technische universiteit eindhoven

Streaming memory consistency for efficient MPSoC design

Jan Willem van den Brand, Marco Bekooij and Arno Moonen

ES Reports
ISSN 1574-9517

ESR-2007-03
11 April 2007

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Streaming memory consistency for efficient MPSoC design

J.W. van den Brand[1], M. Bekooij[1] and A. Moonen[2]
[1] NXP Research, [2] Eindhoven University of Technology
contact: jan.willem.v.d.brand@nxp.com

Abstract

Multiprocessor systems-on-chip (MPSoC) with distributed shared memory and caches are flexible when it comes to inter-processor communication but require an efficient memory consistency and cache coherency solution.

In this paper we present a novel consistency model, streaming consistency, for the streaming domain in which tasks communicate through circular buffers. The model allows more reordering than release consistency and, among other optimizations, enables an efficient software cache coherency solution and posted writes.

We also present a software cache coherency implementation and discuss a software circular buffer administration that does not need an atomic read-modify-write instruction.

A small experiment demonstrates the potential performance increase of posted writes in MPSoCs with high communication latencies.

1 Introduction

In this paper, we consider heterogeneous Multiprocessor Systems-on-Chip (MPSoCs) with distributed shared memory (DSM) and caches. In DSM, a single shared address space is distributed over multiple physical memories. Processors communicate through shared memory.

A typical example of an architecture with a number of processors P with caches \$ and with multiple physical memories is shown in Figure 1. The processors communicate through an interconnect which can be a bus or a Network-on-Chip (NoC). Examples of real architectures with a similar structure are the TI OMAP platform [5], Philips Semiconductors' Viper [8] and Silicon Hive architectures [3].

Multiprocessor systems in which processors communicate through shared memory via caches require a cache coherency solution and a memory consistency model. Cache coherency assures that processors observe up-to-date data in the cache. We are not aware of efficient hardware cache coherency solutions for MPSoCs with NoCs. Therefore, we use a software cache coherency solution for such systems.

A memory consistency model defines the order in which memory operations from one processor appears to other processors. It affects both performance and programming model. Many consistency models have been proposed for high performance computers. Sequential consistency (SC) [15] is a model that allows no reordering of memory operations. No reordering is natural from a programmers point of view. Relaxing the ordering constraints enables pipelining of shared memory accesses, which can significantly improve performance, especially for high communication latencies.

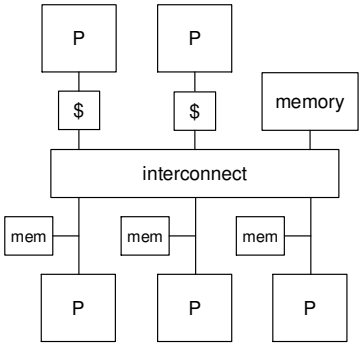


Figure 1. Multiprocessor system with background memory.

The release consistency model (RC) [11] relaxes many of the SC ordering constraints. RC requires a programmer to use acquire and release synchronization sections. It allows many hardware optimizations compared to SC [1].

In this paper, we propose a new consistency model, streaming consistency (StrC), which is targeted at the streaming domain. Examples of streaming applications are MPEG4 [7], Digital Radio Mondiale [13] and face recognition [14]. StrC allows more reordering and thus more pipelining than RC. Furthermore, it enables optimizations such as efficient software cache coherency and posted writes. These optimizations are desirable for MPSoCs, especially when a NoC interconnect is used.

The key contribution of this paper is the introduction of a new consistency model, StrC, which allows more reordering than RC and which enables optimizations.

This paper is organized as follows. In Section 2 we discuss related work. Then, in Section 3 we give a brief introduction to cache coherency and memory consistency. Existing consistency models and hardware solutions are discussed and we show why these solutions are not well suited for MPSoCs. In Section 4 we present the StrC model that fits such systems. In Section 5 we present software solutions for cache coherency and circular buffer administration. Section 6 shows the potential performance increase of StrC by means of an experiment. Section 7 concludes.

2 Related work

Memory consistency for MPSoCs is discussed in [17]. Release consistency (RC) is chosen as consistency model but it is not made clear why this model fits their purposes. Our consistency model allows more reordering than RC and enables several optimizations.

In [18], a heterogeneous architecture is presented to which applications modeled as Kahn Process Networks (KPN) are mapped. Buffers are mapped to background memory. The work allows caches to be placed in so called shells and argues, as we do, that a more efficient cache coherency mechanism is possible due to explicit communication. The used caches are dedicated to streaming data. We use a generic cache with minor adjustments that is used for streaming data as well as program data and instructions.

Experimental results in [10] show, in the context of high performance computing, that the performance gain of using relaxed models can be significant (10-40%) compared to strict models in systems with networks.

In [6], the coupling between memory abstraction and interconnect is discussed. Cache coherency for NoC based MPSoCs is identified as a challenging issue. Snooping and directory based solutions are mentioned as unlikely candidates. We also discard these hardware solutions (see Section 3.4). We use a software cache coherency solution.

Experiments in [1] show that performance of software cache coherency solutions in shared memory systems performs comparable to hardware solutions for a broad class of programs. It is shown that for well-structured programs the software based approaches out-perform hardware based approaches.

3 Cache coherency and memory consistency

In this section we give a short introduction to cache coherency and memory consistency. We give examples of potential coherency and consistency problems and we discuss existing consistency models. Finally we describe widely used hardware solutions for cache coherency and memory consistency and we explain why we think that they are not well suited for MPSoCs with NoCs.

3.1 Cache coherency

Processors in a cached shared memory multiprocessor environment can observe different data for the same memory address. This occurs when writes from one processor are not propagated to caches of other processors. For instance, in case of a write-back cache, write data goes by default to the cache and not to background memory. Without a cache coherency policy, this data is only visible for the processor that performed the write.

Caches that have a write through policy propagate all writes to shared memory. That does not mean that no cache coherency policy is required. On a read, the cache controller marks the fetched line associated to the read address as valid. The write of another processor to the same address is not visible to the first processor without a cache coherency policy.

We illustrate cache coherency with the following example. Consider the communicating tasks in Figure 2 which are mapped on two cached processors, P_1 and P_2. Note that the print instruction implies a read operation. First the task of P_1 writes to A. The value of A is propagated to shared memory if the cache of P_1 has a write through policy. Next, on the read action of P_2, its cache fetches the value of A from shared memory marking the associated cache line as valid and the task prints the result $A = 1$. Then P_1 writes another value to A which is again propagated to shared memory. However, on the next read, the cache of P_2 will return the old value of A, $A = 1$, because its line was marked valid.

A similar problem occurs if the cache of P_1 has a write back policy. The value of A would then not be propagated to shared memory in the first place and the cache of P_2 will read a unknown value from memory.

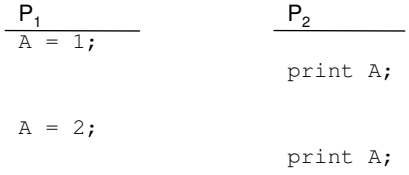


Figure 2. Cache coherency problem.

In a cache coherent system, data of all memory addresses that involve interprocessor communication must be observed with the same value for all involved processors. A cache coherency policy is necessary to assure this.

3.2 Memory consistency

Every multiprocessor system in which processors communicates through a shared memory should have a memory consistency model that defines how ordering of memory accesses are handled. Shared memory accesses can be conflicting and non-conflicting. Accesses to a shared address are said to be conflicting if they come from different processors and at least one of them is a write. The behavior of a program on a conflicting access depends on the order in which the accesses are observed between processors. In order to write a program that exhibits correct functional behavior, a programmer must know what the ordering behavior of the system is. Reordering of memory operations is only allowed if local program order of the processor is maintained (e.g a write followed by a read from the same address can never be reordered).

The memory consistency model of a shared memory system specifies the order in which memory operations will appear to execute to the programmer. Consider for example the tasks in Figure 3, taken from [4], where the ordering of operations influences functional behavior. The intention of this program is clear; processor P_1 communicates variable A to processor P_2 through shared memory and uses a flag (shared variable) for synchronization. The underlying assumption is that the write of A becomes visible to P_2 before the write to flag. If this assumption fails, the program will not fulfill the programmer's expectation.

The consistency model determines the programmer's view on shared memory and therefore has a major influence on the programming model.

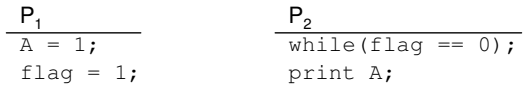


Figure 3. Event synchronization through a flag.

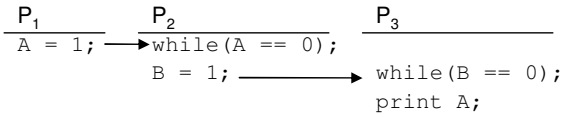


Figure 4. Importance of write atomicity for SC.

3.3 Existing consistency models

There are many different memory consistency models proposed in literature (see for instance [2, 4]). These consistency models target off-chip multiprocessor systems such as high performance computers with limited resource and energy consumption constraints.

Here we discuss sequential consistency [15] (SC), processor consistency [1] (PC) and release consistency [11] (RC). The models differ in the amount of reordering that is allowed. Reordering can enable pipelining of shared memory accesses. A model is relaxed compared to another model if it allows more reordering and thus enables more pipelining of shared memory accesses. More relaxed models are introduced to allow more pipelining.

SC requires program order to be maintained among all operations. Every task appears to issue complete memory operations one at a time and atomically in program order [4]. Writes issued by different processors appear in the same order to all processors (write atomicity). This is very natural from a programmer's point of view. For instance, the example from Figure 3 executes perfectly in a SC system. Unfortunately, this very intuitive model poses restrictions on hardware and compiler optimizations [2].

The PC model relaxes the ordering rules defined by the SC model. In this model, writes issued by a single processor are observed at another processor in the same order as they are issued. However, writes issued by different processors do not appear in the same order to all processors and therefore it does not satisfy write atomicity. It reflects the reality of complex interconnects where the latencies between nodes differ for different processor pairs. Because writes in the PC model are guaranteed to appear in order, the example in Figure 3 will work correctly. However, programs written with the SC model in mind are not guaranteed to work in a processor consistent system. The example in Figure 4 taken from [4] shows a program that can fail under PC but works under SC due to write atomicity. P_2 reads A which is written by P_1 and then writes B which in turn is used by P_3 as synchronization flag. Without write atomicity There is no guarantee that the latest version of A, $A = 1$, is visible to P_3 before $B = 1$ is visible to P_3. The wrong value is printed even if all processors receive write data in the order issued by the processors.

PC also allows reads that follow a write to a different address on the same processor to be reordered with respect to each other. Reads can be issued while a write is still in transfer.

RC, introduced in [11], is an even more relaxed model. For this model, shared memory accesses are categorized and different ordering requirements can apply to different categories. In RC terminology, a conflicting access is *competing* if there is a chance that a read and write occur simultaneously; otherwise it's a *non-competing* access. A conflicting access is made *non-competing* by using synchronization. There are two types of *synchronization* accesses; acquire and release.

A system is RC if the following conditions hold (taken from [11]):

(a) Before a non-competing load or store access is allowed to perform with respect to another processor, all previous acquire accesses must be performed and,

(b) before a release access is allowed to perform with respect to any other processor, all previous non-competing load and store accesses must be performed, and

(c) competing accesses (e.g. acquire and release accesses) are processor consistent with respect to one another.

The conditions give ordering requirements between non-competing and competing accesses and between competing accesses. There are no ordering requirements between non-competing accesses.

A program written with PC in mind is not guaranteed to work on a system with RC. The example of Figure 3 which executed perfectly under PC will have consistency problems because the shared memory accesses are not categorized and no synchronization is added to resolve undesired competing accesses.

The diagram in Figure 5 taken from [12] shows the ordering requirements for shared memory accesses to different addresses from a processor in a multiprocessor system for different consistency models. For each model, a program is shown with shared memory accesses. The arrows denote ordering constraints. SC allows no reordering, PC allows reordering of writes followed by a read. RC allows reordering of all shared memory accesses to different addresses outside the synchronization section and inside synchronization sections.

RC offers extensive pipelining possibilities. In Section 4 we present a consistency model targeted at streaming applications that is more relaxed than RC to allow even more pipelining.

3.4 Hardware solutions

Shared memory architectures that constantly monitor a bus for transactions (snooping bus) have an advantage when it concerns cache coherency and memory consistency solutions. The bus provides a global view on all memory operations. Caches can observe all memory transactions by snooping the bus and take appropriate action when a transaction takes place that concerns data in the cache. Also, ordering of reads and writes according to the selected consistency model can be assured by stalling bus transactions.

However, buses pose limitations on bandwidth. In practice no more than 16 processors are connected to a single shared bus. Moreover, we consider MPSoCs with NoC interconnects. Such interconnects do not provide global observ-

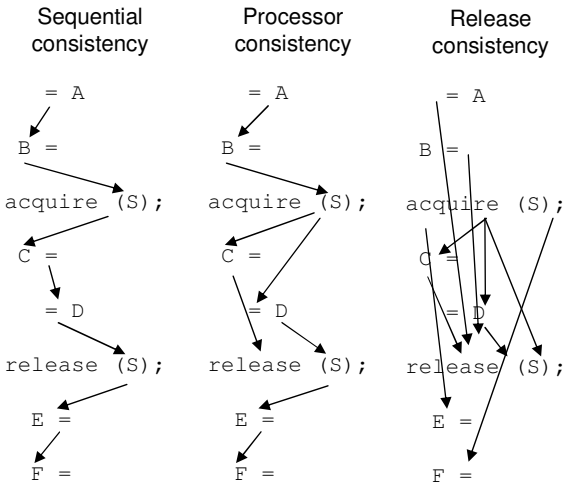


Figure 5. Comparison of three different consistency models.

ability of memory transactions. To the best of our knowledge, there are no snooping solutions for NoCs and it seems unlikely that efficient solutions will be found.

Directory based cache coherency approaches are better suited for network interconnects. In a directory based system, processors and caches assure that they do not violate cache coherency and memory consistency by issuing requests and notifications to a storage place (directory) that maintains state of all relevant transactions. For cache coherency, the directory is notified of all relevant changes of cache state by processors and is therefore capable of determining which cache contains the requested data.

The request and notification communication consume a significant amount of bandwidth, especially when a strict consistency model is used such as SC. The directory memory and control makes the hardware significantly more expensive than bus snooping hardware. Finally, the communication from cache to directory and back and then from cache to cache or from memory to cache introduces more latency. This latency results in additional processor stall cycles.

Therefore we conclude that both bus snooping and directory-based approaches are not well suited for MPSoC embedded systems with NoC interconnects. In the next section we present a consistency model that enables an efficient software solution.

4 Streaming consistency

The previous section discussed existing consistency models which were designed for high performance computers. This section presents a novel consistency model, streaming consistency (StrC), targeted at the streaming domain. It allows more pipelining than RC and enables optimizations such as an efficient software cache coherency solution which fits MPSoCs with NoCs. First we give a definition of StrC. Then, in Section 4.2 optimizations enabled by StrC are presented.

4.1 Streaming consistency model

StrC targets systems that run streaming applications. Interprocessor communication in such systems is performed by sharing units of data through circular buffers that are located in shared memory. These circular buffers can have multiple producers and consumers.

As for RC, StrC only has ordering constraints with respect to acquire and release calls. However, StrC associates these synchronization variables to circular buffers. A system is streaming consistent if the following conditions hold:

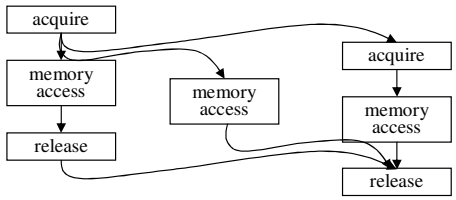
(a) before an access to a circular buffer b is allowed to be performed with respect to any other processor, the associated acquire access, $acquire(b)$, must be performed, and

(b) before a $release(b)$ access is allowed to perform with respect to any other processor, the access to the circular buffer b to which the release is associated must be performed, and

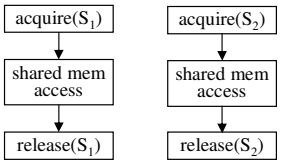
(c) acquire and release accesses for a certain circular buffer are processor consistent with respect to one another, and

(d) circular buffers are only allowed to be accessed within synchronization sections.

StrC allows reordering of synchronization sections that are associated to different buffers, i.e. such synchronization sections are allowed to overtake each other. This is different from RC where synchronization sections can overlap but can never overtake each other. RC conditions allow overlap as long as all accesses are performed before the following release and after the preceding acquire. Figure 6(a), taken from cite [11], shows the overlap possibilities for RC. Synchronization sections can never overtake preceding synchronization sections.



(a) RC



(b) StrC

Figure 6. Overlap possibilities for SC, RC and StrC.

The example in Figure 7 illustrates why these ordering constraints are crucial for RC. The example shows two processes, P_1 and P_2. P_1 writes a value to a outside a synchro-

nization section and writes a value to b inside a synchronization section. The RC conditions guarantee that the write to a is visible to other processors after the release. P_2 uses a and b. Availability of a after the release is guaranteed by the RC conditions. This kind of implicit synchronization is not allowed in StrC as every write or read to shared memory must take place inside a synchronization section and is associated to that section. Therefore, StrC has no ordering constraints between synchronization sections that are associated to different buffers as shown in Figure 6(b).

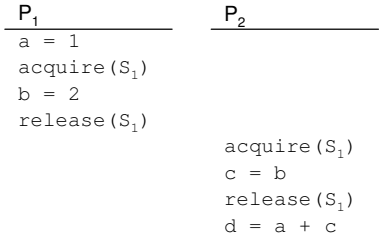
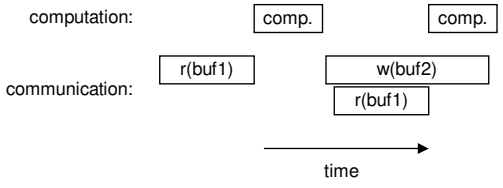


Figure 7. Implicit synchronization in RC.

An example of an application that can exploit the overtake possibility of StrC is shown in Figure 8. It shows a streaming application where data is read from buffer1 followed by a computation with the obtained data followed by a write of the new value to buffer2 over and over again. StrC allows us to completely parallelize the write to buffer2 with the read from buffer1 and the computation. On a RC system, the read can start before the write is released. RC does however not allow the release of the read before the release of the write.

```
while (true)
{
  acquire(buffer1);   //data available?
    a=read(buffer1);
  release(buffer1);   //release space
  b=computation(a);
  acquire(buffer2);   //space available?
    write(buffer2,b);
  release(buffer2);   //release data
}
```

(a) Code



(b) Overlap

Figure 8. StrC overlap example.

StrC relaxes RC. In StrC, the ordering constraints only have to be obeyed with respect to a certain circular buffer. A program written for StrC will run properly on a RC system because a program written for a more relaxed model executes properly on a system with a stricter model.

Concerning the programming model, StrC requires programmers to explicitly program shared memory communication through circular buffers in synchronization sections. Streaming applications expose this explicit communication. StrC does therefore not complicate programming.

Besides more pipelining possibilities, StrC also enables optimizations which are presented in the next section.

4.2 Optimization possibilities

StrC enables several optimizations. These optimizations require StrC and have additional requirements. In this section we discuss three optimizations: efficient software cache coherency, deterministic functional behavior and posting of writes.

4.2.1 Efficient software cache coherency

In Section 3.4 we discussed the limitations of hardware cache coherency solutions. StrC enables an efficient software cache coherency solution. Software cache coherency adds cache line flush calls after writes to shared memory and cache line invalidate calls before reads from shared memory. These calls can be added to acquires and releases. Invalidates of all shared addresses from which is read after an acquire must be executed before the reads that follow the acquire. Flushes of all shared addresses to which is written prior to a release must be executed before the following release.

RC allows programmers to program shared memory communication everywhere in a program. Every memory access is potentially a shared memory access. Shared memory accesses are not directly linked to acquire and release calls which makes it hard to determine which cache lines should be invalidated on acquire and flushed on release.

In StrC, shared data is communicated in synchronization sections through circular buffers that are linked to these sections. Therefore, flush and invalidate calls can easily be added to acquire and release calls.

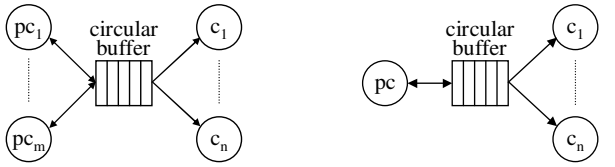
Caches communicate at the granularity of cache lines which typically consist of multiple words. Multiple processors can access different words in the same cache line. This is known as false sharing. Flushing of a line by one processor affects data in shared memory for another processor. False sharing can be eliminated by not allowing multiple processors to access different words of the same cache line.

For StrC, false sharing can be prevented by only allowing a single circular buffer per cache line. This solution leads to inefficient cache use if shared data is scattered in shared memory because this leads to many partially used cache lines. Therefore a circular buffer should be located in a consecutive address range. For RC, it is difficult to prevent false sharing because shared accesses are not explicit in the program.

In Section 5 we discuss a software cache coherency solution.

4.2.2 Deterministic functional behavior

The functional behavior of a task often depends on the received data. We assume that tasks that run on different processors are scheduled independently. The circular buffer in Figure 9(a) has multiple producing and multiple consuming tasks. All consuming tasks access the same data before space is released. What data is put in the buffer by the producers in what order depends on the execution order of the producing tasks and on the communication latency (i.e. data from a task that is scheduled earlier than another task can arrive later at the buffer then data from the second task due to communication latency). This limits the extend to which one can reason about the functional behavior of an application at design time.



(a) Multiple producers (b) Single producer

Figure 9. Number of producers and consumers per circular buffer.

Reasoning about functional behavior at design time is enabled for the StrC model by only allowing a single producer per circular buffer as depicted in Figure 9(b). For RC, it is hard to determine whether a shared address has multiple producers because shared accesses are not explicit.

4.2.3 Posting of writes

A posted write is a write that allows operations that follow the write to be executed without confirmation of the write.

The SC model does not support posting of writes because reordering of memory transactions is not allowed. A write has to be completed before the following operation is allowed to execute.

For RC, all writes that precede a release call have to be finished before the release call is allowed to finish because writes are not allowed to be reordered with respect to a following release. In order delivery at shared memory of writes to shared data and the write to the synchronization variable (i.e. FIFO behavior) is sufficient. However, RC says nothing about where the different addresses that is being written to are physically located. In a DSM system, it is likely that these addresses are distributed over different physical memories. If this is the case, in order delivery of writes to shared data and the synchronization variable is hard to guarantee. In this case RC requires writes to be confirmed and does not support posted writes.

For StrC, a synchronization section is associated to a circular buffer. If the circular buffer is located in the same memory as its synchronization variable, in order delivery is feasible. StrC then supports posted writes.

Furthermore, a write followed by a read from the same address requires a system to assure that the write has performed before the read in order to continue with the read. RC allows a combination of reads and writes to the same shared address. Therefore, a mechanism such as a write buffer is required to assure that correct data is returned on a read following a write.

StrC also allows a combination of reads and writes from and to the same circular buffer in a synchronization section. Special hardware support such as write buffers is not necessary if we do not allow a combination of reads and writes from/to the same circular buffer inside a synchronization section.

5 Software solution

In this section we present a software cache coherency solution. This approach is enabled by StrC, presented in the previous section. As opposed to hardware bus snooping and directory based solutions described in Section 3.4, this solution is well suited for NoC based MPSoCs.

We also discuss a solution for implementing the circular buffer administration in software.

5.1 Cache coherency

In StrC, the interprocessor communication is more explicit than in RC. It is known what address range is communicated and the moments before and after communication of shared data are clearly marked by acquire and release.

This explicit communication allows us to add invalidate and flush instructions when buffer data is read or written. On an acquire, before data is read, all cache lines that contain old data for the involved shared addresses are marked as invalid. This ensures that the latest data is observed. The software cache coherency approach flushes cache lines that contain data of writes to shared memory before the release call to ensure that the shared memory has the most recent copy.

The approach requires caches to have a means to invalidate cache lines by address. Such functionality is present in many caches, ARM11 and TriMedia are examples of cached processors that support invalidation and flushing of cache lines.

5.2 Circular buffer administration

We use a software circular buffer administration to assure that no data is read if no data is available and that no data is written if no space is available. For this purpose, we use the C-HEAP protocol [9, 16]. In this protocol a buffer administration is maintained for number of reads and writes from and to the buffer (read counter and write counter). No hardware support is needed and no atomic read-modify-write operation is required. The read and write pointers are increased after data is read or written. Wrap around of a pointer occurs if a pointer would fall outside the memory range of the buffer. The amount of data or space in a buffer can be deduced from the values of the read and write pointer and the size of the buffer.

Figure 10 shows the steps that are performed on a buffer write and read for software cache coherency (see previous section) and circular buffer administration. A write starts by polling the buffer administration to check whether there is space available in the buffer (1). If this is the case the data is written (2). The data is flushed from the cache after it is written (3). Then, the buffer administration is updated (4). A read to a buffer starts by checking whether there is data in the buffer (5). Next the cache lines that are associated with the data are invalidated (6). Then the data is read from shared memory (7) and the buffer administration is updated (8).

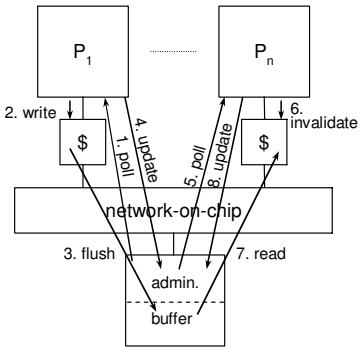


Figure 10. Software cache coherency and memory consistency.

Performing cache coherency instructions adds cycles. However, software approaches remove hardware cost by transferring the cost of detecting coherency problems (protocol overhead) from hardware to software.

6 Experimental results

In this section we show the potential performance increase of using the StrC model by means of an experiment. We focus on the posted write optimization from Section 4.2.3. The experiment shows the negative effect of NoC latency for systems that do not allow posting of writes. SC does not support posting of writes and RC does not support it by default.

A small producer consumer example is used as test application. Each tasks runs on its own ARM7 processor with local memory. The processor runs at 100 MHz. The produced token (data) is put in a buffer that is located in the local memory of the consumer processor. We measure the execution time with and without posed write. In the system without posted write, each write has to be acknowledged .

Figure 11 shows a bar-plot of the execution times of the producer with and without posted write for different NoC latencies and for two different token sizes: 16 and 4 words. The NoC latency is given for the forward connection from producer to consumer. In case of acknowledged writes, the latency of the reverse connection is the same as that of the forward connection. Note that communication from producer to consumer is word-based and that every word has to be acknowledged.

For the system without posted writes, the execution time of the producing task grows linear with NoC latency and message size. The producer is slowed down because it has to wait for acknowledges. For the system with posted writes, NoC latency has no effect and the token size has a minor effect (additional instructions for sending additional words to the consumer).

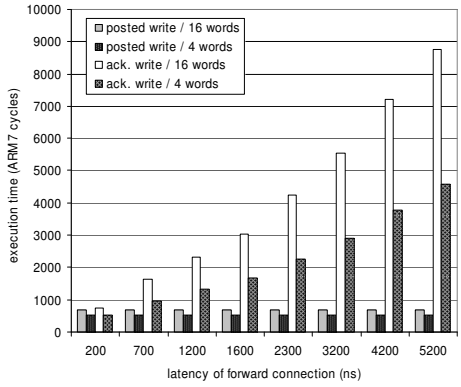


Figure 11. Normalized execution time of SC and StrC for different network latencies.

The experiment shows that posted writes are crucial in realistic MPSoCs with NoCs. Posted writes are always possible in a StrC system if synchronization sections do not contain a combination of reads and writes.

7 Conclusion

Consistency models influence both performance and programming model. A multiprocessor system should support a consistency model that fits the targeted domain.

We have presented the streaming consistency model (StrC) which fits the streaming domain. StrC does not complicate programming. As opposed to RC, StrC allows synchronization sections to overtake each other and thus allows more pipelining.

Besides more pipelining possibilities, StrC also enables optimizations. The efficient software cache coherency solution enabled by StrC overcomes the limitations of well known hardware solutions (bus snooping and directory based). StrC allows reasoning about functional behavior if there is no more than one producer per circular buffer. Posted writes are supported by StrC with little effort.

Our producer-consumer experiment shows that these posted writes are desirable in MPSoCs with NoCs, especially for high communication latencies. Without posted writes, the producer is severely slowed down because it has to wait for acknowledges.

References

[1] S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of hardware and software cache coherence schemes. In *Proc. Int'l Symposium on Computer Architectures*, pages 298–308, May 1991.

[2] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec 1996.

[3] G. Burns, M. Jacobs, M. Lindwer, and B. Vandewiele. Silicon hive's scalable and modular architecture template for high-performance multi-core systems. In *Proc. Int'l Signal Processing Conference and Expo*, 2005.

[4] D. Culler and J. Singh. *Parallel Computer Architecture*. Morgan Kaufmann, 1999.

[5] P. Cumming. *Winning the SoC Revolution*, chapter The TI OMAP Platform Approach to SoC. Kluwer, 2005.

[6] G. De Micheli and L. Benini. *Networks-on-Chip: Technology and Tools*. Elsevier, 2006.

[7] K. Denolf, A. Chirila-Rus, and D. Verkest. Low-power MPEG-4 video encoder design. In *Proc. Workshop on Signal Processing Systems Design and Implementation*, pages 284–289, 2005.

[8] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital TVsystems. *IEEE Design and Test of Computers*, 18:21–31, 2001.

[9] O. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In *Int'l Symposium on System Synthesis (ISSS)*, pages 1–6, Oct 2001.

[10] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory memory multiprocessors. In *Proc. of the Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 245–257, Apr 1991.

[11] K. Gharachorloo, D. Lenoski, J. Ludon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. Int'l Symposium on Computer Architectures*, pages 15–26, Jun 1990.

[12] J. Hennesy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 2nd edition, 2003.

[13] F. Hofmann, C. Hansen, and W. Schfer. Digital radio mondiale (DRM) digital sound broadcasting in the AM bands. *IEEE Trans. on Broadcasting*, 49:319–328, 2003.

[14] V. Kianzad, S. Saha, J. Schlessman, G. Aggarwal, S. Bhattacharyya, W. Wolf, and R. Chellappa. An architectural level design methodology for embedded face detection. In *Int'l Workshop on Hardware/Software Codesign (CODES)*, pages 136–141, 2005.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28:690–691, Sept 1979.

[16] A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Busa, K. Goossens, R. Llopis, and P. Lippens. C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. In *Proc. Int'l Conference on Hardware Software Codesign (CODES)*, pages 206–217, Sept 2004.

[17] F. Petrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures. In *Proc. Euromicro Conference on Digital System Design (DSD)*, Aug 2006.

[18] M. Rutten, J. van Eijndhoven, and E. Pol. Caching techniques for multi-processor streaming architectures. In *Workshop on Media and Signal Processors for Embedded Systems and SoCs (MASES)*, Sept 2004.