Memory Arbitration and Cache Management in Stream-Based Systems

Françoise Harmsze[1], Adwin Timmer[1] and Jef van Meerbergen[1,2]
(1) Philips Research Labs Eindhoven, The Netherlands
(2) Eindhoven University of Technology, The Netherlands

Abstract

With the ongoing advancements in VLSI technology, the performance of an embedded system is determined to a large extend by the communication of data and instructions. This results in new methods for on- and off-chip communication and caching schemes. In this paper, we use an arbitration scheme that exploits the characteristics of continuous 'media' streams while minimizing the latency for random (e.g. CPU) memory accesses to background memory. We also introduce a novel caching scheme for a stream-based multiprocessor architecture, to limit as much as possible the amount of on-chip buffering required to guarantee the throughput of the continuous streams. With these two schemes we can build an architecture for media processing with optimal flexibility at run-time while performance guarantees can be determined at compile-time.

1 Introduction

In many embedded systems, the bandwidth to off-chip memory is becoming an important limiting factor with respect to the system performance. It becomes especially critical when a CPU, peripherals and other (co-)processors must use the same background memory in a unified memory architecture (UMA). In media systems for instance, real-time performance is very important. Signal processing applications like video decoding and processing require a guaranteed bandwidth, (otherwise a fall-back mechanism is necessary), since such systems normally do not have much headroom to catch up when the bandwidth requirements are temporarily not met. On the other hand, a CPU requires low latency for the best performance. These two requirements may easily clash, but in this paper it will be shown that by applying a proper arbitration and cache management scheme, both objectives can be met. For the media processing application domain on which we focus in this paper, we wish to have a *flexible solution* which allows for *run-time reconfiguration* of applications while *performance guarantees* need to be known at compile-time. Exploiting the characteristics of media processing plays a very important role in obtaining the best solution. The signal processing usually consists of stream-based processing with FIFO periodic communication behaviour. We see that video processing is usually done on a field or frame basis. Any reconfiguration of the application can be done at the start of a new video field. For the video processing units we can thus distinguish between two parts of an application: the run-time of a program is when the video processing algorithm is being performed on a video field, and the configuration-time of a program is when any new parts of the program or parameters are fetched at the start of a new video field. In this paper we will explain our usage of the difference between configuration-time and run-time of a program for our memory arbitration scheme.

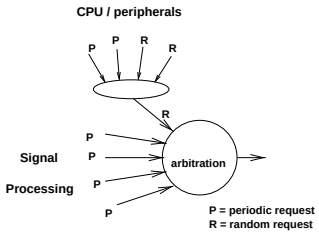

Figure 1. Basic view of the background memory arbitration

While the signal processing shows a periodic communication behaviour, the CPU and its peripherals on the other hand will show random burst behaviour (Figure 1). The CPU requires a low latency for a better performance. We wish to allow the CPU to access background memory in large bursts, as this will give a better average CPU performance in a well-balanced system: that is, a system in which the CPU is not asking too much bandwidth too often. The high performance required for video processing cannot just be obtained by increasing the clock frequency of the processor elements as this will also drastically increase the power consumption of an IC. The reason to consider multi-processor architectures is that for such applica-

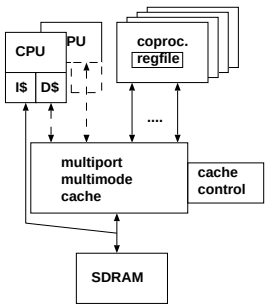

Figure 2. Generic view on multiprocessor memory hierarchy

tions true task-level parallelism is necessary to obtain the required high performance at a reasonable cost. In this paper we will concentrate on the communication aspects between (co-)processors and background memory, in the context of multi-processor architectures for high-throughput media (video) applications. In a multi-processor architecture with many (co)processors, the hierarchy of the memory architecture is very important. Figure 2 shows a generic multi-processor memory hierarchy and the different kinds of usage for a *multi-mode* cache. By multi-mode we mean that part of the cache can be used for FIFO-based caching of data for media processors, while the remainder of the cache can be used for 2nd level caching or even 1st level caching for CPUs, with standard caching and pre-fetching mechanisms. However, the latter mode of caching lies beyond the scope of this paper. In most related work, data flow analysis

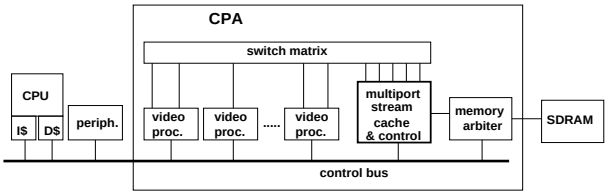

Figure 3. Co-Processor Array (CPA) for video processing

and compilation techniques are used to optimize the memory management [1],[4]. We focus however on an architectural method for the memory hierarchy which gives us flexibility at low area cost. In this paper we describe the stream caching scheme required for the media processors. For this scheme we will introduce a method to lock cache lines for periodic data streams. Figure 3 shows an instance of a multi-processor architecture called the CPA. This CoProcessor Array shares an external background memory with a CPU and peripherals. Figure 3 shows that the coprocessors use on-chip communication via a switch matrix which handles its own arbitration [3]. We will use this architecture to illustrate our caching and memory arbitration scheme.

The problem addressed in this paper is : how to optimize the amount of on-chip buffering for handling data streams to and from background memory with the following three side conditions:

- compile-time guarantees of the performance
- flexibility at run-time and configuration time
- CPU and peripherals have access to background memory in burst mode for optimal performance

We will introduce a background memory arbitration scheme and a method for guaranteeing the bandwidth demands for high-throughput real-time tasks. The task scheduling of the CPU resulting in random streams as shown in Figure 1 lies beyond the scope of this paper, but priority-based schemes [5] can be used for the arbitration. The arbitration for off-chip communication is addressed in Section 2. In Section 3 we will show that a separate cache for each (co)processor will make the system too large, and that a central cache will be more advantageous. In a typical CPU cache the data and instructions of one task can occupy the whole cache. In a multi-processor architecture with many independent streams, data from different tasks can concurrently occupy a central cache. Therefore we require the notion of stream caching [7]. Section 4 describes a method to overcome the disadvantages of cache fragmentation and all methods will be illustrated in section 5 using the CPA architecture.

2 Background memory arbitration and response time calculations

In [2] a simple and efficient memory arbitration scheme is presented which supports continuous streams along with random requests. A service cycle of N clock cycles is defined, see Figure 4, in which M clock cycles are reserved for continuous (media) streams. These media streams do not require a low latency, since their demand for data is known well in advance. Pre-fetching from memory can thus be used for data streams coming from memory, and data streams towards memory can be buffered for a while. Given $N \epsilon \mathbf{N}$ and $M \leq N$, $R = N - M$ clock cycles are available for random traffic such as generated by a CPU. The random traffic has highest priority, thus ensuring low latency, provided that enough cycles within the service cycle are remaining for the continuous streams. If this is not the case, the continuous streams will have priority. Of course, the value of M within N must be large enough to guarantee the periodic streams their required bandwidth.

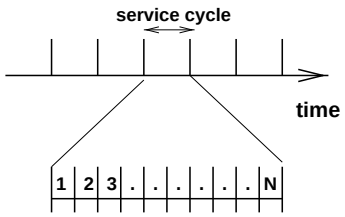

Figure 4. Definition of a service cycle

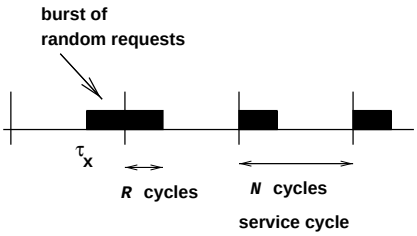

Figure 5. Critical problem instance to find the worst case response time

Although the memory management scheme guarantees that the continuous streams can obtain sufficient memory bandwidth, there are still two issues. The first issue is that the arbitration between different periodic streams has to be solved, and the second issue is that the worst-case response time for any individual periodic stream needs to be calculated. This worst-case response time directly determines the amount of on-chip buffering that is required to guarantee the throughput.

We have chosen a first-come-first-serve (FCFS) scheme for the arbitration of the continuous streams. This approach gives us relatively simple calculations of the worst case response time and the buffer requirements. For these calculations the following definitions are used. Let P be the set of periodic, continuous streams accessing the background memory. Let t_i be the minimum number of clock cycles between two memory requests from any stream $p_i \epsilon P$ and let c be the *maximum* number of clock cycles needed for a burst request from any stream to access the background memory. If N equals the number of clock cycles within a service cycle, then the *maximum* number of cycles M required for requests from the set P within one service cycle is

$$M = \lceil \sum_{p_i \epsilon P} (N/t_i) * c \rceil \quad (1)$$

The worst case response time of a background memory request can be calculated if we consider a *critical instance of the problem.* Consider τ_x, as shown in Figure 5, which is the moment within a service cycle at $N - R$ cycles from the start of the service cycle. If the CPU did not issue any requests in this service cycle until τ_x, then it can obtain access to background memory for a period of $2R$, if the requests continue in the next service cycle. If at τ_x all periodic streams issue a request at the same time when the CPU is granted a burst of requests of size $2R$, it can be seen that we have the critical instance during which the background memory is not accessible for continuous requests. In [6] it is proven that for all $p_i \epsilon P$, the worst case response time W of a memory request of a continuous stream equals

$$W = (c * |P|) + (\lceil (c * |P|)/(N - R) \rceil + 1) * R \quad (2)$$

In practical cases, we want R to be large to allow for large bursts from a CPU. In that case, as we can see from equation 2, the value of R is dominating in the calculation of the worst case response time. For a large value of R, equation 2 simplifies to $W_{large\ R} = c * |P| + 2R$. This can also be seen intuitively by looking at Figure 5. $|P|$ periodic requests at time τ_x will lead to a worst case response time of $c * |P|$ which must then be added to the $2R$ cycles occupied by random requests. This means that the worst case response time for all continuous streams is more or less the same regardless of the arbitration scheme between these streams. Earlier in this section we decided to use a first-come-first-serve scheme for the arbitration of continuous streams. We can now say that taking a simple arbitration scheme for the continuous streams can be done at no extra cost in terms of response times.

3 Buffering of continuous streams

Since we wish to keep the total amount of on-chip memory as small as possible, it is necessary to use the buffering as efficiently as possible. One option would be to give every stream its own buffer memory, in which case every buffer would require enough locations to handle the peak bandwidth demands. The peak bandwidth demands depend on the application. This can result in a very large amount of on-chip buffering as the following calculations will show.

For the video processing architecture from Figure 3 we can determine the number of buffers and the size of each buffer which we would require for a solution with separate buffer memories. For the CPA we have the case in which 20 individual streams require access to background memory. For the on-chip communication data streams each can have a peak bandwidth M_{peak} of 128 MB/s. For the background memory we use a standard 32-bit SDRAM running at 96MHz with burst size b of 64 bytes (16 words). The overhead of switching between reading and writing is on average 2 clock cycles, so an average of $c = 16 + 2 = 18$ clock cycles are required for every memory access of one burst. The length of a service cycle N has been set to 1024,

since for this order of magnitude a reasonable part of the CPU cache can be refilled in one burst and the amount of on-chip buffering is still acceptable. In [6] it is proven that the situation where $R = 1/2N$ is the corner case resulting in the largest buffer requirements.

For this example each stream has its worst case response time $W = 18 * 20 + (\lceil 18 * 20/512 \rceil + 1) * 512 = 1384$ clock cycles, in which for a stream with a peak bandwidth of 128MB/s, 1845 bytes can arrive. We do know that the total bandwidth of the background memory is 4 bytes∗96MHz=384MB/s which does not allow for all streams to use the peak bandwidth at the same time. However, since all individual streams can use this peak bandwidth at different applications, we have to give all streams the maximum required buffering for these circumstances. This means that for buffering the individual streams, 20*1845 bytes = 37kB of buffering will be required.

We wish to ensure that we can work with the average amount of buffering per stream rather than with the required peak amount for each individual stream. Therefore we use one large cache for all continuous streams to and from background memory.

Based on the worst case response time, the amount of buffering B for all streams can be calculated. It can be shown that the amount of buffering required for all strictly periodic streams is as follows:

$$B = |P| * (b + W * b/t_{ave}) \quad (3)$$

where t_{ave} is the average over t_i and where W as given in equation 2 is proportional to N. This formula shows that a trade-off can be made between the size of the service cycle N and the amount of on-chip buffering. A large value of N allows for a better average latency for random requests. This is obtained at the cost of additional on-chip memory for the continuous streams and possibly a larger deviation of the latency for the random requests in case of overload or saturation. The latter can be understood from Figure 5. When N is increased, M must increase proportionally to allow the periodic streams their required bandwidth. For an overloaded system there are more random requests than those that can be handled in a period of R cycles. This will result in a longer period in which no random requests are allowed access to background memory. Therefore we see that for this case a large period of N and therefore a large period of M cycles will result in a large latency for those random requests that remain pending when R cycles have already been consumed within a service cycle.

For our example with 20 streams we can calculate the required amount of buffering again if we take $R = 1/2N$ as the worst case situation. $W = 1384$ clock cycles and t_{ave} can be calculated if we consider that 20 streams share $N - R = 1/2N$ for their requests, and each request takes $c = 18$ cycles, so $t_{ave} = 1/40 * N * 18 = 460.8$ clock cycles. This means that $B_{ave} = 64 + 1384 * 64/460.8 = 256$ bytes. For 20 streams we therefore need 5kB in one memory rather than 37kB in 20 individual memories.

4 Cache fragmentation

In the previous section it was shown that using one cache memory instead of separate buffer memories is more advantageous. To guarantee the real-time constraints for all data streams, each stream must be able to claim the necessary amount of buffering at any time. To allow all streams to use the same cache memory and claim a certain amount of buffering, buffer locations must be allocated within the memory for the separate streams. Each individual continuous stream is assigned a number of cache lines. To determine the required number of lines we use equation 3 for one single stream: $B_i = b + W * b/t_i$ where t_i is the minimum inter-arrival time of requests which is determined by the real-time characteristics of the stream. These assigned cache lines can only be occupied by data for that particular stream, so they are *locked* if the cache is to be used for regular caching mechanisms as well. This locking of the cache lines will guarantee that the continuous streams will obtain the desired buffering. Since continuous streams will start and stop independent from the other streams, the locking of cache lines for a particular stream will also be independent of the locking for others. Because all streams are independent and because buffers in the cache memory are dynamically allocated and de-allocated, the cache memory will suffer from fragmentation. The run-time reconfiguration of each independent stream does not leave a singular moment when no stream is active. This means that there is no singular moment when the cache is not in use, therefore garbage collection or de-fragmentation will be difficult. To avoid the disadvantage of cache fragmentation an ordered list of non-assigned cache lines can be maintained. The initial situation is shown in Figure 6, where the HEAD of the list points to line $\#0$. If a number of cache lines has to be assigned to a

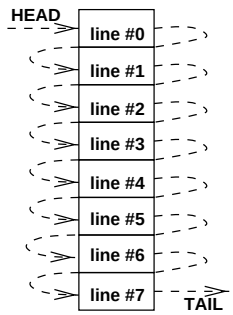

Figure 6. Linked list: initial state

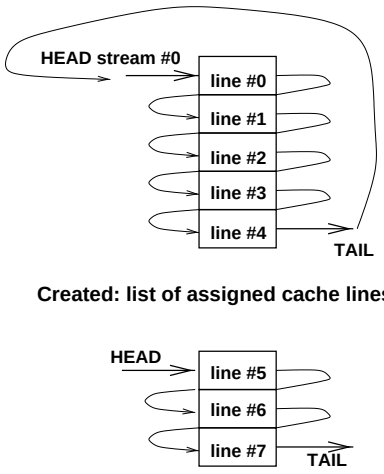

Figure 7. Linked list: lines assigned for stream $\#0$

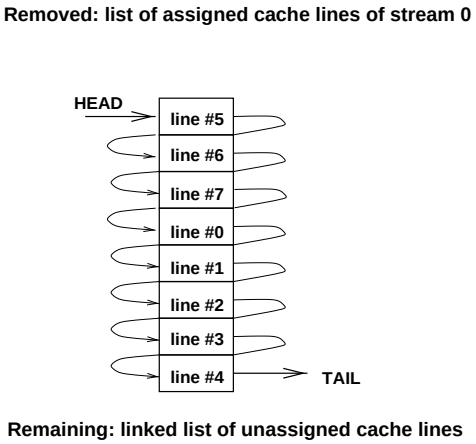

Figure 8. Linked list: released lines appended to list

certain stream, these lines are taken from the list, starting at the head of it. Subsequently, the new head of the list will now point to the new first cache line that is not assigned (Figure 7), and a new list of cache lines for this particular stream has been created. If assigned lines become unassigned, they can be either appended (Figure 8) or prepended to the list. The advantage of this method is that we can avoid difficult garbage collection schemes. The disadvantage is that only FIFO-based streams are easily supported in this mode of the cache with respect to (re)placement strategies. However, since we explained in Section 1 that for media processing FIFO-based behaviour is dominating, this special stream mode of our cache is not a disadvantage for the architecture.

5 Application: coprocessor array

We have applied both the arbitration scheme and the cache management scheme in the CPA shown in Figure 3. As mentioned in Section 1 the CPA shares an external background memory with a CPU and peripherals. The off-chip communication between the SDRAM and the memory arbiter uses arbitration at 3 levels, as shown in Figure 9.

Level 1 handles the arbitration between random requests and periodic requests. This arbitration uses the scheme given in Section 2. Level 2a decides between debugger requests and other random requests, normally giving priority to debugging. Level 3a splits the remaining random requests into two types: requests from the CPU and requests from the graphics accelerator (GFX). The value of a variable, *GFX priority*, can be used to ensure that the GFX unit can claim some requests from the CPU which in general will have higher priority. Level 2b uses the scheme from Section 2 again, to handle the stream requests which always occur at run-time, and the (periodic) control or instruction fetches which require a low latency. The latter type of requests are again subdivided at level 3b, where a difference is made between control requests which occur at configuration-time and run time parameter or instruction requests originating from specific video processing units requiring extra information at run-time. The arbitration

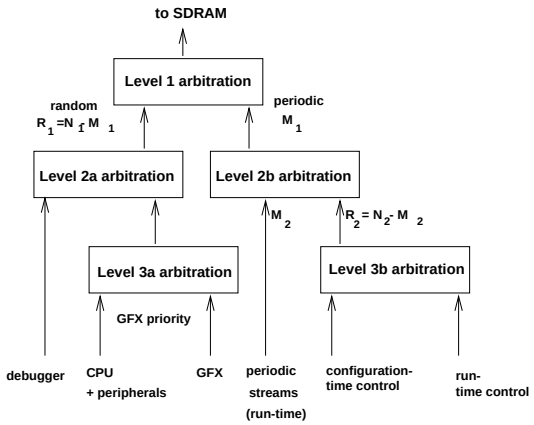

Figure 9. Arbitration at 3 levels for the CPA

scheme in Figure 9 is valid for our application domain because it gives all random requests a low latency, provided they do not overload the memory. The scheme also gives all periodic requests a guaranteed bandwidth, while at the same time it gives periodic requests at configuration-time such as programming parameters a lower latency if required. Figure

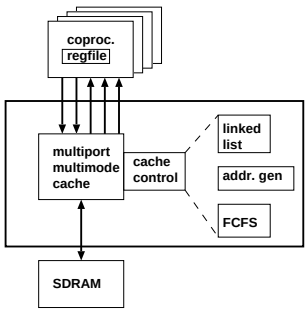

Figure 10. Stream cache and cache control of the CPA

10 shows the CPA instantiation of the stream cache and the cache control from Figure 3. To ensure that all video processors get sufficient bandwidth for transporting data to and from background memory, 5 parallel buses are connected to our cache. Small serial-to-parallel conversion buffers are used at the input, to convert from 16-bit words to 128-bit words. The reverse is done at the output with parallel-to-serial conversion buffers. The maximum number of streams which can use the cache is 20. The Linked List block performs the function explained in Section 4 and the FCFS block is responsible for the First-Come-First-Serve arbitration between all periodic streams. The CPA has been processed in a 0.35 μ technology. The cache memory is 3 mm^2, with an additional 3 mm^2 required for address generation and multi-stream accessing. The linked list requires 0.8 mm^2 and the FCFS unit which collects the requests from all 20 streams is 1.3 mm^2. Figure 11 shows the complete CPA layout. The area within the white box comprises the cache and its address generation, the linked list and the background memory arbitration.

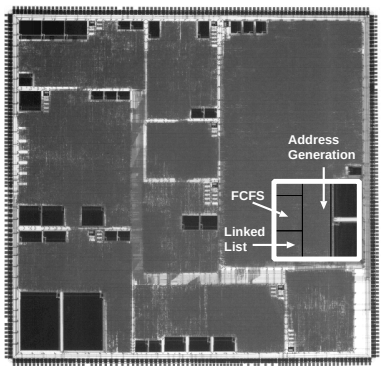

Figure 11. Layout of the CPA IC, comprising the arbitration scheme and caching

6 Conclusions

In this paper we have presented a memory arbitration scheme and a cache management scheme which have both been effectively used in a video processing architecture. The memory arbitration scheme can be used for systems where both continuous high-throughput and random low-latency requests are present. The cache management scheme is very effective for stream-based buffering of data. By using one cache memory for several independent data streams, a cost-effective solution has been obtained. The scheme allows for flexibility in the reconfiguration of applications while at the same time we can guarantee at compile-time that all run-time constraints will be met. This guarantee is obtained by using the calculated worst case response time to determine the required amount of buffering and by locking the corresponding cache lines for continuous streams. This means that none of the video processing units require a fall-back mechanism for cases in which the real-time constraints are not met. Both schemes offer a good scalability for increasing the number of processors. The memory arbitration scheme and the cache management scheme have both been used in a CoProcessor Array IC for video processing. This IC has been processed in a 0.35μ technology in which the total area for caching, cache control and memory arbitration is 8.1 mm^2.

References

[1] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, December 1996.

[2] S. Hosseini-Khayat and A. Bovopoulos. A simple and efficient bus management scheme that supports continuous streams. *ACM Transactions on Computer Systems*, 13(2):112–140, 1995.

[3] J. Leijten, J. van Meerbergen, A. Timmer, and J. Jess. Stream communication between real-time tasks in a high-performance multiprocessor. *Proc. Design, Automation and Test in Europe Conference*, pages 125–131, 1998.

[4] B. Lin, G. DeJong, C. Verdonck, S. Wuytack, and F. Catthoor. Background memory management for dynamic data structures intensive processing systems. *International Conference on Computer-aided Design*, November 1995.

[5] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[6] A. Timmer, F. Harmsze, J. Leijten, M. Strik, and J. van Meerbergen. Guaranteeing on- and off-chip communication in embedded systems. *Proc. IEEE Computer Society Workshop on VLSI '99*, pages 93–98, 1999.

[7] D. Zucker, M. Flynn, and R. Lee. A comparison of hardware prefetching techniques for multimedia benchmarks. In *Proceedings of the International Conference on Multimedia Computing and Systems*, June 1996.