the value is not in L1P/L1D and is not in L2, the program/data is copied from external memory to L2, then is copied into L1P/L1D and sent to the CPU.

2.2 Determinism

A key concern in a DSP system is maintaining real-time execution. In many DSP systems, a data sample is taken and operated on until the next data sample arrives. In this scenario, there is a time budget that must be adhered to. Overrunning the time budget means missed data and perhaps improper algorithm execution. If we know that execution of a particular algorithm always completes in the same amount of time, then we can more easily maintain real-time execution. This concept is known as determinism. Given what we have learned so far about cache operation, can execution determinism be achieved in a cache based system? Let us frame this in a different way. Can we be certain that real-time is never lost? This is the more crucial question rather than knowing that each time the algorithm runs it takes the same, fixed amount of time.

In a cache-based system, we can determine an upper bound for this question. We can make the assumption that the cache is empty (or cold) and that the instructions and data must be fetched from memory outside the cache. From this assumption, we can then examine whether there is enough time to perform the algorithm. We can have similar concerns in a flat memory model. As we mentioned earlier, in a flat memory model the programmer manages the memory hierarchy rather than the cache controller. Care must often be taken by the programmer when managing the data flow in a flat memory system to ensure real-time operation. Are multiple DMA channels competing for resources that would restrict movement of code and data, thereby preventing processing completion in real-time? Are interrupts constantly serviced so that the necessary processing cannot be done between data samples? These situations demonstrate that the solutions for the question of determinism are system-dependent, and are not unique to a cache based-system. The programmer must consider all the factors that would affect his/her ability to maintain real-time in either memory model.

2.3 Direct Mapped Cache

Let us look at a particular type of cache named direct mapped. In a direct map cache, any location in the processor's memory map has one location in the cache. Figure 3 shows a direct mapped cache. In this example, we have a memory map of 16 locations, and a cache that can hold four values.

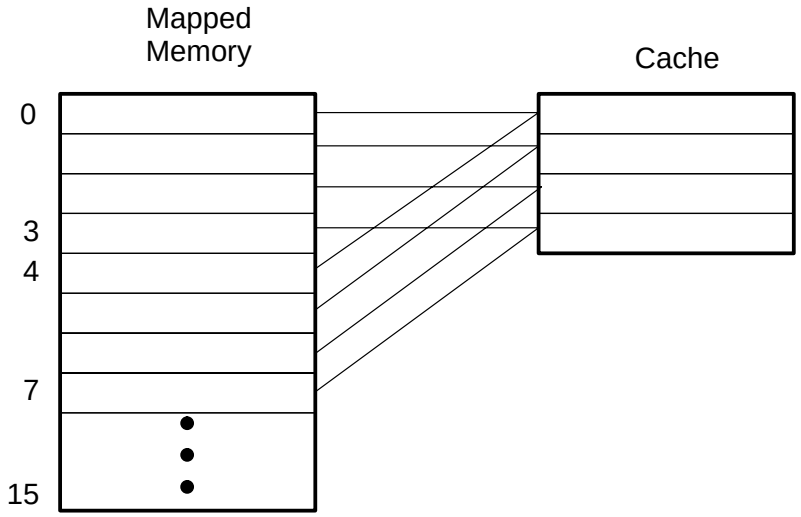

Figure 3. Direct Mapped Cache

In this example, every block of four mapped memory locations fills the cache. Notice that locations 0 and 4 reside in the same cache location. What happens if the CPU makes alternating accesses to mapped locations 0 and 4 in our example? Each access requires a cache update because this cache can only hold one of these values at any one time. In this case, the cache is constantly overwritten or thrashed.

Fundamentally, thrashing is an extreme example of cache misses. It is apparent that the best way to maximize CPU performance is to minimize cache misses. Cache misses can be minimized both by the design of the cache itself, and by the way software is designed at a system level.

3 C64x Cache Design – Brief Overview

The C64x L1P cache is a direct mapped cache that is 16Kbytes in size. This corresponds to 4K instructions. DSP algorithms such as filters and FFTs primarily consist of loops that execute the same code on multiple data locations. The L1P is large enough to hold multiple kernels. Typically, these kernels are executed sequentially. Therefore, if these algorithms are placed together in memory, program cache misses can be avoided in many cases.

The C64x L1D cache is a two-way set associative cache that is also 16K bytes in size. A two-way set associative cache allows a specific address in external memory to potentially have two separate locations in cache, each of which is referred to as a cache way. This is shown in Figure 4. Each L1D cache way is 8K bytes. A two-way set associative data cache is beneficial in a system where multiple arrays are being accessed simultaneously. In the sum of products example we showed earlier, the **x** and **y** arrays are being accessed simultaneously, but they may not be close together in memory. Having a two-way set associative cache allows us to keep both **x** and **y** arrays in the cache, and we can access both simultaneously.

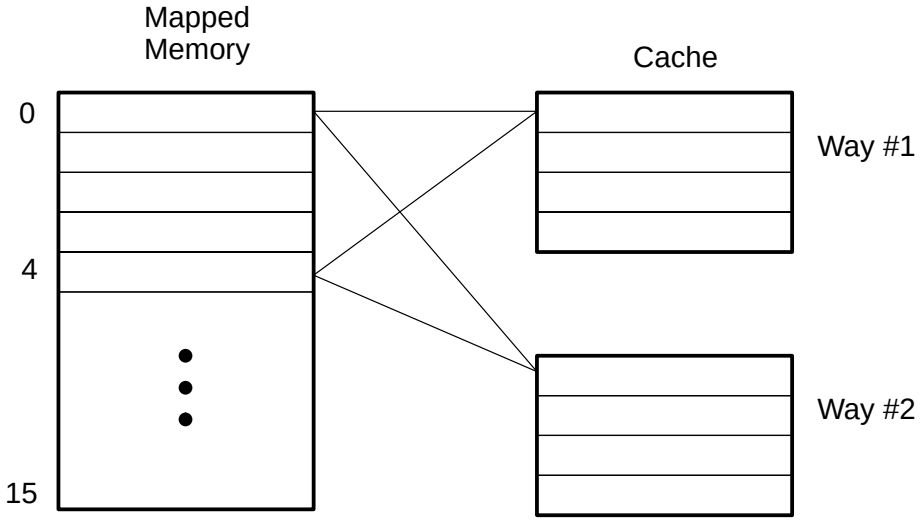

Figure 4. Two-Way Set Associative Cache

The C64x L2 memory can hold both program and data. On initial C64x devices, the L2 memory size is 1024K bytes. The large size of L2 memory greatly increase the effectiveness of the cache because misses are much less likely to occur. The L2 memory consists of 256Kbytes that can be configured as combinations of SRAM and a four-way set associative cache, and 768Kbytes that is always used as SRAM. Changing how memory is mapped allows the user to lock critical code, such as interrupt service routines or commonly called functions, in on–chip RAM. It also allows critical data sections, such as the software stack and often reused coefficients, to be locked in on–chip RAM. For the purpose of simplification, we treat L2 only as SRAM in this document.

The amount of data requested by the cache when it experiences a miss is called the line size. When the C64x L1P experiences a miss, it requests 32 bytes, or 8 instructions, from L2. When L1D experiences a miss, it requests 64 bytes from L2. The C64x architecture provides a mechanism to increase cache efficiency when there are multiple L1P/L1D cache misses. This technique is known as “miss pipelining”. If misses occur in clusters, the retrieval of the missed lines can be overlapped. This increases cache effectiveness by reducing the time it takes to retrieve the missed lines.

With these details of the C64x cache in mind, we now concentrate on how the programmer can optimize his/her system software design for performance in a cache-based system.

4 Code Behavior in Caches

The performance of a cache is sustained by maintaining locality. Algorithms that do a large amount of processing on each piece of data perform very well in a cache environment. Examples of these algorithms are block filters and Fast Fourier Transforms (FFTs). These algorithms are iterative by nature, so we achieve both code reuse and data reuse by taking advantage of temporal locality. These algorithms typically achieve 95% efficiency.

Algorithms such as a vector dot product or a vector sum do not reuse input data. These algorithms also require very little processing per sample, so more time is spent requesting new input data. Therefore, the performance of these algorithms can be improved by employing system optimizations. With system optimization, efficiencies of 85% and greater are possible.

Even if an individual kernel does not exhibit high cache effectiveness when looked at in isolation, the overall system cache efficiency can be quite high. We should focus the code structure optimizations around these functions.

5 Optimizing System Software for Cache

Optimizing system software for cache efficiency is not much different from optimizing a flat memory model system for effective code and data usage. For example, we want to group functions that work frequently together close together in memory. We do this in a flat memory system so that the DMA can effectively be used to copy these functions from external memory to internal memory. We group functions together in a cache-based system to maximize both temporal and spatial locality.

Here are some guidelines for optimizing your code system. We touch on some of these briefly as we examine the benchmarks for specific code examples.

- Allow each function to do as much processing on each piece of data as possible to increase data reuse.

- Organize data to increase cache hits.
- Partition algorithms to utilize program cache and data cache evenly.
- Group functions that work together on the same data together in memory.

5.1 Data Reuse

As previously mentioned, block filter algorithms work well in a cache environment. The block FIR filter code in Table 1 is benchmarked with different filter lengths. As the filter length is increased, the amount of work done on each particular sample increases. The convolution window is larger so there is more data reuse as well. A data set of 1024 points of 16-bit data was used with filter lengths from 16 to 64 real taps (T).

Table 1 shows the efficiency of the cache as a function of filter length. Notice that the efficiency of the cache increases as we increase the filter length.

Table 1. Effect of Filter Length on Cache Efficiency

Filter Size T	Cache Efficiency
16	92.4%
32	96.0%
48	97.3%
64	97.8%

Initially, the circular data buffer used by this algorithm is empty. This buffer, located in L1D, needs to be filled with data from the L2 memory. The filling process can be made faster by using a function that pre-fetches the data that is needed. This function takes advantage of the data cache miss pipelining mechanism discussed previously. The data pre-fetch function is similar to an initialization routine in a flat memory system that copies data from off-chip memory to on-chip memory. By utilizing this function, the cache effectiveness can be augmented even further, as shown in Table 2.

Table 2. Effect of Data Pre-Fetch on Cache Efficiency

Filter Size T	Cache Efficiency Without Data Pre-Fetch	Cache Efficiency With Data Pre-Fetch
16	92.4%	95.7%
32	96.0%	97.8%
48	97.3%	98.5%
64	97.8%	98.9%

5.2 Data Organization

In the FIR filter example above, input data was accessed primarily in consecutive order within the circular buffer located in the L1D cache. Some functions do not naturally access consecutive values, and may actually have a large data reach requirement. One example of this is an interleaver which uses a lookup table to reshuffle data. The input data is read randomly and the output data is written sequentially. These random reads from the lookup table can cause thrashing in the L1D, as the input data may have a reach greater than 16K bytes. The output data does not need to be stored in L1D. Note that, when there is an L1D write miss, the data is not allocated in L1D but written directly into L2. Therefore, the lookup table can be reordered such that the input data gets read consecutively and then gets written out randomly into a larger address reach. The output data does not effect L1D, as this data goes into L2, which has a much larger reach of 1024K bytes. This can boost the cache efficiency from 60% to 85%.

5.3 Algorithm Partitioning and Function Grouping – Video Scaling Example

Another optimization technique is partitioning the algorithm. An algorithm can be broken up into smaller pieces when either the program is too large to fit into the L1P cache, or the data is too large to fit into the L1D cache. A number of functions can fit into L1P, while at the same time providing adequate data buffers for these functions in L1D.

A video scaling application provides an example to observe both code partitioning and data partitioning optimization techniques. The video scaling code is split up into a number of kernels and data buffers. Rather than scaling the entire image horizontally and then scaling the entire image vertically, the two stages are interleaved. Also in this example, we have three different colors in the image with independent processing paths for each color. This is shown in Figure 5.

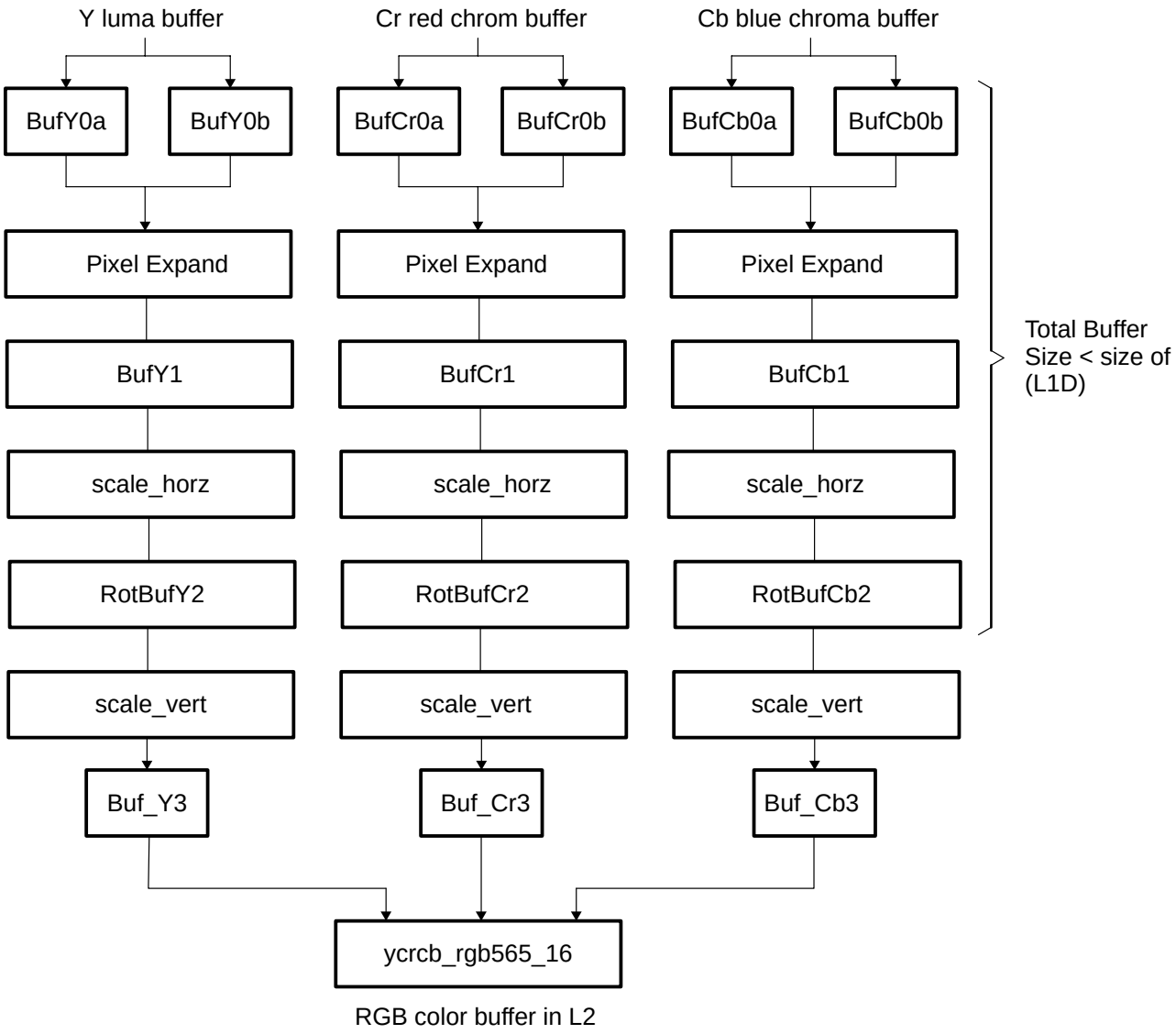

Figure 5. Data Flow for the Video Scaling Function

Each color is separated into two data buffers. Using the Y luma as an example, we have BufY0a and BufY0b. The data in the input buffers are expanded with the pixel expand function. The output data created by the pixel expand function is stored in BufY1. The expanded data is scaled horizontally by the scale_horz function, and the intermediate result is stored in the RotBufY2. This intermediate result is then scaled vertically by the scale_vert function, and the result of this scaling is placed in Buf_Y3. It is at this point that the results of the individual colors are stored in a common data buffer in L2 memory.

Besides partitioning the code and data, it is also beneficial to group the algorithms that work on the same data together in memory. For example, we would want the functions that work on the Y luma color path to be placed together in memory. We would also want to place the Y luma data buffers together in memory for the same reason. Future versions of Code Composer Studio™ Integrated Development Environment (IDE) will provide visual feedback on the effectiveness of the various code and data partitioning scenarios selected by the programmer.

Code Composer Studio is a trademark of Texas Instruments.

When you apply code and data partitioning and function grouping optimization methods as described above, cache efficiency of over 90% can be achieved.

5.4 Applying System Optimization Techniques – Vector Search Example

As shown in Figure 6, a complex reference vector is compared to nine complex input vectors. The input vector that most closely matches the reference is the one that is selected, and a dot product is performed between the reference vector and the selected input vector.

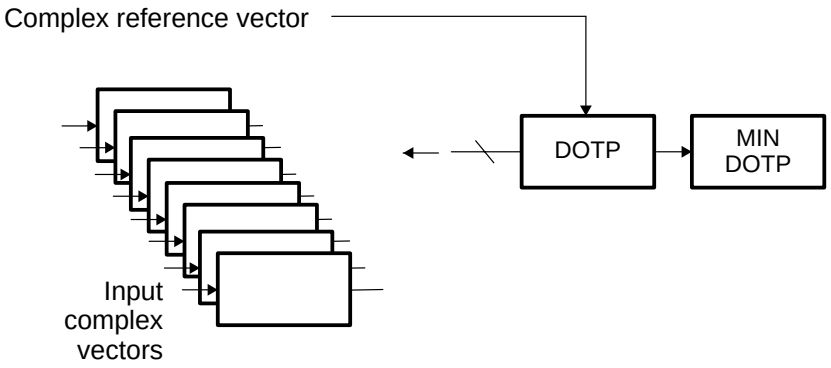

Figure 6. Complex Reference Vector

As pointed out earlier, algorithms such as a vector dot product do not reuse input data. These algorithms also require very little processing per sample, so more time is spent requesting new input data. The individual functions in the application (the vector search and dot product) do not exhibit high-cache effectiveness when looked at in isolation. However, even without applying system-level optimization techniques, the application as a whole has an overall cache efficiency of 79%. Thus, it is misleading to look at individual kernels when trying to measure the cache efficiency of the system. Furthermore, after applying the optimization techniques of function and data partitioning, function grouping, and miss pipelining, the cache efficiency for this application is raised to 93%.

6 Conclusion

Caches can solve memory bandwidth issues by supplying code and data to the CPU at the speed of the processor. Caches are small, fast memories that effectively utilize temporal locality and spatial locality to keep the CPU close to the instructions and data it is currently using. Those instructions and data that have not been used recently are located farther away from the CPU in a separate level of memory. This is similar to a flat memory architecture. In a flat memory model, when we want the fastest possible performance, we keep the code and data we need in on-chip memory. Those instructions and data that are not of immediate importance are located in slower, off-chip memory. The main difference between a flat memory model and a cache memory model is in the management of the memory hierarchy. In the flat memory model, the programmer manages the hierarchy (brings new code and data on-chip when needed) while in a cache system, the cache controller manages the hierarchy. System development can be made much quicker and easier by employing a multi-level cache architecture, since it removes the burden of managing the memory hierarchy.

Algorithms that do a large amount of processing on each piece of data, such as a block FIR filter or an FFT, perform very well in a cache environment. These algorithms are iterative by nature so we achieve both code reuse and data reuse by taking advantage of temporal locality. Algorithms such as a vector dot product or a vector sum do not reuse input data. They require very little processing per sample so more time is spent in requesting new input data. Therefore, the performance of these algorithms can be improved by employing system software optimizations.

The optimizations discussed in this application report show how structure of the overall system code can play a role in maintaining locality. These optimizations include organizing functions to permit data reuse, arranging data to reduce thrashing, and performing algorithm- and data partitioning to allow the best utilization of the on-chip memory resources. These optimizations techniques augment cache efficiency to further enhance the performance of a DSP application in a C64x-based system.

7 References

1. *TMS320C6000 Peripherals Reference Guide* (SPRU190)
2. *TMS320C6211 Cache Analysis* (SPRA472)

IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265