Streaming consistency: a model for efficient MPSoC design

J.W. van den Brand and M. Bekooij
NXP Research
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.

Keywords: streaming, memory consistency, cache coherency, MPSoC, NoC

Track, topic area: T1 Systems-on-a-chip/in-a-package, multi-processors

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.

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.

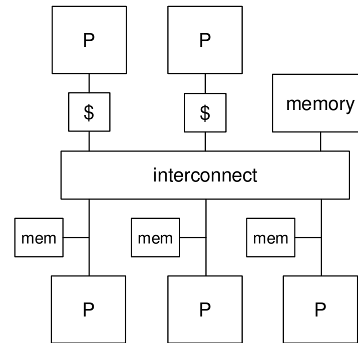


Figure 1. Multiprocessor system with background memory.

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. This is natural from a programmers perspective. Relaxing the ordering constraints enables pipelining of shared memory accesses, which can significantly improve performance, especially for high communication latencies.

The release consistency model (RC) [10] 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.

10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)
0-7695-2978-X/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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 [11] 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 software cache coherency solutions perform 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.

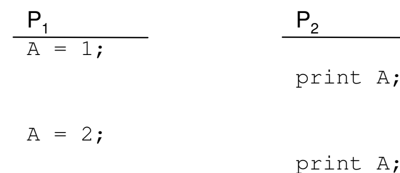


Figure 2. Cache coherency problem.

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

IEEE COMPUTER

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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.

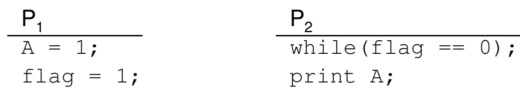


Figure 3. Event synchronization through a flag.

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 [10] (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.

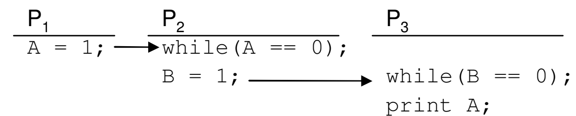


Figure 4. Importance of write atomicity for SC.

RC, introduced in [10], 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 [10]):

(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-

IEEE COMPUTER

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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.

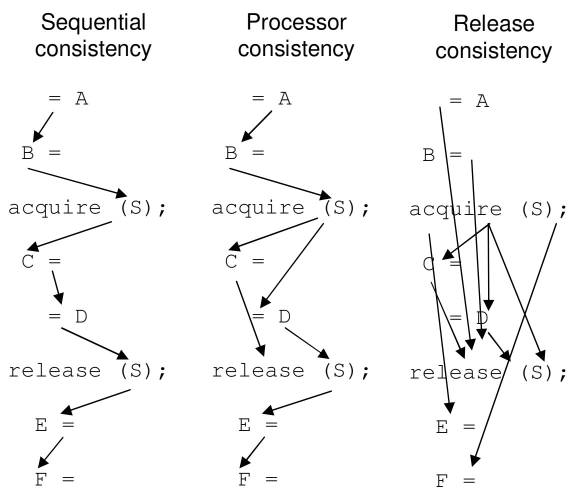


Figure 5. Comparison of three different consistency models.

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 observability 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. Inter processor 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

IEEE COMPUTER SOCIETY

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

(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 circular buffer are processor consistent with respect to each other, 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 [10], shows the overlap possibilities for RC. Synchronization sections can never overtake preceding synchronization sections.

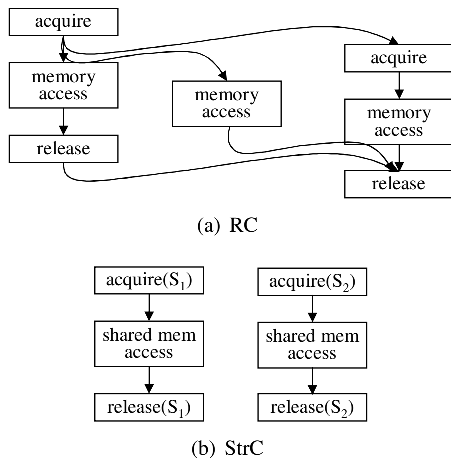


Figure 6. Overlap possibilities for SC, RC and StrC.

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 synchronization 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).

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 fol-

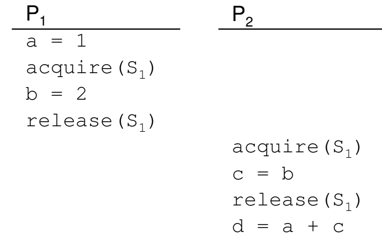


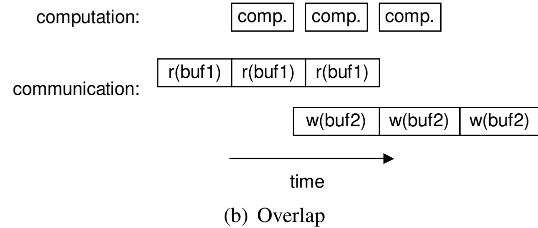
Figure 7. Implicit synchronization in RC.

lowed by a computation with the obtained data followed by a write of the new value to buffer2 over and over again. The read from buffer1 for the second iteration can start immediately after the first read. The read from buffer1 can be pipelined with the computation and the write to buffer2.

On a RC system, the acquire of buffer1 for the second iteration has to wait for the acquire of buffer2 of the first iteration. Also, buffer1 can not be released before buffer2 is released. This limits the pipelining possibilities.

```
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.

IEEE COMPUTER SOCIETY

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.