A tuneable software cache coherence protocol for heterogeneous MPSoCs

Ophelders, F.E.B.

Award date:
2009

Master's Thesis

A Tuneable Software Cache Coherence Protocol for Heterogeneous MPSoCs

Frank Ophelders
April 2009

Eindhoven University of Technology
NXP Semiconductors

Supervisors:
Prof. dr. Henk Corporaal (Eindhoven University of Technology)
Dr. ir. Marco Bekooij (NXP Semiconductors)

Abstract

In a multiprocessor system on chip private caches introduce the cache coherence problem; because processors can have an incoherent view on the shared memory because of stale data in their caches. Cache coherence protocols have been designed to ensure a coherent view on the background memory. Both hardware as well as software solutions have been introduced in the past, and hardware solutions can be roughly divided in two approaches. The first is a snooping-based approach, which relies on global visibility of writes and write atomicity, and is therefore difficult to apply efficiently in combination with a network-on-chip. The second is a directory-based approach, which has as a major drawback the increased memory access latency. This increase is a result of consulting the directory upon memory accesses.

Software cache coherence protocols have the potential to ensure cache coherence without requiring global visibility of writes and without increasing the memory access latency. Therefore software cache coherence protocols can be a candidate to be used in combination with a network-on-chip.

We propose a software cache coherence protocol that can be applied in a heterogeneous MPSoC with a network-on-chip. This cache coherence protocol ensures that caches are coherent on synchronization points, which is sufficient to support Release Consistency, on top of which standard communication libraries, f.i., Pthreads can be implemented. Our software cache coherence protocol can be applied to heterogeneous systems without modifying the caches, as long as processors are able to control their cache through *clean* and *invalidate* instructions.

Our software cache coherence protocol is implemented on an MPSoC with two ARM9 processors, which has been mapped on a Xilinx Virtex 4 FPGA. Several Splash2 applications are used for the experimental performance evaluation; therefore we have implemented Pthreads calls. We demonstrate that putting shared data in a separate address range, and coupling data structures and synchronization points, can significantly improve the performance. From experiments we conclude that for standard problem sizes in the Splash2 benchmark set the protocol overhead is small compared to the computation time. The speedup observed, for the applications on a two processor MPSoC, is between 1.89 and 2.01.

Acknowledgements

I would like to thank my supervisors; Marco Bekooij, for giving me the opportunity to work at NXP Research and for guiding me with his own dynamic, effective and creative way during my work; Henk Corporaal for his suggestions and the active interest for the results of this project. I feel that the meetings and the interesting discussions involved greatly helped in achieving our goal and helped to improve my engineering skills and my reasoning.

Marco has always showed his interest in the issues involved in the project and he was quite involved in the project by staying responsible for the hardware throughout the entire project. I have really appreciated all the effort he put into the project, which kept me motivated at times when extensive debugging was required, or when the problems appeared to be unsolvable.

Furthermore, I want to thank Ben Juurlink for showing his interest in the project and being part of the Assessment Committee.

Finally I want to thank the people working in the SoC Architecture and Infrastructure group at NXP Research, for the enjoyable times, like the daily lunch and coffee breaks, during my project.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Problem description

For many years processor performance has been improved successfully by higher clock frequencies, exploitation of instruction level parallelism, and better memory hierarchies. However, because of a limited amount of instruction level parallelism in applications, it becomes important to exploit task level parallelism as well.

Exploitation of task level parallelism creates some challenges; first of all it is difficult for human beings to reason about parallel processing. Secondly, people still tend to apply successful techniques from the uniprocessor domain in the multiprocessor domain. However it is not always possible to, without modifications, successfully apply techniques from the single processor domain in multiprocessor systems. A well known example is that caches introduce *cache coherence* issues in the multiprocessor domain. A related issue in multiprocessor system on chip (MPSoC) architectures is *memory consistency*, and it is important to understand both *memory consistency* and *cache coherence*.

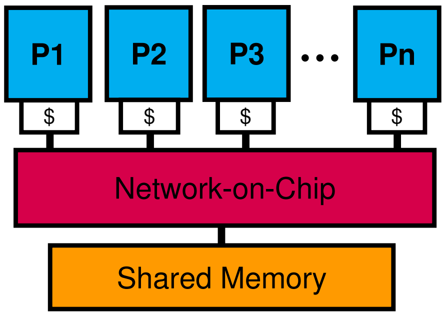


Figure 1.1: Example shared memory MPSoC

First, let's consider an example of a shared memory MPSoC, shown in Figure 1.1. Each processor is connected to the shared memory through caches and communication between processors is through shared memory. In a cached shared memory MPSoC it is possible to have multiple copies of a single location of the shared memory in different caches. Clearly, a value of a location is stored in the shared memory, but a copy of this value may exist in each cache. If a processor modifies the value at the memory location, the write, if no measures are taken, will not become visible to other caches. Those caches may still contain stale data. This is called the *cache coherence* problem. This problem can be avoided by using a *cache coherence protocol* which guarantees that writes will eventually become visible to all processors.

Cache coherence protocols can be divided in two classes, *hardware based* and *software based.* Hardware based cache coherence protocols are subject to research since the 1980's and we distinguish between two different approaches. The first is the *snooping based* approach and the second is the *directory based* approach.

A *snooping-based* cache-coherence protocol relies on all caches to "snoop" the interconnect and take appropriate actions based on transactions on the interconnect [10]. For instance, all caches having the value of a memory location X, invalidate their copy if another processor writes a value to location X. However, for popular snooping protocols such as MSI, and MESI [10], to function correctly the MPSoC needs to support two properties [10]. First of all, all memory accesses should be visible to all processors. Second, all memory accesses should appear in the same order. These properties are easily supported in an MPSoC with a bus, because of the nature of the bus. However, in an MPSoC with a network-on-chip (NoC) it is difficult to support these properties. A NoC handles memory accesses as transactions in parallel, consequently, memory accesses are not observed by other processors. In addition, processors can observe different latencies to memories, which makes it difficult to guarantee one single order of writes being observed by all processors.

A second hardware approach is the *directory-based* cache-coherence protocol [10], which is based on a centralized directory that stores information about the status of caches for each location in the shared memory. The idea is as follows; a centralized directory is consulted before writing to a shared location X, and the directory stores information about all processors having the shared location X cached. On a write of a processor P to X, the directory will respond with sufficient information to ensure that all other caches invalidate their copy of X, i.e., ensuring that the next time any processor reads X, the latest value will be read. This type of hardware cache coherence protocols is believed to be a scalable solution for MPSoCs with a NoC. However, according to [24] the memory overhead of the directory can reach up to 20% of the total memory. Additionally there will be an increase in traffic due to consulting the directory. This can also lead to contention, because all memory accesses to a memory location X require consulting the same directory, even if the directory is physically distributed. Additionally, the time spent in sending and receiving transactions to and from the directory is added to the memory access latency.

Both hardware cache coherence protocols require support from all caches in the MPSoC. The design complexity of integrating heterogeneous processors on MPSoCs is not trivial since it introduces several problems in both design and validation due to different bus interface specifications and incompatible cache coherence protocols [36]. An example of a hardware/software methodology to ensure cache coherence in heterogeneous MPSoC is proposed in [36] but, it can only be applied when *all* caches support hardware cache coherence.

Software cache coherence protocols rely on processor instructions to clean and invalidate cache lines. As a consequence different processors can be supported, as long as instructions to clean and invalidate the cache are provided. Software cache coherence protocols have the potential to be a scalable cache coherence protocol that can be applied to MPSoCs with a NoC, because software cache coherence protocols do not require global visibility of writes or require one single order of memory accesses being observed by all processors. Furthermore, software cache coherence protocols do not increase the memory access latency, because a directory is not consulted on memory accesses.

Most software cache coherence protocols rely, to the best of our knowledge, on explicit synchronization. In particular, the caches are guaranteed to be coherent on synchronization points. This poses the restriction that our MPSoC is limited to executing software with explicit synchronization, but we expect that this does not significantly restrict the applicability of the software cache coherence protocol, as most parallel programs rely on synchronization to guarantee correct behavior.

The hardware implementation cost of a software cache coherence protocol can be significantly less than the implementation cost of a hardware protocol. Additionally we expect that verification of our software cache coherence protocol is easier than the verification of existing hardware cache coherence protocols, which has been subject of many research projects [6, 30, 26, 35].

Cache coherence is important if data is being shared between multiple processors in an MPSoC, because it ensures that writes to a location will eventually become visible to other processors. However, in parallel programs it is usually expected that a read returns the value of a particular write, in other words, an order between memory accesses to different locations exists. This order is not implied by cache coherence, and as a result there is a need for a *memory consistency* model, which poses constraints on the order in which memory accesses have to be completed and have to become visible to other processors. This includes order between accesses to the same or to different locations, where the accesses may be performed by different processors. As a consequence, memory consistency subsumes cache coherence [10].

1.2 Contribution

This thesis presents a tuneable software cache coherence protocol that is highly suitable for heterogeneous MPSoCs with a NoC and with off-the-shelf processors. The software cache coherence protocol ensures that caches are coherent on synchronizations, which is sufficient to support Release Consistency (see Section 2.2.4), on top of which standard communication libraries, e.g., Pthreads and OpenMP can be implemented. More specifically, we have embedded the protocol in several Pthreads calls.

The software cache coherence protocol is evaluated in an ARM926EJ-S MPSoC which is mapped on an FPGA. Several applications from the SPLASH2 benchmark set [41] are executed in parallel on the MPSoC. The overhead of the protocol can be surprisingly low, because the speedup for most SPLASH2 applications is between 1.89 and 2.01 on a two processor MPSoC.

In addition to the software cache coherence protocol we have identified several optimizations to increase the performance of the protocol. Firstly, it is important to provide separate address ranges for private and shared data. As a consequence, private data and shared data can be cached more efficiently, and cache coherence operations can be limited to the shared address range. Secondly, for some applications it may be beneficial to provide a specific programming model which can further improve the efficiency of the software cache coherence protocol. A suitable programming model would be restricting interprocessor communication to sharing data through First-In-First-Out (FIFO) buffers.

Furthermore, the software cache coherence protocol is designed to be applicable in a *predictable* and *composable* MPSoC. *Predictability* discusses to what extent performance guarantees of threads can be given at design time. *Composability* is a property that ensures that a thread can not impact the execution of *any* other thread. Both predictability and composability require the cache coherence operations to be interruptible; therefore task switches can not be postponed indefinitely by cache coherence operations. In addition, cache coherence operations are local, consequently cache coherence operations on one processor do not impact the execution of threads on other processors, whereas invalidation request messages in hardware cache coherence protocols can impact the execution of threads on other processors.

1.3 Organization

The rest of this report is organized as follows. Chapter 2 discusses cache coherence and memory consistency in more detail. Following this, Chapter 3 gives an overview of related work in cache coherence protocols; hardware protocols will be briefly discussed, and several related software approaches will be discussed. Chapter 4 provides a description of the hardware and software platform on which the protocol is implemented; this platform is used as experimental setup. Chapter 5 presents the software cache coherence protocol to ensure a Release Consistent MPSoC. Chapter 6 discusses issues in predictability and composability, related to our MPSoC. Chapter 7 explains the experiments to assess the performance of the protocol. Chapter 8 discusses suggestions to improve the cache controller. Lastly, concluding remarks will be given in Chapter 9 and Chapter 10 discusses directions for future work.

2 Cache coherence and memory consistency

Cache coherence and memory consistency are challenging issues in shared memory multiprocessor systems. In the first part of this section we will discuss cache coherence issues, followed by explaining its relation to memory consistency. This report proposes a software cache coherence protocol, which ensures cache coherence, but as we will illustrate, memory consistency is related and it is important for reasoning about outcome of a program. Memory consistency will be discussed by describing several memory consistency models; each model has its constraints on the software, hardware, and optimization options.

2.1 Cache coherence

This section discusses *cache coherence* in MPSoCs. Caches have been introduced in uniprocessor systems to reduce average memory access latency. Intuitively a memory with a cache hierarchy holds values, and on a read, a memory returns the last value written to it. We would like to apply this intuitive model also to a shared memory in an MPSoC. Unfortunately if in an MPSoC processors have a private cache, without taking precautions, danger exists that one may see stale values in its cache. This is called the *cache coherence problem*, which is discussed in [10, 19]. The cache coherence problem is illustrated in Example 2.1, which is related to Figure 2.1. However, before discussing the cache coherence problem we will provide background information on caches and cache maintenance operations.

A cache is used to temporarily store copies of memory locations. These copies are stored in lines of a certain number of bytes in the cache, and these lines can be read and modified. A *write-through* cache has the property that if a processor modifies the line, the write will also be propagated to the memory. If only one word is modified, then only the modified word will be updated in the memory. This type of cache can cause a lot of write traffic because all writes are sent to the memory. A *write-back* cache stores copies of memory locations, and upon a write only this copy will be updated. The write is *not* sent to the memory, but will become visible to the memory if a *clean* operation is called, or if the line is replaced.

Cache maintenance operations, such as *invalidate* and *clean* a cache line, are used to ensure cache coherence. Assume a processor *P1* executes an *invalidate* of line X; this operation ensures that the next read of X by *P1* is a cache miss, and consequently the read will return the value of X which is stored in the shared memory. The second cache maintenance operation, *clean* a cache line, is slightly different. Let's assume *P1's* write-back cache holds a location *X* and *P1* writes to location *X*. This would result in a write to the cache, but the shared memory would not be updated. A *clean* operation of the line holding *X* would cause the contents of the line to be copied to the shared memory, i.e., updating the shared memory.

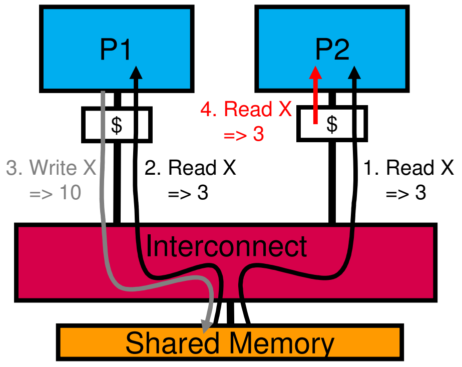


Figure 2.1: Example cache coherence problem

Example 2.1. *See Figure 2.1. Two processors with write-through caches are connected through, e.g., a bus or NoC to shared memory. First (1), processor* P2 *reads a value from shared memory at memory location* X*, i.e., copying the value to its cache. Then (2), processor* P1 *reads memory location* X*. Then (3), processor* P1 *changes the value from 3 to 10. Because* P1's *cache is a write-through cache the write will also update shared memory. However, when* P2 *reads* X *again (4) its cache will return 3 instead of the correct value 10, which is only valid in* P1's *cache and the shared memory. This is a* cache coherence *problem. With write-back caches the situation is slightly different. Because then only the cache of* P1 *would contain the correct value for* X*, and the shared memory will only be updated when* P1's *cache replaces the line holding* X*, or the line holding* X *is cleaned. Until the shared memory is updated all processors reading from the shared memory would read old invalid data.*

Example 2.1 showed a situation where processors could read stale data and we need to find a method to guarantee cache coherence. Let's first define the conditions that have to be satisfied to obtain cache coherence.

- A read made by a processor *P* to a location *X*, that follows a write by the same processor *P* to *X*, with no writes to *X* by another processor occurring between the write and the read instructions made by *P*, must always return the value written by *P*. This condition is related with *program order preservation*, and this must be achieved even in uniprocessor architectures.
- A read made by a processor *P1* to location *X* that follows a write by another processor *P2* to *X* must return the written value made by *P2* if no other writes to *X* made by any processor occur between the two accesses. This condition defines the concept of coherent view of memory.
- Writes to the same location must be sequenced. In other words, if location *X* received two different values *A* and *B*, in this order, by any two processors, the processors can

never read location X as B and then read it as A. The location X must be seen with values A and B in that order.

These conditions result in two properties in cache coherence: *write propagation* and *write serialization*. *Write propagation* means that writes become visible to other processors. *Write serialization* means that all writes to a location (from one or multiple processors) are seen in the same order by all processors. It is important to note that *write serialization* is concerned about the order of subsequent writes to one single location.

2.2 Memory consistency

Cache coherence is essential if information is to be transferred between two processors, where one processor writes to a location and the other processor reads. Eventually, the value written to the location should become visible to the reader. However, coherence says nothing about *when* the write will become visible. Often in parallel programs, the programmer expects that a read returns the value of a particular write; as a result, there should be a known *order* between a write and a read. Consider the example in Algorithm 1 where the program synchronizes through flags. Clearly the programmer intends process *P2* to spin idly on *flag* as long as process *P1* hasn't set *flag* to 1. To get the expected result from $P2$, we assume that the write to A becomes visible to $P2$ before the write to $flag$. However, order between accesses to *different* locations is not implied by coherence.

Algorithm 1 Example synchronization through flags

```
Require: $A$ and $flag$ are initially 0

P1                          P2
  $A \leftarrow 1$                    while $flag == 0$ do
  $flag \leftarrow 1$                   skip
                              end while
                              print $A$
```

Clearly, we need more than coherence to give a shared address space clear semantics. We need an *ordering model* that programmers can use to reason about possible outcome and correctness of their programs.

A *memory consistency model* specifies constraints on the order in which memory operations must appear to be performed (i.e., to become visible to processors) with respect to one another. This includes operations to the same locations or to different locations and by the same processor or different processors. As a consequence, memory consistency subsumes cache coherence [10]. Note that cache coherence is only concerned about reads and writes to a single memory location, and that it requires that a write should *eventually* become visible to other processors.

So far we have discussed cache coherence and memory consistency in general. In the next sections several different memory consistency models will be discussed. Each memory consistency model poses different constraints on the order between memory accesses, and, i.e., differ in the amount of reordering that is allowed. A memory consistency model is relaxed or weaker compared to another model if it allows more reordering, and as a result it enables more pipelining of shared memory accesses. These models are briefly described in an attempt to visualize the spectrum of memory consistency models.

2.2.1 Sequential Consistency

Let's define *program order* as the ordering of memory accesses within a process. Lamport formalized an intuitive model, called Sequential Consistency (SC) [21], which is defined as in Definition 2.1.

Definition 2.1. *A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program (program order).*

Figure 2.2 shows the abstraction of memory provided to programmers when the system is sequentially consistent. Each processor executes all memory operations in program order and the memory is servicing one processor at the time. As a consequence, all processes observe writes in one single order (*write atomicity*). *Write atomicity* implies that all reads and writes that a processor issues after a write W do not become visible to other processor before they too have observed the write W. As a consequence, *write atomicity* guarantees that *all* processors observe writes from other processors in one single order. Memory operations appear *atomic* in this memory consistency model; and the operations appear *globally*, which means that *all* processes observe a memory access before a new access will be executed.

Definition 2.2. *A write is considered* completed *if the location of the write is updated in the shared memory. Consequently, subsequent reads will return the updated value. A read is considered* completed *if it returns the value from the location that is read.*

As with coherence, it is not important in what order memory operations actually have been issued or even complete. What matters for Sequential Consistency is that they appear to complete in a manner that satisfies the constraints just described. Please refer to Algorithm 1; this code executes correctly under sequential consistency, because the write to A becomes visible to $P2$ before the write to $flag$ is issued.

Definition 2.3. *A* read *by* Pi *is considered* performed with respect to Pj *when the issuing of a* write *to the same address by* Pj *cannot affect the value returned by the* read*.*

Definition 2.4. *A* write *by Pi is considered* performed with respect to Pj *when the issuing of a* read *to the same address by Pj returns the value defined by this* write *(or a subsequent* write *to the same location).*

Definition 2.5. *An* access *is* performed *when it is performed with respect to* all *processors.*

A set of sufficient conditions that guarantees Sequential Consistency is as follows. These constraints are adapted from [32, 33].

- Every process issues memory operations in program order

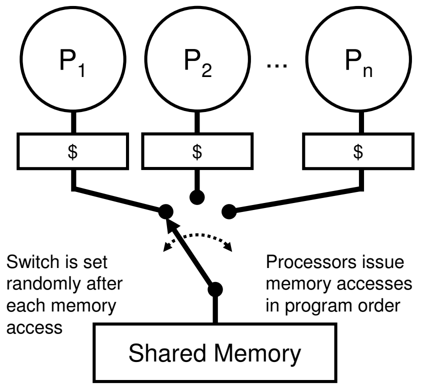

P1
P2
...
Pn
$
$
$
Switch is set randomly after each memory access
Processors issue memory accesses in program order
Shared Memory

Figure 2.2: Programmer's abstraction model for Sequential Consistency

- After a write is issued, the issuing process waits for the write to complete before issuing its next operation
- After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete before issuing its next operation. That is, if the write whose value is being returned has performed with respect to this processor (as it must have if its value is being returned), then the processor should wait until the write has performed with respect to all processors.

2.2.2 Processor Consistency

Restrictions on hardware to ensure Sequential Consistency may be too expensive, which makes it interesting to support a more relaxed model, which poses less restrictions. A more relaxed model, Processor Consistency (PC), is discussed in [15, 3, 17].

Definition 2.6. *A multiprocessor is said to be processor consistent if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program.*

Processor Consistency requires that writes issued by a processor are observed at another processor in the order they are issued. However, writes issued by different processors do not appear in the same order to all processors and therefore processor consistency does not satisfy write atomicity.

Processor Consistency is weaker than Sequential Consistency, thus it allows more reordering. However, Processor Consistency may not yield correct execution if the programmer assumes Sequential Consistency. The example in Algorithm 2 executes correctly under Sequential Consistency, but it can fail under Processor Consistency. $P2$ reads A, which is written by $P1$, and then writes B which in turn is read by $P3$. Without write atomicity there is no guarantee that the write to A by $P1$ becomes visible to $P3$ before the write of $P2$ to B becomes visible to $P3$. Some programs written with Sequential Consistency in mind may execute correctly under Processor Consistency, an example is Algorithm 1. Algorithm 1 executes correctly under Processor Consistency because writes from a single processor appear to be performed in order.

Algorithm 2 Importance of write atomicity for SC

```
Require: $A$ and $B$ are initially 0

P1                  P2                      P3
  $A \leftarrow 1$      while $A == 0$ do         while $B == 0$ do
                        skip                      skip
                      end while                 end while
                      $B \leftarrow 1$              print $A$
```

A set of sufficient conditions to guarantee Processor Consistency is as follows (slightly different from [15]).

- Before a read is allowed to perform with respect to any other processor, all previous reads must be performed
- Before a write is allowed to perform with respect to any other processor, all previous accesses (reads and writes) must be performed

Processor Consistency relaxes the program order of a write followed by a read. The model allows reads following a write to a different address on the same processor to be reordered. This is achieved in hardware by using write buffers and allowing reads to bypass the writes stored in the buffer. Processor Consistency allows a processor to read the value of its own write before the write completes in memory. If the read and write are to the same location then the read operation returns the value of the write from the write buffer. If the read is from a different memory location then it simply bypasses the write, which could still be stored in the write buffer.

The major advantage of Processor Consistency over Sequential Consistency is the ability to hide write latency by allowing reads to complete out of program order with respect to previous writes.

An interesting fact is memory ordering in Intel architectures throughout different generations of architectures. Intel Architecture Software Developer's Manual [2] discusses several memory-ordering models[1], which are related to different generations of Intel architectures. For example, the Intel386 processor enforces *program ordering*, where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances, i.e., enforcing *Sequential Consistency.*

To allow performance optimization of instruction execution (taken from [2]), the IA-32 architecture allows departures from Sequential Consistency to *processor ordering* in Pentium 4, Intel Xeon, and P6 family processors. These processor-ordering variations allow performance enhancing operations such as allowing reads to go ahead of buffered writes; apparently *processor ordering* in Intel architectures is similar to Processor Consistency. Intel states that the goal of any of these departures from Sequential Consistency to weaker memory consistency models is to increase instruction execution speeds.

Intel also briefly discusses memory consistency models that could be guaranteed in the future. Intel recommends that software written to run on Intel Core 2 Duo, Intel Atom, Intel

[1]Intel discusses memory-ordering models, which are in fact memory consistency models

Core Duo, Pentium 4, Intel Xeon, and P6 family processors assume Processor Consistency or a *weaker* memory consistency model. Intel also states that despite the fact that Pentium 4, Intel Xeon, and P6 family processors support Processor Consistency, Intel does not guarantee that future processors will support this model. To make software portable to future processors, it is recommended that operating systems provide, e.g., critical regions and APIs based on locking to synchronize access to shared memory in multiprocessor systems. Because of these statements, we think that supporting a weaker consistency model, like Release Consistency [15], is acceptable in a *general purpose* system.

2.2.3 Weak Consistency

A weaker consistency model can be derived by relating memory request ordering to synchronization points in the program. The Weak Consistency (WC), also known under the name Weak Ordering, model [12] is an example of such memory consistency model.

The benefit is that multiple read requests can also be issued, but not yet completed, at the same time. Reads can be bypassed by later writes in program order, and can themselves complete out or order, thus allowing us to hide read latency. The motivation behind weak consistency is that most parallel programs use synchronization operations to coordinate accesses to data when this is necessary. Between synchronization points, programs do not rely on the order of accesses being preserved [12].

In the Weak Consistency model there is distinction between *ordinary* shared accesses and *synchronization* accesses. The first type of shared accesses considers accesses to shared variables that do not impact the concurrent execution of the program. Usually these variables store the shared information between two or more processes. The second type of accesses are the accesses to variables that control the concurrent execution of the program. These synchronization variables are used to protect access to writable shared memory locations. It is the programmer's responsibility to ensure mutual exclusion for each access to a shared memory location.

Conditions to ensure Weak Consistency are as follows (slightly different from [12, 15]).

- before a *read* or *write* is allowed to perform with respect to any other processor, all previous *synchronization* accesses must be performed, and
- before a *synchronization* access is allowed to perform with respect to any other processor, all previous ordinary *read* and *write* accesses must be performed, and
- *synchronization* accesses are sequentially consistent with respect to one another

2.2.4 Release Consistency

Release Consistency [15] is an even more relaxed model than Weak Consistency. It extends Weak Consistency by distinguishing among types of synchronization accesses. Furthermore, different ordering requirements can apply to different types of shared memory accesses.

First, let's define *conflicting accesses* in Definition 2.7 and *competing accesses* in Definition 2.8.

Definition 2.7. *Two accesses are conflicting if they are to the same memory location, and at least one of the accesses is a write.*

Definition 2.8. *A conflicting access is competing if two conflicting accesses are not ordered, they may execute simultaneously and thus causing a race condition.*

All shared accesses that are not included in the set of *competing accesses* are called *non-competing accesses*. A conflicting access can be made *non-competing* by using synchronization. All accesses that are used to enforce an ordering among processes are called *synchronization accesses*. The categories of shared accesses and their relations are shown in Figure 2.3.

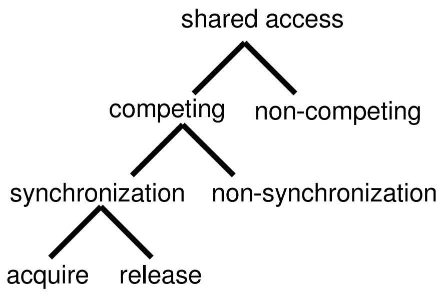


Figure 2.3: Categories of shared memory accesses in Release Consistency [15]

A synchronization access can be either an *acquire* or *release* operation. An acquire operation is performed to gain access to a set of shared locations, and is, e.g., a lock operation or a process spinning on a flag to be set. A release operation gives tasks waiting on an acquire operation the opportunity to perform the acquire operation. Related to the acquire operation, the release may be implemented as an unlock operation or a process setting a flag.

The purpose of a release is to inform other processes that accesses that appear before the release in program order have completed. On the other hand, the purpose of an acquire is to delay future access to shared data until no process is accessing the shared data, that is protected by this acquire.

A set of sufficient conditions for Release Consistency is as follows [40].

- before a non-competing access is allowed to perform with respect to another processor, all previous acquire accesses must be performed, and
- before a release access is allowed to perform with respect to another processor, all previous non-competing accesses must be performed, and
- competing accesses (e.g. acquire and release accesses) are processor consistent with respect to one another

2.2.5 Streaming Consistency

J.W. van den Brand et al. propose a relaxed memory consistency model, called Streaming Consistency [40]. This memory consistency model is targeted at the streaming domain and is weaker than Release Consistency.

In Streaming Consistency is interprocessor communication limited to sharing units of data through circular buffers (FIFO) that are located in shared memory. These buffers can have multiple producers and consumers, although, for predictability, the number of producers will be limited to one producer for each buffer. Figure 2.4 shows the abstraction model for streaming consistency.

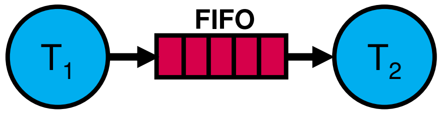


Figure 2.4: Streaming Consistency abstraction model

Different from Release Consistency are, in Streaming Consistency, synchronization operations explicitly related to circular buffers. Moreover, shared accesses to a certain buffer B are only allowed in a critical section that is entered by *acquire(B)* and exited by *release(B)*. This relation enables more reordering opportunities. A set of sufficient conditions for Streaming Consistency is as follows [40].

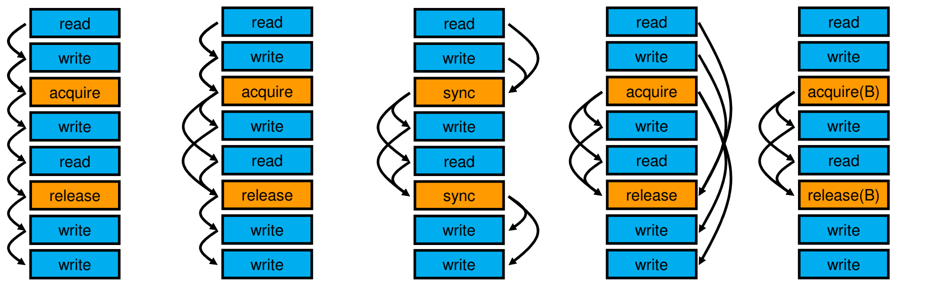
- before an access to a circular buffer b is allowed to be performed with respect to any other processor, the associated acquire access, *acquire(b)*, must be performed, and
- before a *release(b)* access is allowed to perform with respect to any other processor, the access to the circular buffer b to which the release is associated must be performed, and
- acquire and release accesses for a certain circular buffer are processor consistent with respect to one another, and
- circular buffers are only allowed to be accessed within critical sections

Streaming Consistency allows reordering of critical sections that are associated to different buffers, i.e., such critical sections are allowed to overtake each other. This is different in Release Consistency, where critical sections are allowed to overlap, as long as all accesses within the synchronization section are performed before the following release and after the preceding acquire, but never overtake each other.

Streaming Consistency enables more pipelining possibilities, but interprocessor communication is limited to sharing data through circular buffers. This makes it difficult to apply programs sharing a dynamic data structure, e.g., a tree. Fortunately, a program, written with Streaming Consistency in mind, runs correctly on a Release Consistent (or even stronger) MPSoC because a program written for a weaker model executes correctly on a system supporting a stronger model [40].

In order to exploit the additional pipelining possibilities, the programming model is adapted. A programmer has to explicitly program shared memory communication through circular buffers in critical sections. Streaming Consistency won't complicate programming for streaming applications because those applications already expose explicit communication.

However, this restricted programming model can cause problems if one wishes to execute third-party software. It may be that this software is not written with explicit interprocessor



(a) Sequential Consistency (b) Processor Consistency (c) Weak Consistency (d) Release Consistency (e) Streaming Consistency

Figure 2.5: Comparison of ordering constraints of accesses in a single processor in different consistency models

communication through FIFO buffers. As a consequence, this software can (i) either not be executed if shared data is stored in caches, or (ii) the program has to be rewritten such that all interprocessor communication is through FIFO buffers.

For general purpose applications, we expect it is better to support Release Consistency, as it still enables many performance gains over the stronger memory consistency models. In addition to this, it is possible to implement Pthreads on top of a Release Consistent MPSoC. A variant of FIFO communication, as proposed in Streaming Consistency, can be seen as an optimization for our software cache coherence protocol.

Release Consistency is attractive to support, because it allows us to provide an efficient software cache coherence protocol, while it does not significantly restrict the hardware. It is allowed to exploit write buffers to hide write latency, and consequently reads can overtake writes, thus not enforcing execution in program order. Write atomicity and global visibility of writes is not required, consequently, a network-on-chip can be used without taking specific precautions for (hardware) cache coherence.

2.2.6 Comparison of consistency models

Figure 2.5 shows the ordering constraints for the given memory consistency models as seen on one processor. It can be concluded that indeed ordering constraints are omitted if a weaker consistency model is compared with a stronger, or stricter model. Each ordering constraint that is removed can be seen as a potential performance gain.

Figure 2.6 shows relations between cache coherence protocols and memory consistency models. The cache coherence protocols, in general, will be discussed in Section 3. From Figure 2.6 it can be concluded that, to the best of our knowledge, most hardware protocols target at supporting memory consistency models closer to the strict memory consistency models, and software cache coherence protocols target at weaker memory consistency models. A cache coherence protocol together with the hardware always have to be related to a memory consistency model, otherwise a programmer is not able to reason about possible outcome of software.

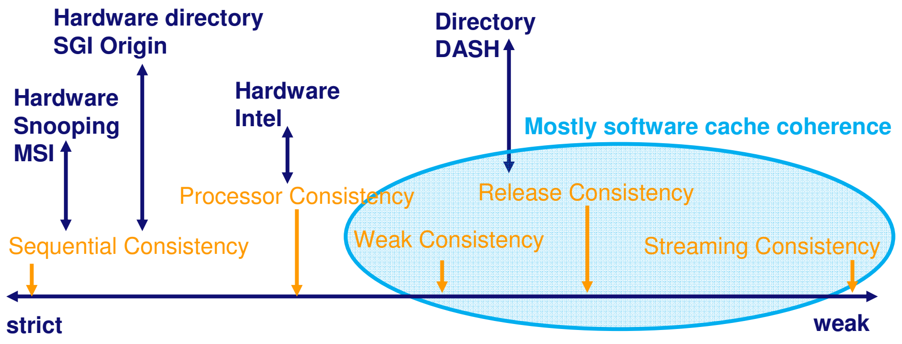


Figure 2.6: Memory consistency models and cache coherence protocols

2.3 Sequential Consistency with a NoC

Memory consistency models pose different requirements to the hardware. This section will discuss issues with Sequential Consistency, in combination with a NoC.

A sufficient condition for Sequential Consistency is *write atomicity*, and write atomicity is enforced by the third condition as given in Section 2.2.1. *Write atomicity* implies that reads and writes that a processor issues after a write W do not become visible to other processors before they too have observed the write W. Write atomicity can be an expensive requirement for the hardware, and we will show that it is undesirable in combination with a NoC. Figure 2.7 illustrates the order between reads and writes that is enforced by write atomicity. Each processor issues reads and writes, and a write operates as a *barrier*, i.e., enforcing a single order between all writes for all processors.

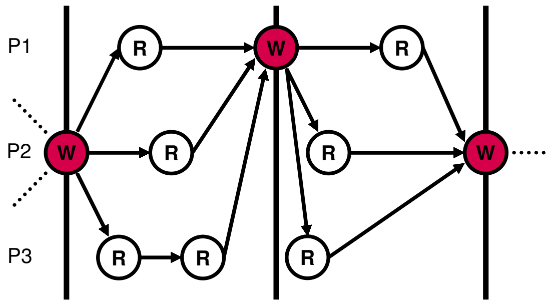


Figure 2.7: Write atomicity

Ensuring that writes appear in one single order to all processors is difficult and undesirable in a NoC. Example 2.2 which is related to Figure 2.8 illustrates this issue.

Example 2.2. *Three processors,* P1, P2, *and* P3 *are connected to shared memory through a NoC. Each processor observes a different latency to the memory,* L1, L2, *and* L3. *Assume* P1 *and* P3 *both issue a write to a location simultaneously. Write atomicity requires that all processors observe writes in one single order, how can we ensure that all processors observe all writes in one single order? Apparently a central directory, called* D *in Figure 2.8, is required, and this central directory is responsible for serializing all writes. However, this*

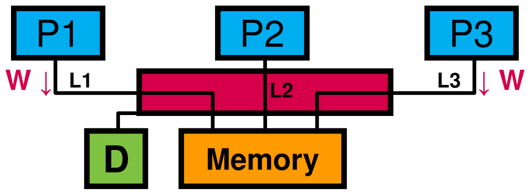


Figure 2.8: Write atomicity in a NoC

example shows exactly why a directory is undesirable in combination with a NoC. A NoC tries to handle as many memory accesses as possible in a given time span, by handling them in parallel. Unfortunately, a central directory will limit the parallelism that is achievable.

The SGI Origin System [10, 23] relies on a directory and cache controllers to serialize memory accesses. SGI Origin's cache coherence protocol ensures that all memory operations complete in program order and it guarantees write atomicity. Write atomicity is ensured by providing the appearance of atomicity by not allowing access to updated values until invalidation acknowledgements from all processors have been received. As a consequence, no processor can see the new value until it is visible to all processors. This is difficult to achieve efficiently, as a high number of acknowledgements are required, consequently generating a lot of traffic in the NoC.

Ensuring Sequential Consistency at the granularity of individual memory accesses is difficult. BulkSC [9] provides a Sequentially Consistent MPSoC based on a directory, an arbiter and it enforces Sequential Consistency on the granularity of groups of memory accesses. BulkSC groups sets of consecutive memory accesses in chunks, and these chunks appear to execute atomically and in isolation. If Sequential Consistency is enforced at chunk granularity, then it appears that Sequential Consistency is provided at the granularity of individual memory accesses [9].

Chunks have to be committed after execution, which means that the writes performed in a chunk become visible to all other processors. An arbiter is responsible for allowing chunks to commit. This arbiter checks, whether accesses from chunks do not overlap with reads or writes from other chunks executing simultaneously. For instance, a chunk reading or writing location X and another chunk writing location X are not allowed to execute concurrently. If a chunk *C1* commits, it sends a commit request to the arbiter. This arbiter checks whether chunk *C1* has read or modified locations that have been modified by other chunks; if there is no overlap of accesses to memory locations, then the chunk is allowed to commit and the modified locations are submitted to the directory in order to ensure cache coherence. If there is overlap the chunk is forced to restart execution, and because of the directory it fetches the most recent values.

The hardware required by BulkSC is expensive to implement. Furthermore, it is difficult to guarantee performance, because speculative execution of chunks makes it hard to derive performance bounds. Ensuring Sequential Consistency efficiently in a NoC is difficult, and maybe even unnecessary. Release Consistency is expected to be easier to implement efficiently in a NoC, and requiring explicit synchronization in software does not pose significant limitations.

3 Existing cache coherence protocols

This section discusses several cache coherence protocols and their classification. A brief explanation of these protocols can help understanding why the protocols are not suitable to be applied in a heterogeneous MPSoC with a NoC.

This section is organized as follows. Section 3.1 proposes a classification for *hardware cache coherence protocols* and *software cache coherence protocols*. Followed by Section 3.2 which discusses protocols that belong to the first class, hardware cache coherence protocols. Lastly, Section 3.3 discusses several protocols and approaches that belong to the class of software cache coherence protocols.

3.1 Classifying cache coherence protocols

In the introduction we have already mentioned *hardware* cache coherence protocols and *software* cache coherence protocols. However, we have not defined what exactly the difference is between a hardware and a software cache coherence protocol. This section attempts to provide a clear classification.

In [39] hardware protocols are loosely classified as: hardware approaches make the maintenance of coherence fully transparent to all levels of software. Whereas software protocols are loosely classified as: software approaches lift the transparency of the problem above the operating system or compiler, so hardware support is less complex, generally these approaches restrict the programming model more than hardware approaches.

The classification from [39] results in the issue that, for protocols such as DASH directory [24], it is not entirely clear whether it belongs to the class of hardware protocols, or to the class of software protocols. A directory-based cache coherence protocol is expected to belong to the class of hardware protocols because it relies on the directory in hardware, but, the DASH protocol relies on both the directory in hardware and explicit synchronization operations in software. As a consequence the DASH directory protocol is not completely transparent to the programmer, and thus, according to [39] DASH is not a hardware protocol.

We attempt to provide a clear classification and this classification is based on whether hardware or software *initiates* actions to ensure cache coherence. For example, *initiating* cache coherence actions by hardware corresponds to really *performing* these cache coherence operations in hardware. As a result, if a protocol, through software calls, requests hardware controllers to perform cache coherence operations, then the protocol fits in the class of hardware cache coherence protocols, because the cache coherence operations itself are handled by hardware controllers. It is important to note that the minimum hardware requirement for all software protocols, *cache maintenance operations* executed on the processor, in fact is part of the software. For example the cache maintenance operations, clean and invalidate, are part of the standard instruction set of ARM9 processors.

Cache coherence protocols can be roughly divided into two classes, hardware cache coherence protocols and software cache coherence protocols. Protocols are classified based on whether hardware or software initiates cache coherence actions. Some protocols, e.g., DASH directory, should belong to the class of hardware cache coherence protocols but also require specific features in the software. As a consequence there is a third class, which we call hybrid cache coherence protocols. All protocols that are mainly hardware protocols, but also require features similar to software protocols, and vice versa, belong to this class of hybrid cache coherence protocols.

The DASH directory protocol is a hybrid cache coherence protocol according to the new classification. The cache coherence operations are handled by the directory, which is implemented in hardware. All caches in the MPSoC require protocol support in the cache controller and these controllers and the directory also react to network transactions to maintain cache coherence, which is transparent to the software. However, because DASH supports Release Consistency it requires the software to contain explicit synchronization, because it only guarantees completion of memory accesses before the release of a critical section. Consequently it is a hardware cache coherence protocol, with some requirements to the software.

All cache coherence protocols that rely on software to initiate and handle cache coherence operations belong to the class of software cache coherence protocols. An example is our cache coherence protocol, which calls cache coherence operations on each synchronization, without requiring administration in either hardware or software. In addition, these cache coherence operations are local to a processor, whereas in a hardware protocol a processor can initiate cache coherence operations on other processors. Other examples are given in Section 3.3 which discusses for instance Conditional invalidation [37] and Shared regions [4], which both initiate cache coherence operations by software calls, after consulting a software-based administration.

3.2 Hardware cache coherence

This section discusses concepts from the class of hardware cache coherence protocols, namely snooping-based protocols and directory-based protocols. However, a specific implementation of a directory-based cache coherence protocol could belong to the class of hybrid cache coherence protocols, e.g., DASH directory.

3.2.1 Snooping protocols

A hardware solution for the cache coherence problem discussed in [10, 19] is a *snooping based cache coherence protocol*. This protocol targets, but is not limited to, multiprocessor systems as shown in Figure 2.1, in which the interconnect is a bus. Because of the nature of a bus, each processor can observe every bus transaction and all actions are serialized because only one processor can have access to the bus at a point in time. When a processor issues a request to its cache, the cache controller examines the contents of the cache and takes an appropriate action, which may include generating bus transactions, e.g., requesting data from the shared memory. Coherence is maintained by having all processors "snoop" the bus and monitor all transactions, and taking appropriate actions based on relevant bus transactions. Figure 3.1 [10, 16] illustrates this situation.

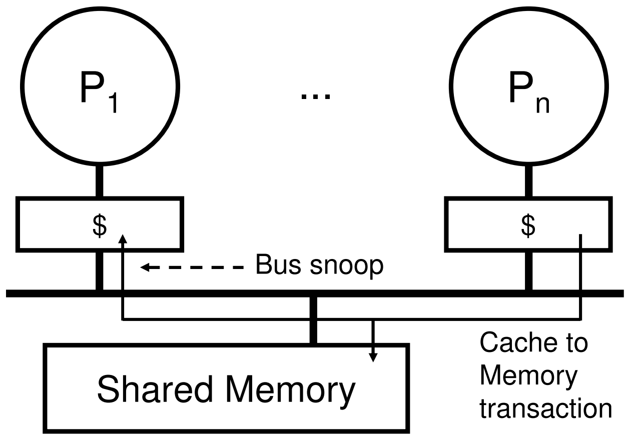


Figure 3.1: A bus-snooping cache coherent multiprocessor

The key properties of a bus that supports cache coherence are the following. Firstly, all transactions that appear on the bus must be visible to all cache controllers. Secondly, the transactions are visible to all controllers in the same order.

The simplest illustration of the snooping protocol, given in [10] and shown in Figure 3.1, is a system that has single-level write-through caches. Every write operation causes a write transaction to appear on the bus, which causes *write propagation*. As a result of these transactions every processor observes every write. If a processor's cache, say of processor *P1*, observes a write to a memory location X which is stored in its cache, it either invalidates its copy of X (in an *invalidation-based protocol*), or the copy of X is updated in the cache (in an *update-based protocol*). In either case, on the next read of X by processor *P1* the most recent value will be returned, either because of a *read miss* or because of the updated value in its cache. With write-through caches the shared memory will always have the most recent data, so the cache does not need to take cache coherence actions on a read.

The main problem with this simple write-through system is that every write goes to memory, potentially requiring a high memory bandwidth, which is why most modern processors use write-back caches. All writes that are unnecessarily put on the bus consume precious bandwidth, which leads to poor scalability. This led to the design of several snooping based protocols that targeted at MPSoCs with write-back caches. Examples are, invalidation-based protocols, e.g., MSI and MESI, and update-based protocols, e.g., Dragon. These protocols will not be discussed here as a snooping-based approach is not suitable to be applied in an MPSoC with a NoC; explanation of these protocols can be found in [10].

We are aiming at a scalable software cache coherence protocol for heterogeneous MPSoCs with a NoC. This makes it undesirable to use a snooping based protocol, as a snooping based protocol requires *write propagation* and *write serialization.* Although it is possible to provide global visibility of writes by, e.g., broadcasting each write, it is very costly and undesirable. *Write serialization* is only concerning writes to a single location. *Write atomicity* enforces one single order of writes to any location, but this is very hard to achieve in a NoC, which is already discussed in Section 2.3.

3.2.2 Directory protocols

This section discusses an example directory based cache coherence protocol [10, 19]. A brief overview of the operation of a directory based cache coherence protocol will be given by taking a very simple directory organization as an example, see Figure 3.2. All processor's caches are in write-back mode.

When a processor *P1* incurs a cache miss, a request for information is sent to the directory. For instance on a write miss, the directory identifies which processors have a copy of the written location and invalidate or update messages (depending on the policy) may be sent to these processors. Because these messages can be sent through disjoint paths in the network, and thus are potentially reordered, each receiver needs to acknowledge the messages, and the sender has to wait for all acknowledges to be performed, before performing the read or write. The basic operation of the simple directory based protocol will be illustrated in Example 3.1 and Example 3.2.

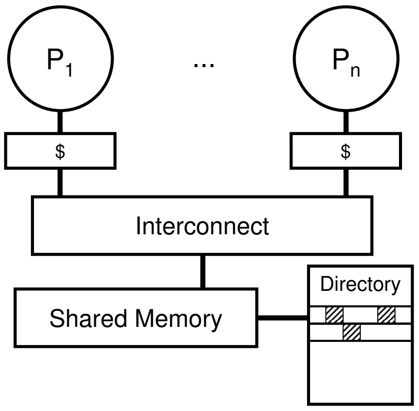
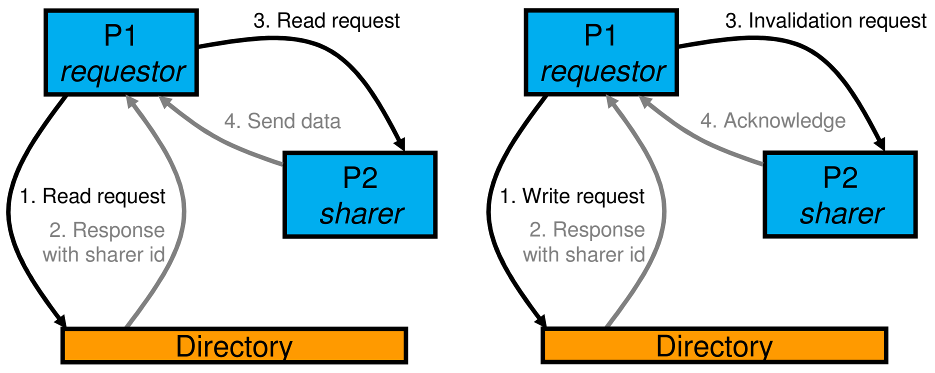


Figure 3.2: A directory-based cache coherent multiprocessor

Example 3.1. *See Figure 3.3(a). Let processor* P2 *hold a block* X*. Processor* P1 *tries to read* X*, but incurs a read miss. The proper action is that* P1 *sends a read request (1) to the directory. The directory responds (2) with the identities of all sharers of the block. A* sharer *is a processor that has a valid copy of the block in its cache. If there are no sharers, then the directory responds with the data that is requested. In this example only* P2 *is sharer of*



(a) Read miss to a block in modified state in a cache (b) Write miss to a block that is stored in another cache

Figure 3.3: Basic operation of a simple directory

X. *Processor* P1 *sends a read request (3) to sharer* P2*, and* P2 *sends the requested data in a response message (4).*

Example 3.2. *See Figure 3.3(b). Let processor* P2 *hold a block* X*, and processor* P1 *wants to modify* X*. If* P1 *concludes from the state of its cache that other processors have a copy of* X*, or* P1 *doesn't have a copy of* X*, a write miss will occur.* P1 *sends a write request to the directory, and the directory replies with the identities of all sharers. In this case only* P2 *has a copy of the block, and* P1 *sends an invalidation request to* P2*. Because* P1 *is not allowed to write before the invalidation has actually been performed, it has to wait for an acknowledgement of each sharer. The copy of* X *will be updated in* P1's *cache upon the write, and the directory sets the state of* P1 *to sharer of* X*.*

Although this example showed an extremely simple directory based protocol and many improvements have been made, we think this example illustrates the notion of directory based protocols. The idea stems from the introduction of directories in [8] and it has been experimented with in the DASH project [24]. According to the latter research project the directory can be quite costly, because it implements a logically centralized directory. The overhead in size could reach 20% of total memory according to [24]. Additionally the directory has to receive all requests initiated by read and write misses. Although the directory can be physically distributed, when the same memory location is being accessed extensively, the directory still forms a bottleneck, that could easily cause contention in the NoC. Additionally, as we need to consult the directory for reads and writes the memory access latency will be increased significantly.

3.3 Software cache coherence

The idea behind software cache coherence protocols is to invalidate, by software, a cache line when accessing shared data that is known to have changed by a different process. Several different approaches in the class of software cache coherence protocols have been developed. A survey of software solutions is given in [38], which proposes a classification of software cache coherence protocols.

The classification is made based on several criteria, of which we only discuss two criteria, the first is *dynamism*, which denotes whether decisions about cache coherence operations can be made at compile time (*statically*), or at run-time (*dynamically*). At compile time could either be, during design time, where the cache coherence operations are inserted by the programmer, or during the compilation process, where the cache coherence operations are inserted by the compiler. A second criterium is *selectivity*, which describes whether cache maintenance operations are done on the entire cache, or on regions of the cache (*selectively*).

In this short overview of related work in software cache coherence protocols we only consider software protocols that do not rely on a special compilation process that, e.g., determines shared accesses and inserts cache coherence operations. In our software cache coherence protocol decisions about cache maintenance operations are made statically; the programmer is responsible for inserting synchronization operations in the program, and cache coherence operations are performed on each synchronization operation.

To the best of our knowledge very few papers have discussed a software cache coherence protocol since the survey in [38], still most research focuses on a hardware approach. In this report two related projects will also be discussed, namely [29] which proposes a software oriented solution to avoid cache coherence issues and [40] which presents an efficient software cache coherence protocol for a novel memory consistency model (Streaming Consistency, see Section 2.2.5). The proposed software cache coherence protocol for Streaming Consistency is similar to one of the optimizations of our software cache coherence protocol, which will be discussed in Section 5.

3.3.1 Conditional invalidation

Tartalja et al. [37] propose a class of software protocols for cache coherence in multiprocessors with shared memory. Three cache coherence protocols are introduced and each protocol gradually performs better in terms of lowering the number of invalidations, by omitting invalidation in case the shared data hasn't been modified since the last access by a processor. We are only interested in their best protocol, which is *Version Verification.*

Shared data is stored in *segments*, i.e., a segment is the unit of data sharing, and access to a segment is protected by a critical region. Shared segments are *only* accessed in critical regions. More specifically, a shared segment is *explicitly linked* to a critical region. Decisions about invalidation of the shared segment are made upon the entry of a critical region.

The pseudocode in Algorithm 3 (taken from [37]) represents the idea of the Version Verification scheme. The shared memory stores the version number of each shared segment in the shared memory, on entry of a critical region this version number is fetched from the memory and compared with the version number local to the process (which denotes the version of the shared segment when it was accessed last). Example 3.3 and Figure 3.4 together show an example of accessing a shared segment S in this protocol.

Example 3.3. *This example is related to Figure 3.4. Assume an MPSoC with private write-back caches. A processor tries to access a shared segment* S *for both reading and writing. No process is currently accessing segment* S, *thus* Enter_Region *is allowed to perform. The last time when segment* S *was accessed by this processor its version was 3, but the current version in the memory is 5. As a result,* Enter_Region *decides to invalidate all lines belonging to shared segment* S. *Because this processor is requesting write access, the segment version will*

Algorithm 3 Entry and exit procedure of a critical region

```
Enter_region(processor, access_mode)

  Entering activities (e.g. synchronization);
  if (current version of the shared segment is different of the last version of the same segment used by the processor) then
    Invalidate cache lines belonging to the shared segment;
    if (access_mode = Read_Only) then
      Update private evidence of the shared segment version;
    end if
  end if
  if (access_mode = Read_Write) then
    Increment segment version;
    Update private evidence of the shared segment version;
  end if

Exit_region
  /* in case of a write-back cache */
  Copy cache lines belonging to the shared segment into the shared memory
```

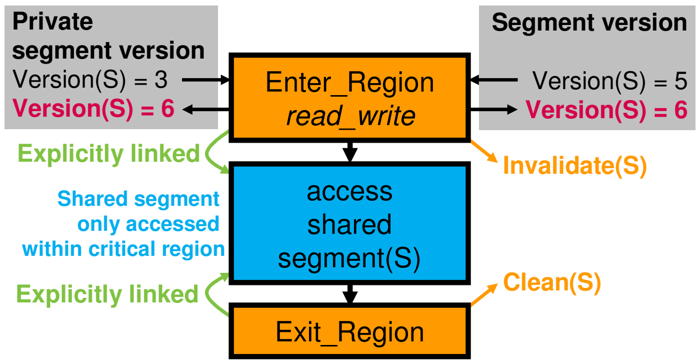


Figure 3.4: Accessing a shared segment under *Version Control*

also be incremented. After completing Enter_Region *the shared segment will be accessed, and following these accesses* Exit_Region *will be executed, i.e., cleaning the cache and copying back the written data to the shared memory.*

Major differences compared to our software cache coherence protocol are that we don't need to explicitly link specific shared accesses to a specific critical region, except for FIFO communication. Additionally, we don't need an administration about shared segments in the memory. More importantly, if an MPSoC relies on a specific programming model for cache coherence, then it is impossible to correctly execute software written without using this programming model.

3.3.2 Shared regions approach

A second software cache coherence protocol is called *Shared regions* [4] that is based on shared regions, which are loosely defined as a set of memory locations that are accessed together (e.g. within the same task) and in the same mode (reading or writing). Shared regions are assumed to be cache line aligned, since cache maintenance operations usually operate at cache line granularity.

In [4] two algorithms for cache coherence are presented. Only the second algorithm, *cache-validate*, will be discussed here as it tries to minimize the number of invalidations by exploiting administration in the shared memory.

For generality it is assumed that a write-back cache is used, but of course a write-through cache is also supported. Four synchronization primitives are presented:

- **ReadAccess** requests read-only access to a shared region
- **ReadDone** signals that the process has finished accessing (only reading) the shared region
- **WriteAccess** requests exclusive writing access to a shared region S (waits on all current ReadAccess to S to complete)
- **WriteDone** signals that the process has finished accessing (reading and writing) the shared region

In the cache-validate algorithm a primitive only performs invalidations if it is required (according to the administration). The administration is put in shared memory and is uncached. The administration contains the status of shared regions for every cache in the MPSoC. The status is either *HI*, which means the cache has an invalid (or no) copy of the shared region, or HV which means that this processor has a valid copy in the background memory. Invalidations are performed on entry of a critical region; shared segments are only accessed during critical regions, and the accessing mode (reading and/or writing) is known. Pseudocode of the algorithm is given in Algorithm 4.

The algorithm is based on the fact that shared regions are explicitly linked to critical regions. Upon entry of a critical region it is known whether the region will be read from or written to. The decision about invalidation is based on the administration in the memory, and the administration is only updated on entry of a critical region (processor only updates its own

Algorithm 4 ReadAccess and WriteAccess, WriteDone primitives in Cache-Validate. Region R is accessed by processor P

```
ReadAccess and WriteAccess

  if (status[P] == HI) then
    invalidate(R)
  end if
  status[P] = HV

WriteDone
  clean(R)
  for ((each cache ≠ P) and (status[cache] == HV)) do
    status[cache] = HI
  end for
```

administration), or on exit if the region has been modified (processor updates status of all caches; WriteAccess is mutually exclusive). For an example of the cache-validate algorithm see Example 3.4 which is related to Figure 3.5.

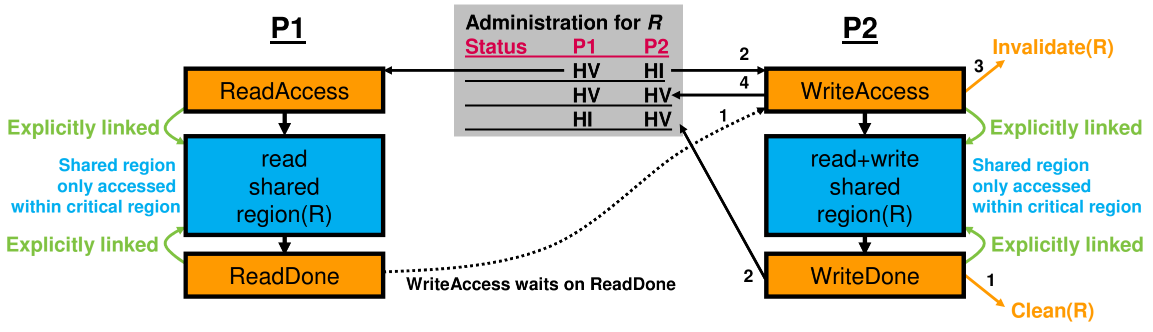


Figure 3.5: Example of *cache-validate* algorithm. Two processors access the same region R, $P1$ for reading, $P2$ for writing.

Example 3.4. *This example is related to Figure 3.5. Assume shared region* R *is put in write-back cacheable memory. Two processors access the same region* R*. First,* R *is accessed for reading only by* P1. P1 *checks the status of its cache in the administration, it reads* HV *which means that invalidation is not needed.* ReadAccess *returns and the critical region is entered. Then, while* P1 *is in its critical region, processor* P2 *tries to get* WriteAccess*, but has to wait on* P1 *to finish reading.* P1 *signals* P2 *that it has finished reading (1), and the administration does not have to be updated.*

P2 *checks the adminstration (2), and takes the appropriate action to invalidate region* R *(3), updates the administration as it has obtained a valid copy of* R *(4), and enters the critical region in which region* R *is modified. Then, in* WriteDone *region* R *is cleaned (1), and afterwards the administration for* all *caches is updated (set to* HI*, as the other caches contain an invalid copy of* R*) (2).*

Although this algorithm performs only necessary invalidations, it may only be useful if (more than) two threads are alternatively reading a shared region R in a critical region when it hasn't been modified in between two reads of one process. A major problem with this approach is that, as for conditional invalidation, this algorithm can not be applied to software written without this specific programming model. If there is no coupling between shared data being accessed and a critical region, and, more specifically, if the access mode is not explicit then the cache-validate algorithm can not be applied. Again, similar to conditional invalidation (see Section 3.3.1) our software cache coherence protocol does not use an administration in shared memory.

3.3.3 Software cache coherence in a NoC based MPSoC

A recent paper [29] discussed cache coherence and memory consistency issues in NoC based shared memory MPSoCs, which makes it relevant to our work. In their work they propose a software oriented solution, in which the cache coherence problem essentially disappears. Their approach is based on the fact that the system integrator knows what data in the software is being shared among processes and what data is private to a process. Based on this knowledge the data has to be separated into shared and private regions, for which shared regions are mapped to noncacheable regions of the memory, whereas the private regions can be mapped to cacheable regions. The idea behind leaving shared data uncached and local non-shared data cached has already been discussed in [44].

Their idea is to support Pthreads [1] and the following conditions have to be met:

- Usual automatic variables are allocated by the compiler on the thread stack, so they are strictly local
- Dynamically allocated local variables should be allocated using *local_malloc()*, which allocates memory in a private region
- Dynamically allocated shared variables should be allocated using *shared_malloc()*, which allocates memory in a shared region
- Thread global variables, known as the *thread specific data area* in the POSIX standard [1], must be allocated using *local_malloc()*.
- By definition, the C global and static variables are shared. This implies that the *.data* and *.bss* sections of the executable must be loaded in shared space.

We agree with the approach of introducing several mallocs and several heaps, one will be used for private data, the other heap will be used for shared data. The responsibility for the programmer or integrator to separate shared and private data is not expected to be a problem. However, leaving shared data uncached can lead to a severe performance penalty. We see separating shared and private data as an attractive optimization for our software cache coherence protocol.

4 Experimental hardware and software platform

This section describes the hardware platform on which we have implemented our software cache coherence protocol. Knowledge about the hardware and software platform is important, as this is a prerequisite to understand the implementations and its constraints.

This section is organized as follows. First we will briefly discuss the MPSoC, followed by a description of the ARM9 processors that are used. Then, the Celoxica RC340 with a Xilinx Virtex 4 FPGA will be described. Lastly, we will discuss the software platform, and we will focus on, the operating system, task daemon, load balancing, and the cache maintenance operations.

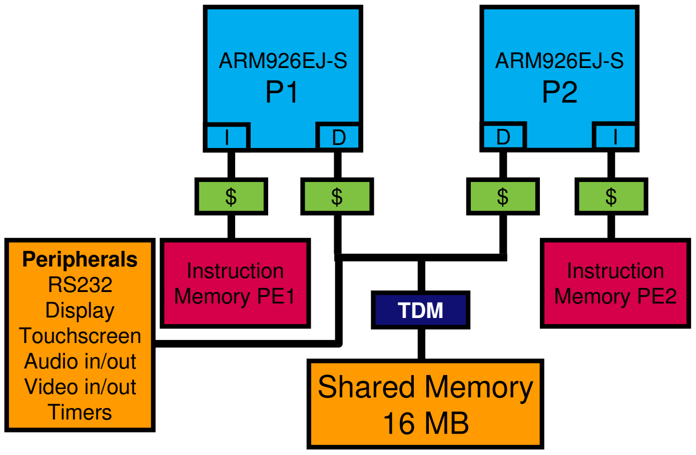


Figure 4.1: Multiprocessor system-on-chip architecture

4.1 Multiprocessor system on chip

Figure 4.1 shows an overview of the MPSoC architecture that is used throughout this project. This system consists of two ARM926EJ-S processors that are directly connected to the Time Division Multiplexing (TDM) arbiter on the shared memory. We have implemented variants of this MPSoC, e.g., in which the interconnect is a NoC instead of directly connected, or where an SDRAM is used as an additional shared memory, thus essentially constructing an MPSoC with a distributed memory. If any of the variants is used in an experiment it will be explicitly mentioned.

Our MPSoC has in total 16 MB of shared memory, which consists of two 8 MB SRAMs. Both processors are directly connected to the memory controller which gives each processor access to the memory according to a TDM arbitration scheme. Each processor is given an equal share of the total budget, in our case 50% of the replenishment interval is given to each processor. If a processor does not consume its slice it is wasted.

In the memory hierarchy there will be no reordering of memory accesses. All memory requests are put in a buffer, and the requests will be serviced in order. Each memory request, a read or a write, will be completed in memory before a new read or write is fetched from the buffer.

Memory accesses issued by a processor will not be reordered in the interconnect, not even in the Æthereal NoC. All accesses will be serviced in the order they have been issued by a processor. This is an important property of our MPSoC, as we can omit the acknowledgement of write completion.

The Æthereal NoC that is used in variants of the MPSoC is briefly discussed here. The MPSoC is shown in Figure 4.2, each processor is connected to the NoC and to a private 8 MB instruction memory. All peripherals, two 8MB SRAMs, and a 256MB SDRAM are connected to the NoC, and can be connected to each processor as soon as the network connections are set up. The connections in the network are currently set up during initialization by processor *PE1*, which essentially creates FIFO paths from, e.g., a processor to a memory. As a consequence all transactions from the processor to the memory will take the same physical path through the network, and no reordering will take place. Although the network can be reconfigured during execution, and transactions can be pipelined, this is not implemented and/or exploited in our current MPSoC.

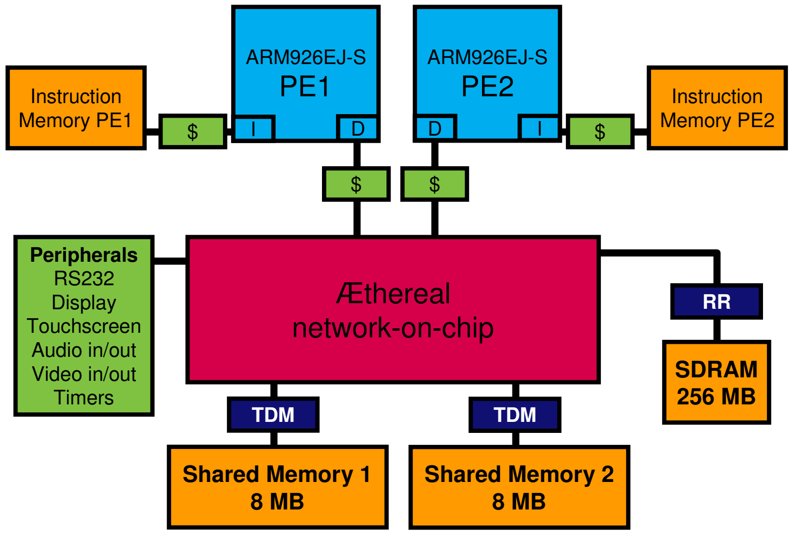


Figure 4.2: Multiprocessor system-on-chip architecture with Æthereal NoC

Processor mode	Description
User	Normal program execution mode
FIQ	Supports a high-speed data transfer or channel process *Used for task switching*
IRQ	Used for general-purpose interrupt handling
Supervisor	A protected mode for the operating system (entered upon *software interrupt*)
Abort	Implements virtual memory and/or memory protection
Undefined	Supports software emulation of hardware coprocessors
System	Runs privileged operating system tasks

Table 4.1: ARM processor modes

4.2 ARM9 processor

We have embedded two ARM926EJ-S processors in our MPSoC. The ARM9 processors have a ARMv5TEJ architecture, and have a Harvard architecture. ARM is a 32 bit RISC processor architecture, and ARM926EJ-S has a 5 stage pipeline, which consists of fetch, decode, execute, data, and write-back.

The standard instruction set can be extended by adding so-called coprocessors. For example the *system control coprocessor (CP15)* manages the memory management unit (MMU), caches, translation lookaside buffer (TLB), and the write buffer. The only defined system control coprocessor instructions are:

- MCR instructions to write an ARM register to a CP15 register.
- MRC instructions to read the value of a CP15 register into an ARM register.

In ARMv5 architecture these instructions can only be executed in *privileged* mode. The ARM processor can execute in 7 different modes, which are shown in Table 4.1. All modes other than user mode are known as privileged modes. They have full access to system resources and can change mode freely.

Our ARM926EJ-S implementation is equipped with on-chip L1 cache which is connected to external SRAM. The processor has a 16kB instruction cache and a 16kB data cache, which consists of four ways with each 128 cache lines of 32 bytes. The instruction cache is read-only, whereas the data cache is read/write with write-through or write-back policy. The policy can be set differently for regions of the addressing space; in our implementation we split the addressing space in sections of 1MB, for which we can set the policy to *noncacheable, write-through,* or *write-back*. Additionally ARM926EJ-S supports lock-down instructions to lock critical sections of code and data in cache.

Caches and write buffers are used to improve average system performance by hiding access latency to the background memory. The basic unit of storage in a cache is the *cache line* which contains 32 bytes of data. Cache operations operate on the granularity of cache lines. Cacheable regions, and whether these are write-through or write-back cacheable, are specified through settings in the MMU.

Let's define what happens when there is a *read miss* in a cacheable region. Typically ARM allocates a line in the cache and fetches the words to put in the cache line from the shared memory. The critical word is fetched first which enables to start computation with the critical word, while the rest of the cache line is still being transferred. In case of a *write miss*, in ARMv5, the data is written to the write buffer without allocating a cache line; in ARMv6 it is possible to instruct the processor to allocate a cache line upon a write miss.

When the processor reads data that is valid in the cache a *cache hit* occurs, and the data is fetched from the cache. In case of a *write hit*, the actions taken depend on the cache policy. If the cache is set as *write-through*; the data is written to the background memory (typically through a write buffer). If, on the other hand, the cache is set as a *write-back* cache; only the contents in the cache are modified and the line is marked as *dirty*. This means that it contains data that is more recent than the data in the background memory. When the cache line is replaced by a different memory address, or when a *clean* operation is called, the data is copied back to the background memory.

4.3 Celoxica RC340

The Celoxica RC340 [7] is used for prototyping and development of high-performance, high-throughput FPGA and soft-core microprocessor-based applications. The RC340 includes a 16M Gate Xilinx Virtex 4 FPGA, direct access to 32 MB of pipelined SRAM, a DIMM socket for DRAM and is supported with a wide range of video I/O and peripherals.

We have mainly used 8 user-programmable LEDs for debugging purposes. Through an API, provided by Celoxica, we were able to download the memory contents, which has been quite useful during debugging. During the project we have extended our hardware platform to use more functionalities provided by Celoxica, e.g., sending debug messages from the board to the computer through RS232.

4.4 Software platform

This section discusses the software platform. Each part of the software is able to access the entire addressing space, and there is no memory protection. The GCC compiler is used to compile C code for ARM processors, and in our setup we will build three binaries, the first is put into the instruction memory of *PE1*, the second is put into the instruction memory of *PE2*, and the third is put into the shared memory.

The software consists of roughly two phases, the *configuration*, and the *execution* phase. The configuration will set up the processor, e.g., initializing the TLB, MMU, caches, stack, and heap, before calling the *main* function that is implemented in C. The *main* function belongs to the execution phase, and the processor is initialized during configuration in order to execute software written in C. Additional initializations, e.g., initializing FIFO buffers, shared heap, and the operating system, will be called from the main function.

4.5 Operating system

The operating system, or kernel, has already been used in previous projects and is slightly adapted to make it applicable to our ARM9 MPSoC. Tasks, or threads, are created and activated in the kernel; at least one task needs to be started and executing forever before the kernel is started. The scheduler in the operating system is a TDM scheduler, which gives each task a slice of a specified length. During run-time it is allowed to dynamically create, start, and stop tasks, as long as always at least one task is executing. Originally the TDM scheduler gives a slice to a task, which is allowed to completely consume its slice, wasting the slice if nothing useful is to be done, e.g., spinning idly on a lock or flag. Therefore we have added a *yield* signal to the kernel, which is sent by the active task (which is the task currently consuming its slice) and instructs the scheduler to perform a task switch. The yield signal is added to the software at places where the task would spin idly otherwise, e.g., when blocking on a lock.

The kernel is responsible for servicing task switches. During a task switch the values in the processor's registers are put on the *active* task stack, and that task stack is written back to the memory. Then, following a round-robin fashion, the next task is chosen, whose task stack is loaded from the memory, and the registers are restored to the values that were put on its task stack and the newly scheduled task can continue execution. The task switch itself is triggered by a *fast interrupt request* (FIQ), which is triggered by an external counter.

4.5.1 Task daemon

We have mentioned that, whenever the kernel is running, there has to be at least one task that can be scheduled. Additionally we would like to have the property to arbitrarily start, stop, and replace tasks. In order to do this we have introduced a *task deamon*. Two daemons are executing in our MPSoC, one on each processor. Communication with a task daemon is through a FIFO buffer.

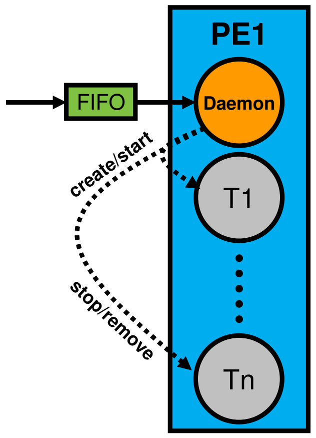


Figure 4.3: Task daemon

Figure 4.3 shows a processor, with $n + 1$ tasks. The first task is the task daemon, the other tasks are $T1$ through Tn. The daemon receives commands, e.g., create/start/stop/remove a task, through a FIFO buffer. The daemon is responsible for putting tasks into the gray circles, which represent task slots, in which a task can be executed.

A second processor also has an active daemon, which is able to manage tasks on its own processor. As our FIFO buffers only allow one writer, there has to be only one thread that can write into the FIFO buffer, this is for our applications not yet a problem. As for example in the SPLASH2 applications there is only one process that creates threads. In the future this should be changed, as there can be a higher need for load balancing; or when multiple applications will be executed simultaneously.

4.5.2 Load balancing

In future projects, where the focus is likely to be more on performance, the need for load balancing can arise. The current operating system, including the cache coherence operations, can be easily extended to support load balancing. First, it has to be possible to migrate tasks from one processor to another processor. In our software framework it is explicit where the stack of the active task, or another task, is put in the shared memory. The actions that have to be taken to migrate tasks is given in Example 4.1.

Example 4.1. *Suppose we want to migrate task $T1\_PE1$ from processor $PE1$ to processor $PE2$. $PE2$ is already executing two tasks, and therefore $T1\_PE1$ will be migrated to $T3\_PE2$ on $PE2$. The actions needed for migration are as follows. First, stop execution of $T1\_PE1$ and clean the range belonging to $T1\_PE1$'s stack (assumed that stack is in write-back region). Then, task daemon of $PE1$ will signal task daemon of $PE2$ that it wishes to migrate $T1\_PE1$. Task daemon of $PE2$ will create a new* dummy *task in task slot $T3$, and the data from $T1\_PE1$ will be copied to that slot, i.e., overwriting dummy data. Then $T3\_PE2$ can be started and the task has successfully been migrated.*

The most challenging part of load balancing appears to be how to decide which processor has the lowest load, and whether it is beneficial to migrate a task. A simple feasible solution may be using a single work pool that contains tasks to be executed, and have a processor fetching a task from the pool as soon as a slot becomes available. But with a work pool there is not yet a need for migration.

4.6 Cache maintenance operations

The cache maintenance operations, *clean* and *invalidate*, are implemented through system control coprocessor register instructions. These instructions can only be executed in a privileged mode, therefore we will execute these instructions during a software interrupt routine.

We use the following instructions for cache maintenance operations, taken from the ARM Technical Reference Manual [25]. $< Rd >$ contains the data which is required by the instruction, e.g., modified virtual address (MVA), or set/way.

```
MCR p15,0,<Rd>,c7,c6,1      @ invalidate data cache line (MVA)
MCR p15,0,<Rd>,c7,c6,2      @ invalidate data cache line (set/way)
MCR p15,0,<Rd>,c7,c10,1     @ clean data cache line (MVA)
MCR p15,0,<Rd>,c7,c10,2     @ clean data cache line (set/way)
MCR p15,0,<Rd>,c7,c14,1     @ clean + invalidate line (MVA)
MCR p15,0,<Rd>,c7,c14,2     @ clean + invalidate line (set/way)
```

The ARM9 cache consists of in total 512 cache lines and these lines are distributed over 4 ways having 128 sets/lines each. A line in the cache can be identified using a combination of set and way, or by an MVA. This MVA corresponds to the contents of a cache line, and if an invalidate MVA operation is called the cache controller first checks whether any of the 4 lines (each way) that can store the line corresponding to this MVA holds a copy of the location that should be invalidated.

ARM also provides instructions to *test and clean/invalidate* the entire cache, and instructions to *clean/invalidate* the entire cache, but these instructions can not be used because of predictability and composability issues, which will be discussed in Section 6.

Additional system control coprocessor instructions that are used in our system are:

```
MCR p15,0,0,c7,c10,4        @ DSB: ensure completion of memory accesses
                            @ flush write buffer
MCR p15,0,0,c7,c10,5        @ DMB: maintain order between memory access
```

Algorithm 5 illustrates the loop which is used to clean and invalidate an entire way. Important to note is that this loop is interruptible and that memory barriers are only needed after the entire loop has finished. These memory barriers ensure that cache coherence operations have been performed on all cache lines, before the barriers complete, which allows the software interrupt routine to return.

Algorithm 5 Clean and invalidate entire way

```
Require: r2 = 128, r1 contains set/way, r0 = 0
  while r2 > 0 do
    MCR p15,0,r1,c7,c14,2 {clean + invalidate line (set/way)}
    ADD r1,r1,0x20 {increment set}
    SUB r2,r2,0x1
  end while
  MCR p15,0,r0,c7,c10,5 {DMB}
  MCR p15,0,r0,c7,c10,4 {DSB}
```

5 A software cache-coherent Release-Consistent MPSoC

In previous sections we have discussed cache coherence, memory consistency and several protocols and approaches to maintain cache coherence. This section proposes our tuneable software cache coherence protocol. Our software cache coherence protocol is designed to support Release Consistency, which is a weak memory consistency model that relies on explicit synchronization. This explicit synchronization enables the use of software instructions to ensure cache coherence, as it is not required to ensure cache coherence on the granularity of individual memory access, but on the granularity of synchronizations. Furthermore, it is possible to implement standard communication libraries, such as Pthreads, on a Release Consistent MPSoC.

First, we'll start by explaining the *basic software cache coherence protocol* which poses almost no restrictions to the software and hardware. This basic protocol will be gradually altered by exploiting more information from the programming model, and these changes will significantly increase the efficiency of the software cache coherence protocol. These variants will show in what way the protocol performance can be tuned, e.g., by restricting the range on which cache maintenance operations are performed, or by changing the programming model.

Although we have implemented the cache coherence protocol on ARM9 processors and some design decisions may be dependent on the ARM9 implementation, this protocol can be implemented on *any* processor, as long as the processor can control its cache through cache maintenance operations. Examples of processors that provide cache maintenance operations are ARM processors, TriMedia VLIW mediaprocessors, and Xilinx MicroBlaze soft core processors. Specific information about ARM9 processors and our MPSoC is given in Section 4.

The rest of this section is organized as follows. Section 5.1 discusses the basic software cache coherence protocol, followed by Section 5.2 which discusses the separation of shared and private data, and its importance. Section 5.3 discusses the actions that are taken, initiated by a synchronization operation. Section 5.4 discusses how the software cache coherence protocol can be embedded in several programming models. Section 5.5 discusses an optimization which lowers the impact of cache coherence operations on private data. Lastly, Section 5.6 provides a brief summary of the proposed software cache coherence protocol, and in addition to this it identifies tradeoffs in the software cache coherence

Primitive	**Description**
acquire	Used to enter critical section, in which (shared) data is exclusively accessed; entire cache will be *cleaned and invalidated*
release	If a write-back cache for shared data is used: cache will be cleaned; the lock for exclusive access will be released

Table 5.1: Primitives for Release Consistency and the basic software cache coherence protocol

protocol.

5.1 Basic software cache coherence protocol

It is important to note that key properties of our software cache coherence protocol are:

- Caches are only coherent on synchronization points
- Off-the-shelf processors and caches can be used, as long as the processor can control the caches through *cache maintenance operations, e.g., clean or invalidate*

The protocol requires explicit synchronization in the software, this should not be a problem, because most parallel programs use synchronization operations to coordinate accesses to data when this is necessary. Between synchronization points, programs do not rely on the order of accesses being preserved. This reasoning has been part of the motivation for Weak Consistency [12], and is adopted by other weak memory consistency models.

It is the programmer's responsibility to add synchronization operations, thereby ensuring that shared data can not be simultaneously accessed by multiple threads while at least one of the threads is modifying the data. These synchronization operations can be divided in *acquire* and *release* operations, which are similar to the terminology of Release Consistency.

An *acquire* is performed before shared (competing) data can be accessed, and a *release* is performed after the thread has completed accessing shared (competing) data. These calls can be used to make competing accesses non-competing (see [15] or Section 2.2.4). Therefore, the acquire operation has to be a blocking lock operation for obtaining mutual exclusive access, and the release has to be a non-blocking unlock operation.

After performing an acquire, it is expected to read the latest values, therefore the acquire consists of a lock, followed by a *clean and invalidate* operation. The latter operation ensures that, if regions of the memory are put in a write-back cache, all modified data is copied to the memory first, and then, all (shared) data is invalidated, ensuring that subsequent reads will fetch the latest value from shared memory.

The release is used to make writes visible to other processors, which is done by a clean operation, followed by a release of the lock, enabling processors waiting on an acquire to enter their critical section. Independent of a write-through or write-back cache being used, the shared memory is up-to-date when no thread is currently executing a critical region. Therefore it is sufficient to perform invalidations on the entry or a critical region. The synchronization operations have semantics as given in Table 5.1.

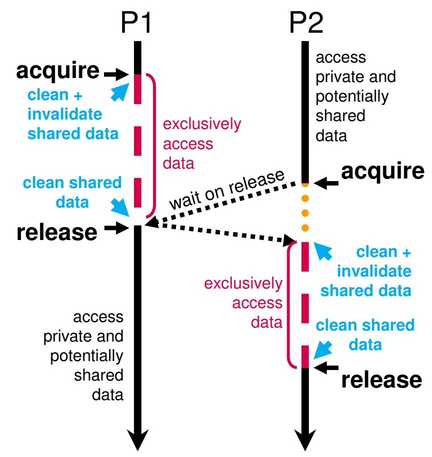

P1
P2
acquire
clean + invalidate shared data
exclusively access data
clean shared data
release
wait on release
access private and potentially shared data
access private and potentially shared data
acquire
clean + invalidate shared data
exclusively access data
clean shared data
release

Figure 5.1: An overview of the software cache coherence protocol

An overview of the protocol is shown in Figure 5.1. In this figure we mention cache maintenance operations on *shared data*, however, in the basic software cache coherence protocol shared data is not explicit. Shared data may be scattered throughout the entire addressing space, and as a consequence, it may be mixed with private data. It is not known which memory locations will be accessed during the critical section, and consequently, on an *acquire* the entire cache needs to be *cleaned and invalidated*, if regions of the memory are put in a write-back cache.

The granularity of cache maintenance operations is very important for software cache coherence protocols. *False sharing*, causing unnecessary evictions in protocols like MSI and MESI is non-existent in software cache coherence protocols. However, software cache coherence protocols can have a *sharing problem*, which can lead to incorrect results. This problem is illustrated in Example 5.1.

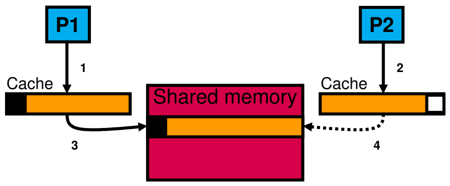


Figure 5.2: Sharing problem in software cache coherence protocols

Example 5.1. *See Figure 5.2. Imagine two processors* P1 *and* P2*, both having a write-back cache.* P1 *reads and modifies location* X1*, and* P2 *reads and modifies* X2*.* X1 *and* X2 *are mapped on the same cache line, which is in this example the granularity of cache maintenance operations. Assume both* P1 *and* P2 *hold a valid copy of the cache line in their cache. If first* P1 *modifies its copy in the cache (1), and then* P2 *modifies its copy in the cache (2), both cache lines are partially up-to-date. Let's assume an eviction (or clean operation) is performed on* P1*, then the cache line will be copied back to shared memory (3). However, an eviction or clean operation on* P2 *(4) will overwrite the values written by* P1*, which leads to incorrect behavior.*

The sharing problem illustrated in Example 5.1 can be avoided by changing the granularity of cache maintenance operations, or ensuring that no two variables share a cache line. The first would be possible if a cache maintenance operation would operate on, e.g., bytes instead of cache lines and only modified bytes would be replaced in the background memory. A second alternative would be forcing data structures to be cache line aligned. If data structures are cache line aligned the maximum number of data structures that can be mapped to a line is one, which can lead to inefficient memory usage.

Fortunately there is a third option to avoid the sharing problem. And this option is the most promising, because the sharing problem is only an issue if write-back caches are used. An easy way to avoid the sharing problem would be using write-through caches. Unfortunately, a write-through cache requires more memory bandwidth than a write-back cache because every write goes to shared memory. A major drawback of the basic cache coherence protocol is the lack of information about shared data, forcing us to put the entire memory in write-through caches to avoid the sharing problem, consequently requiring a lot of bandwidth.

5.2 Separation of shared and private data

The performance of the basic software cache coherence protocol can be increased significantly by separating shared and private data in the address range. In the basic protocol it was not possible to use write-back caches due to the sharing problem. It is preferable to keep the stack in a write-back cache, because a lot of writes will be issued to this memory region and these writes do never need to become visible to other processes as we assume the stack to be private. The first optimization is separating shared and private data, which enables us to put private data in a write-back cache (please note that there is no sharing problem for private data), and put shared data in a write-through cache (which avoids the sharing problem).

We assume that *global variables* are shared data and we propose that dynamic shared data is allocated on a *shared heap*. Therefore, we provide two heaps, a *private heap* and a *shared heap*. The first is mapped to a write-back cacheable memory region, and the latter is mapped to a write-through cacheable region. This separation will decrease the memory bandwidth requirements significantly compared to using a write-through cache only. Shared data should still be accessed in a critical region, and upon entry *only* shared data has to be cleaned and invalidated.

5.3 Entry and exit of a critical region in detail

Section 4 already described different instructions that can be used to clean and invalidate the entire cache or parts of the cache. These instructions are used to provide three options for cache maintenance operations on entry of a critical region. One metric to decide which option to use is the number of *false invalidations*. A *false invalidation* is defined as an invalidation of a cache line that may contain invalid data, but in fact it contains valid, potentially dirty, data. Another metric is the cost in terms of cycles, which is given in Figure 5.3 for three options, *entire cache*, *entire way*, and *line based on modified virtual address (MVA)*.

The first option is invalidating lines based on MVA. An advantage of this operation is that no false invalidations will occur, because the cache controller checks the contents of a cache line before invalidating that specific cache line. The drawback is that the software has to loop through all MVAs, which could consume a lot of cycles (see figure 5.3) if the range of memory addresses to be (cleaned and) invalidated is large. For instance, invalidating a shared heap of 3 MB requires invalidation of 98.304 MVAs.

The second option is invalidating the *entire* cache on entry of a critical region, which is acceptable if the memory range to be (cleaned and) invalidated is large, or if no information about the range to invalidate is known, e.g., in the basic software cache coherence protocol. A drawback of invalidating the entire cache is the eviction of private data, e.g., stack and private heap, which is likely to cause a large number of false invalidations.

A third option is invalidating a certain way entirely. From Figure 5.3 it can be concluded that it takes less cycles to invalidate one way than invalidating the entire cache. ARM9 cache has 4 ways, and our *invalidate entire cache operation* is implemented as invalidating 4 ways. The motivation for this implementation is *predictability* and this will be discussed in Section 6.

From a cycle count perspective it is preferable to invalidate only one way, and this also ensures that the other 3 ways, potentially caching private data, will not be invalidated. However, this cache maintenance operation can only be applied to guarantee cache coherence, when shared data is forced into a certain way; this will be discussed in Section 5.5. Although this ensures correct behavior, performance of some applications may degrade as this effectively decreases the cache size and associativity for shared data. This decrease in size and associativity results in a higher number of collisions and cache misses. Additionally, false invalidations can still occur, although, compared to invalidating the entire cache is the number of false invalidations limited to only one way, instead of the entire cache.

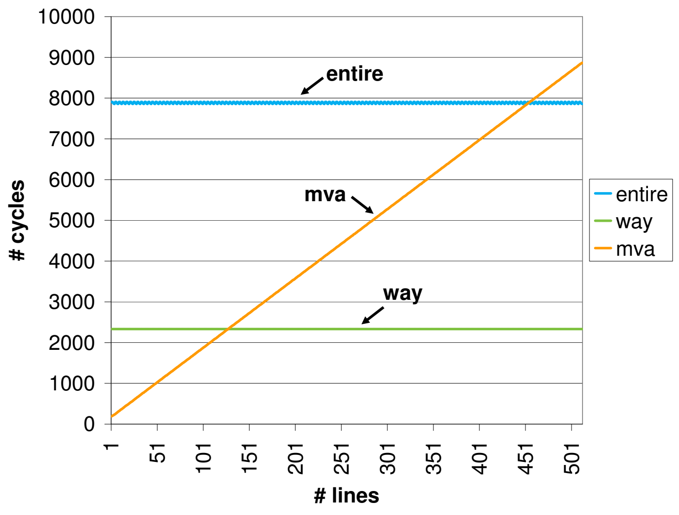
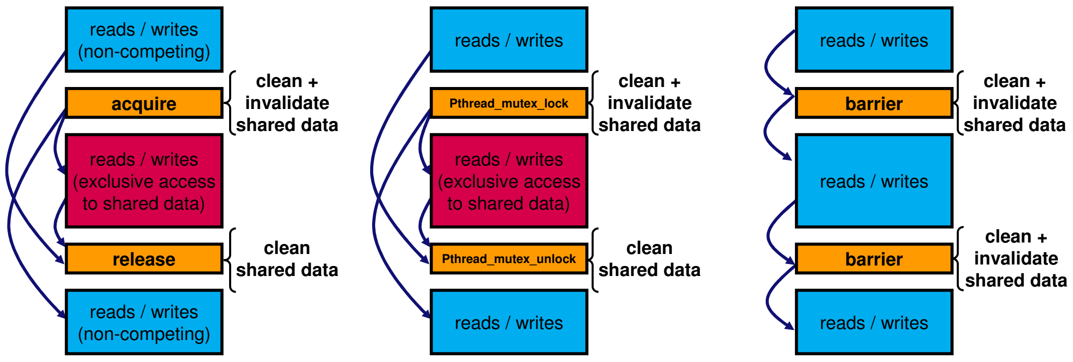


Figure 5.3: Cycles needed for invalidate operation


reads / writes (non-competing)
acquire
clean + invalidate shared data
reads / writes (exclusive access to shared data)
release
clean shared data
reads / writes (non-competing)
reads / writes
Pthread_mutex_lock
clean + invalidate shared data
reads / writes (exclusive access to shared data)
Pthread_mutex_unlock
clean shared data
reads / writes
reads / writes
barrier
clean + invalidate shared data
reads / writes
barrier
clean + invalidate shared data
reads / writes

(a) Release consistency (b) POSIX threads: lock/unlock (c) POSIX threads: barrier

Figure 5.4: Overview of ordering constraints of the software cache coherence protocol for different programming models

5.4 Programming models and the software cache coherence protocol

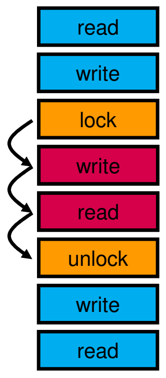
The programming model has an impact on the efficiency of the software cache coherence protocol. We state that exploiting information about the programming model and the memory map can increase the protocol's performance. This section discusses the general model in which our software protocol provides primitives for Release Consistency, POSIX threads, and FIFO communication.

5.4.1 Release Consistency

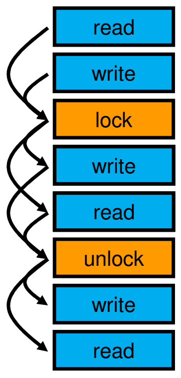
First, remember the primitives, acquire and release, that have been proposed in Section 5.1. These primitives can be used as an API for a programmer, who uses these to write a parallel program for a Release Consistent MPSoC.

Figure 5.4(a) shows the order between non-competing accesses, synchronization operations, e.g., acquire and release, and exclusive accesses to shared and/or private data. The lines represent the ordering constraints, which correspond to the ordering constraints defined by Release Consistency. There is no specific order required between accesses within each block, although of course the compiler should take care of data dependencies.

The order of memory accesses in our implementation is slightly stronger than requested by Release Consistency. This results from lacking expressiveness in, e.g., the C programming language. Our assumption is that *volatiles* will not be reordered with respect to each other, although in some compilers there may be issues with *volatiles* [13]. Locks are implemented by functions that read/modify volatile variables, and all accesses to shared data are programmed as accesses to volatile variables. This will ensure the order, shown in Figure 5.5(a), among a lock/unlock and a shared access, but it will unfortunately also enforce program order between shared accesses within a critical section, which is not required by Release Consistency. Additionally, it is not completely clear whether normal non-volatile accesses may be reordered with respect to volatile accesses. Let's assume this is allowed, then as a consequence there is no order ensured between normal accesses and lock/unlock operations.



(a) Volatile shared accesses



(b) Opaque function calls

Figure 5.5: Overview of compiler ordering constraints in implementations

A different option treats lock and unlock calls as *opaque function calls*, which is a function potentially reading/modifying any variable. The compiler can not reorder any reads/writes with respect to this function call, see Figure 5.5(b). This enables us to program shared accesses as normal accesses (omitting the *volatile* keyword). Consequently, in this implementation the compiler is allowed to reorder among shared accesses within a critical section, but it is not anymore possible to reorder ordinary accesses with respect to an acquire or release, which is allowed by Release Consistency.

Clearly, both implementations result in a stronger implementation than required by release consistency, thus potentially losing performance. In both implementations the processor is still able to reorder accesses, and *memory barriers* are used to enforce order between locks and accesses.

5.4.2 POSIX threads

POSIX threads, also known as Pthreads, [1] provides a standard for an API for creating, manipulating, and managing threads. Because of two reasons our MPSoC supports Pthreads, firstly, the Pthreads programming model is widely used and accepted for writing general purpose multi-threaded programs, and secondly, we use the SPLASH2 benchmark applications [41] for our experiments and these applications rely on Pthread calls.

Pthreads intentionally avoids stating a memory consistency model. Instead of a formal description the standard states [5, 1]:

> Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers. In addition, most of the formal work in the literature has concentrated on the memory as provided by the hardware as opposed to the application programmer through the compiler and runtime system. It was believed that a simple statement intuitive to most programmers would be most effective.

And in an attempt to give a clear and concise description [5, 1]:

> Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads. The following functions synchronize memory with respect to other threads: e.g., pthread_mutex_lock(), pthread_mutex_unlock(), ...

Clearly, following from these informal descriptions, the programmer is responsible for ensuring exclusive write access to a shared variable. Pthreads provides several functions that can be used to synchronize memory, e.g., pthread_mutex_lock() and pthread_mutex_unlock(). We have implemented these calls in our MPSoC, and ordering constraints between these calls and Pthread functions are given as lines in Figure 5.4(b). These ordering constraints resemble the constraints for Release Consistency (Figure 5.4(a), where *pthread_mutex_lock* is replaced by an *acquire*, and *pthread_mutex_unlock* is replaced by a *release*).

Parallel programs can be synchronized in different ways, previously we have shown that Pthreads provides *mutexes* that can be used for synchronization. Another primitive, that is used often in the SPLASH2 applications, is synchronization based on a *barrier*. A barrier has the semantics that each thread that is going to be synchronized stops execution when it reaches the barrier, and it waits until *all* threads have reached the same barrier, before proceeding execution. We have implemented the barrier in software, which will be discussed in Section 6.2.2. The ordering constraints are given as lines in Figure 5.4(c). Remember that the programmer is responsible for guaranteeing that no two threads can access the same memory location while at least one of the threads is modifying it. As a result it is not allowed to reorder accesses from any block with respect to a barrier. All reads/writes should be *safe* (locations are only read by threads, or exclusively written), and the barrier is used as synchronization.

The implementation of ordering constraints of barrier synchronization resembles Figure 5.4(c); the barrier call is *opaque*, which ensures that reordering with respect to the barrier is not allowed. All reads/writes can be normal accesses (not *volatile*) which would allow for reordering by the compiler. The insertion of a *memory barrier* before a barrier ensures that, in hardware, all reads/writes are transferred to the memory before the barrier is performed. Opaque functions and the memory barrier also ensure that the compiler does not optimize subsequent writes to the same memory location to writes to a register, as a consequence, these writes can become visible to other processors.

The primitives provided to the programmer have semantics as given in Table 5.2.

5.4.3 FIFO communication

In addition to Streaming Consistency [40], J.W. van den Brand et al. proposed an efficient software cache coherence protocol. We propose FIFO communication as an optimization of our software cache coherence protocol. This optimization restricts interprocessor communication to sharing units of data through FIFO buffers. Figure 5.6 shows a model of FIFO communication, in which a FIFO buffer is shared between two processes. One process *T1* writes data into the buffer, while the second process *T2* reads from the buffer.

Primitive	Description
pthread_mutex_lock	Used to enter critical section, in which shared data is exclusively accessed; entire cache will be *cleaned and invalidated*
pthread_mutex_unlock	If a write-back cache is used: cache will be cleaned; the lock for exclusive access will be released
barrier	Used to synchronize threads; entire cache will be *cleaned and invalidated*

Table 5.2: Primitives for Pthreads

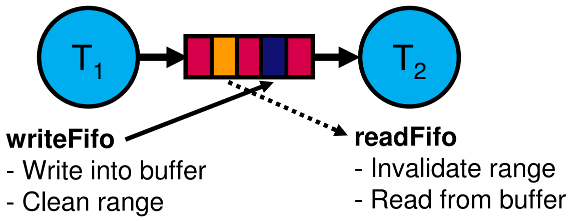


Figure 5.6: FIFO communication

A FIFO buffer is stored in a consecutive address range and administration for the buffer is stored in an uncacheable memory region. The producer consults the administration before it writes into the buffer (checks for space), and the consumer consults the administration before it reads from the buffer (checks for data).

The FIFO buffer consists of a number of buffer places (tokens) of a certain size (in bytes). The function *writeFifo* consists of obtaining exclusive access to a buffer place in the FIFO buffer, writing to the buffer place, cleaning only the memory locations that have been written to, and releasing exclusive access. The function *readFifo* consists of obtaining exclusive access to a buffer place of the circular buffer, invalidating only the memory locations belonging to the buffer place that will be read, reading the data from the shared memory, and releasing exclusive access.

This protocol is different from the previous proposed protocols, in the sense that it requires a specific programming model. Accesses to shared data is restricted to accessing data in FIFO buffers, and these accesses are only allowed during critical sections. In addition to this, a critical section is explicitly related to a FIFO buffer. Moreover, it is explicit whether during a critical section the thread accesses the FIFO buffer for writing, or reading. This enables a more efficient software cache coherence protocol, because a thread writing to the buffer only needs to clean the part of the buffer that it has accessed, analogously, a thread reading the buffer only needs to invalidate the part of the buffer that it will access.

In Release Consistency and Pthreads such relation between critical sections and shared data does not exists. Every thread is always allowed to access shared data, as long as no two threads can access the same memory location simultaneously, while at least one of the threads is modifying the location. As a consequence it is required to clean and/or invalidate the *entire* shared address range on a synchronization.

5.5 Optimization by forcing cache way

Each programming model in this section has its own primitives. In the first two, Release Consistency and Pthreads, we make no assumptions about specific shared data being accessed in specific critical regions. Consequently, the only option is to perform the cache maintenance operations on the entire shared address range, which could be done easily by operating on the entire cache. Unfortunately, this operation is likely to have a high number of false invalidations, which makes it an expensive operation. What we would like to have is an *invalidate range* operation that efficiently invalidates a range of memory locations, but unfortunately our ARM9 does not provide such an instruction.

In our MPSoC we can only choose between *invalidate entire cache*, *invalidate entire way*, and *invalidate line based on MVA*, as already discussed in Section 4. Figure 5.3 shows the number of cycles needed to perform the cache maintenance operations for a certain number of cache lines. Using these operations we attempt to implement an efficient software cache coherence protocol. The goal is to minimize the overhead, which would be minimizing the number of cycles needed to ensure cache coherence, and secondly, minimizing the number of false invalidations.

An idea is to use at most one of the four ways in the cache for caching shared data; then cache maintenance operations only have to be performed on this way, which is expected to lower the number of false invalidations. We have implemented this idea by changing the ARM9 cache controller; the cache controller is instructed to put cacheable memory locations above 7MB into way 1. Software is written such that private data is stored in the address range at locations below 7MB, and shared data is put in the range above 7MB. On entry and exit of a critical region the *way-based cache maintenance operations* are called.

The way-based cache maintenance operations are implemented in a software interrupt. No inputs are needed as we have to loop through the entire way. The ARM9 has a 4-way cache, and each way consists of 128 lines. To clean or invalidate the entire way it is sufficient to loop through all 128 lines for way 1. As a result, this clean/invalidate operation takes approximately as many cycles as cleaning or invalidating 128 lines using MVA.

The way-based approach successfully decreases the number of cycles involved in cache maintenance operations if the memory range to perform these operations on is more than 128 lines, compared to operating on the entire cache or looping through all MVAs. Additionally the number of false invalidations is expected to be lower than performing the cache maintenance operation on the entire cache.

Unfortunately the performance of the application with way-based cache maintenance operations is highly dependent on the application, as the size and associativity of the cache for shared data has decreased by 75%. If this is too small for the application, the application will suffer from a lot of cache misses, either caused by a small cache, or by a high number of collisions.

5.6 Summary

This section has discussed several variants of our software cache coherence protocol, and these protocols will be summarized here. The software cache coherence protocol relies on the fact that the programmer uses synchronization operations to ensure mutually exclusive access to shared variables. More specifically, the programmer is responsible for ensuring that no two threads can access a shared variable simultaneously while at least one of them is modifying that variable.

Caches are only coherent on synchronization points, and when no process is in a critical section the shared memory holds the latest values. Cache maintenance operations are embedded in synchronization operations; *acquire* has to perform a *clean and invalidate*, and *release* has to perform a *clean* operation.

The costs of cache maintenance, or cache coherence, operations are twofold. The first cost is the number of cycles that it takes to perform the cache maintenance operation. In a predictable ARM9 MPSoC we have three different options; clean and invalidate *entire cache*, *entire way*, and *line based on MVA*. A graph of the number of cycles required for invalidating a certain number of cache lines is given in Figure 5.3. The ratio between the total number of cycles needed for cache coherence operations and the total number of cycles needed for computation determines, to some extent, the impact of the cache coherence protocol on the performance.

The second cost is in the number of *false invalidations*, a false invalidation is the eviction of a cache line while the invalidation of that line is not required. This can happen when shared data is invalidated while it hasn't been changed, i.e., the cache already has the most recent value. Or, more likely, false invalidations take place because lines that store private data are evicted, e.g., invalidating the stack or private heap. Clearly the operation *invalidate entire cache* is likely to cause more false invalidations than *invalidate entire way*, and definitely more false invalidations than *invalidate line based on MVA*.

One of the most important optimization is the separation of shared and private data, which enables us to keep private data write-back cached, and shared data write-through cached. It also enables us to perform the cache maintenance operations on only the shared address range, either by looping through all MVAs in the shared range, or by using the way-based cache maintenance operations and forcing shared data in a certain way.

Another optimization is a specific programming model, which restricts interprocessor communication to sharing data through FIFO buffers. In this programming model all shared data accesses are explicitly linked to a critical section. It is also known whether the buffer will be accessed for writing, or for reading. Consequently, it is sufficient for a processor P to only clean the written locations if P is writing to the buffer. If a processor P is reading the buffer, then it is sufficient to only invalidate the locations that will be read in the critical section. These operations can be performed efficiently by cache maintenance operations using MVA, because only a small number of locations is expected to be accessed in a critical section.

6 Predictability and composability

MPSoCs grow in complexity with an increasing number of independent applications integrated on a single chip. This complexity makes it difficult to give guarantees about performance. Additionally if applications perform differently in isolation than when mapped on an MPSoC, it will be extremely difficult and time consuming to integrate multiple applications into one system.

A template for Composable and Predictable Multi-Processor System on Chips (CoMPSoC) is introduced in [18]. This template limits the interference between tasks executing on an MPSoC. As composability and predictability become more important we would like to design and implement our software cache coherence protocol such that these properties are not violated. We show that this is feasible with our software cache coherence protocol. It is expected to be difficult, maybe even infeasible, to construct a composable hardware cache coherence protocol, because many hardware protocols rely on the update or invalidation of the contents of other caches. Clearly, if threads on one processor cause invalidations on other processors the composability requirement can easily be violated.

Definition 6.1. *A job is a set of tasks that provides a certain functionality, e.g., DRM radio decoder job.*

Definition 6.2. *A system is predictable if bounds on the temporal behavior of one job can be derived at design time.*

Definition 6.3. *A system is composable if the temporal behavior of one job is not affected by other tasks.*

For predictability it is important that performance guarantees can be given at design time. This property poses several restrictions on the hardware as well as the software. Impact of software on predictability and composability, e.g., task switches, cache maintenance operations, and locks, will be discussed here.

6.1 Interruptible cache maintenance operations

For predictability we need to derive useful bounds on the temporal behavior of jobs. Therefore it is desirable that task switches are not postponed significantly. As a result we require the cache maintenance operations to be interruptible.

ARM9 [25] provides several instructions for cache maintenance operations. The cache can be controlled by system control coprocessor register 7. The reference manual states:

> The system control coprocessor register 7 provides operations for invalidating and/or cleaning the entire cache. If these operations are interrupted, then the R14 value that is captured on the interrupt is the address of the instruction that launched the cache operation + 4. This allows the standard return mechanism to restart the operation.

This makes these instructions not suitable for our predictable MPSoC, as it is not possible to derive useful bounds because this instruction can take a variable number of cycles to complete, depending on the schedule. Imagine a situation in which the task switch time is smaller than the time needed to complete the operation; this causes the operation to be restarted on each task switch.

An alternative cleaning (and cleaning with invalidation) scheme is optional in ARMv5. It provides an efficient way to clean, or clean and invalidate, a complete cache by executing an MRC instruction (taken from [25]). A global cache dirty status bit is used to loop through the cache.

To clean and invalidate an entire data cache, the following code loop can be used:

```
tci_loop    MRC p15, 0, r15, c7, c14, 3     ; test, clean and invalidate
            BNE tci_loop
```

This code loop is not suitable for our MPSoC, as it is not sure what will happen to this instruction if it is interrupted. If the task becomes active again after it has been interrupted in this loop it might restart, or continue with an arbitrary cache line. In either case it may result in incorrect, or unpredictable, behavior.

Additionally we need to be able to derive bounds on the time needed to actually perform the cache operations. Figure 5.3 shows the number cycles required for invalidating a certain number of cache lines, a similar graph can be given for performing the *clean* operation, which of course depends on the number of dirty lines.

Concluding, the only interruptible cache maintenance operation for an entire cache will be by looping through all cache lines; an example loop for operating on an entire way has been explained in Section 4. The loop is completely implemented in software by cleaning and invalidating all sets in all four ways. The loop is interruptible because it is entirely implemented in software. The loop can be successfully continued when the task is scheduled again, because the values in the registers and the stack are saved on each task switch.

6.2 Synchronization

6.2.1 Locks for mutual exclusion

Synchronization is important in our software cache coherence protocol. In our MPSoC threads will synchronize using locks (mutexes), barriers, and the C-HEAP protocol for

FIFO communication. The overhead of the implementation of the locks is not considered to be part of the software cache coherence protocol, as any implementation of locks, either software or hardware, can be used, as long as predictability requirements are met. As a result we can still state the software cache coherence protocol being scalable, even if the implementation of the locks does not scale well.

Our MPSoC is designed to be predictable, and this property has an impact on the implementation of the locks. Currently the hardware does not support hardware locking. For instance, test-and-set, compare-and-swap, and load-linked-store-conditional are not supported. Fortunately our MPSoC has the following important properties: (i) writes to a location are atomic (ii) reads and writes appear to have completed in the order they have been issued by the processor, technically speaking, both the interconnect and the memory controller do not reorder memory accesses.

Because of the lack of hardware locks we resorted to implementing locks in software. Several synchronization algorithms have been discussed in the literature, e.g., by Dijkstra [11], Peterson [28], and Lamport [22]. We have chosen to implement the bakery algorithm [22] that was proposed by Leslie Lamport. A proof of correctness is given in [22] and we will briefly discuss the algorithm here.

Informally the algorithm is analogous to a ticketing machine at the entrance of a bakery which gives each customer incrementally an unique number. Customers will be served in order, based on the unique given number. After a customer has been served, the customer who is next in line will be served. In software, these customers will be threads, and each thread will receive an unique number when it tries to obtain a lock. The pseudocode for the bakery algorithm is given in Algorithm 6 and Algorithm 7.

Definition 6.4. *$((number[j], j) < (number[i], i))$ means*
$((number[j] < number[i])$ or $((number[j] == number[i])$ and $(j < i)))$

The algorithm works as follows; a process i tries to enter its critical section, e.g., trying to obtain mutually exclusive access to shared data. First the process sets its private index in the *choosing* array to 1, to ensure that other processes, potentially spinning on the lock, will observe that i tries to retrieve a number, which could be the same or even a lower number. The next instruction computes the maximum ticket number in the queue, and this is incremented by one. Then the choosing array index is set to 0 and the while loop is entered. In this while loop process i will check for all other processes whether they are in a critical section, or trying to enter the critical section. Imagine a process j being in a critical section, then $number[j]$ will be unequal to zero, and $number[j]$ may be smaller than $number[i]$, in other words, process j acquired a lower ticket number, or if j and i have the same ticket number, then the process with the smallest id gets access to the critical section.

The shared variables for the locks have to be put in a noncacheable region of the shared memory to ensure that all writes will become visible to other processors. This has the disadvantage that processes blocking on a lock will spin on a variable in the background memory. However, the impact of spinning is limited in our MPSoC, because of a *yield* signal in the operating system and a TDM arbiter on the memory. Processes spinning will minimally impact the execution of other processes because of the TDM arbiter, which guarantees budget to each processor, and if a processor does not consume its slice then the memory controller is idle.

Algorithm 6 Bakery algorithm: process i tries to enter its critical section

```
Require: integer array choosing[1:N], number[1:N], N equals the number of processes
  $choosing[i] \leftarrow 1$
  $number[i] \leftarrow 1 + (\max i; 1 \leq i \leq N; number[i])$
  $choosing[i] \leftarrow 0$
  for $j = 1$ to $N$ do
    while $choosing[j] \neq 0$ do
      $yield();$
    end while
    while $(number[j] \neq 0)$ and $((number[j], j) < (number[i], i))$ do
      $yield();$
    end while
  end for
```

Algorithm 7 Bakery algorithm: process i exits its critical section

```
Require: integer array number[1:N], N equals the number of processes
  $number[i] \leftarrow 0$
```

6.2.2 Barrier synchronization

Many applications from the SPLASH2 benchmark set rely on *barrier* synchronization. A barrier has the semantics that each process that reaches the barrier stops execution until all processes that are synchronized on this barrier reach it. We have implemented the barrier entirely in software, based on locks for mutual exclusion. Pseudocode for the software barrier [10] is given in Algorithm 8, and this algorithm relies on processes spinning on a flag in the shared memory.

Algorithm 8 Barrier implementation in software

```
Require: local_sense[id] is private to thread id, myCount is a local variable
  $local\_sense[MyNum]++$
  $LOCK(entrylock)$
  $myCount = ++counter$
  $UNLOCK(entrylock)$
  if $myCount == P$ then
    $counter \leftarrow 0$
    $flag \leftarrow local\_sense[MyNum]$
  else
    while $flag \neq local\_sense[MyNum]$ do
      $yield();$
    end while
  end if
```

The barrier is based on an unique identifier, *MyNum*, of each process, a private variable *local_sense[MyNum]*, a global variable flag, a global variable counter, and a local variable *MyCount*. First, *local_sense[MyNum]* will be incremented, and then a lock for exclusive access is obtained. During exclusive access *myCount* is determined, and the global counter is incremented. Following, the lock is released and the private counter *myCount* is compared to the number of processors P. If *myCount* is smaller than P, then the process will start spinning on *flag*, which is continuously compared to *local_sense[MyNum]*. The variable flag is set by the last process to arrive (when *myCount* $==P$), following resetting the global counter to 0.

6.3 Composability

An additional preferable property is composability. Composability, as in Definition 6.3, is interesting because it enables the guarantee that a job's execution is not affected by *any* other job. Informally we could say, if a thread has functional and temporal properties, adding additional threads to the MPSoC does not change the guarantees for the first thread.

This property is difficult, maybe even impossible to achieve with a hardware cache coherence protocol. Contention on the directory causes threads to influence each others execution. A second, but more important, example is *false sharing*. False sharing does not exist in a software cache coherence protocol, but it can clearly violate composability requirements. False sharing causes evictions of cache lines, as the protocol forces processors to invalidate cache lines, if another processor has written to these cache lines, even while the data on the lines is not necessarily shared. If thread *T1* is the only thread in an MPSoC, then false sharing is nonexisting, but as soon as *threads* are added these threads could suddenly cause evictions, and thus influencing the temporal behavior of the thread *T1*.

Our software cache coherence protocol is suited to satisfy the requirements for composability. All decisions about cache coherence operations are local, and these operations do not impact the execution of other threads. Even when cache lines are shared, (false) sharing is avoided by putting the shared data in a write-through cacheable region. Because all invalidations are local, a thread on a processor *P1* will never influence the execution of a thread on a processor *P2*.

Threads on a single processor can influence each other, by flushing the cache, or fetching data in to the cache and consequently evicting another thread's data from the cache. This *inter-task* influence can easily be avoided by flushing the entire cache on each task switch [14]. Actually, because composability is defined at the level of jobs, caches only have to be flushed when switching between jobs. More complex ways to ensure composability include cache set-partitioning [27] and cache way-locking, but discussing these aspects is beyond the scope of this report. In addition to this, composability requirements are not taken into account in the experiments.

7 Experimental performance evaluation

This section discusses the experimental performance on our MPSoC that has been mapped on the Celoxica RC340 board. The experiments are executed on the *basic* MPSoC with two ARM9 processors, which are directly connected to the shared memory (see Figure 4.1). If a variant of this MPSoC is used, e.g., Figure 4.2, it is explicitly mentioned in the description of the experiment.

7.1 SPLASH2 benchmark applications

Several applications from the SPLASH2 benchmark suite [41] are used in the experimental performance evaluation. The SPLASH2 applications rely on a number of Pthreads calls; which we have ported to our MPSoC. In this section we will briefly discuss what applications have been executed on our MPSoC, and what the problem instances are. The applications, the standard problem sizes, the number of locks, and the uniprocessor execution time T_1 are given in Table 7.1. The uniprocessor execution time is the time that it takes to complete the application in two threads, on one processor without cache coherence operations. The descriptions about the programs are taken from [41].

Cholesky: it performs a blocked Cholesky Factorization on a sparse matrix; a sparse matrix is factored into the product of a lower triangular matrix and its transpose. It is similar in structure and partitioning to LU, but two major differences are; (i) it operates on sparse matrices, which have a larger communication to computation ratio for comparable problem sizes, and (ii) it is not globally synchronized between steps. See [31] for more information.

FFT: it performs a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize interprocessor communication. The data set consists of the n complex data points to be transformed, and another n complex data points referred to as the *roots of unity*. Every process is assigned a contiguous set of rows. Communication occurs in three matrix transpose steps, which require all-to-all communication. In our experiments we execute FFT and its inverse FFT, in order to have a longer execution time and additionally we can immediately check the outcome. See [43] for more information.

LU: it factors a dense matrix into the product of a lower triangular and upper triangular matrix. The dense $n{\times}n$ matrix A is divided into an $N{\times}N$ array of $B{\times}B$ blocks ($n = NB$)

Application	**Problem size**	**Locks**	T_1 **(M cycles)**
Cholesky	lshp	305	467.99
FFT	64K points	13	1242.32
LU contiguous	512×512 matrix 16×16 blocks	66	8850.97
LU non-contiguous	512×512 matrix 16×16 blocks	66	9355.27
Radix	256K integers radix 1024	12	1656.75
Raytrace	teapot 64x64	12237	1446.55

Table 7.1: SPLASH2 applications

to exploit temporal locality on submatrix elements. According to [41] the block size should be fairly small ($B = 8$ or $B = 16$). Two versions of LU factorization are given, *contiguous*, which is optimized for minimizing cache misses, and *non-contiguous*, which suffers from more cache misses. See [43] for more information.

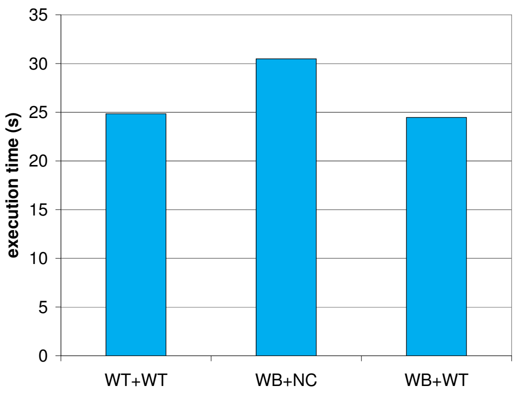
Radix: it performs a iterative integer radix sort. The algorithm performs one iteration for each radix r digit of the keys. In each iteration, a process passes over its assigned keys and generates a local histogram. The local histogram are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. See [20, 42] for more information.

Raytrace: it renders a three-dimensional scene using ray tracing. A ray is traced through each pixel in the image plane, and reflects in unpredictable ways off the object it strikes. Each contact generates multiple rays, and the recursion results in a ray tree per pixel. The image plane is partitioned in contiguous blocks of pixel groups. The data access patterns are highly unpredictable in this application. See [34] for more information.

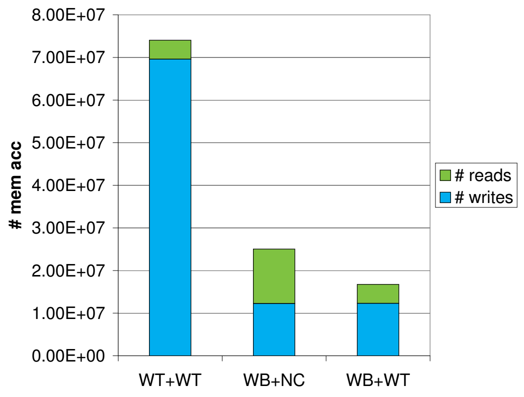
7.2 Separation of shared and private data

This experiment determines the impact on the number of memory accesses of separating shared and private data, and additionally putting shared and private data in different types of caches. The separation of shared and private data appears to be one of the most important optimizations, and it is likely to be easily achievable. We call *private* data all data that is *local* to a thread, this means that no other thread should require access to this data, neither for writing nor for reading. Examples of private data are; dynamic data allocated on the private heap, and the thread's stack. We call *shared* data all data that is *potentially* read or modified by other threads. Examples of shared data are; the data that is communicated between threads, dynamic data allocated on the shared heap, and global variables.

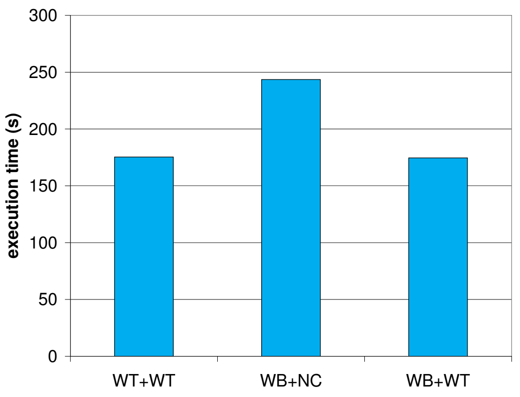
In this experiment we'll show, by using two applications, what the main advantage of separating shared and private data is. We execute two experiments, firstly we execute FFT, and secondly, LU contiguous on one processor, with both executing in two threads.



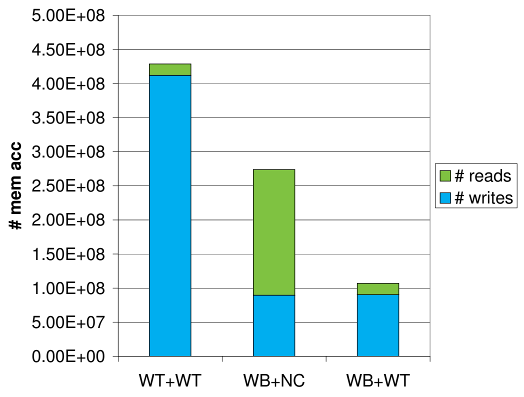
(a) Execution times FFT



(b) Number of memory accesses FFT



(c) Execution times LU contiguous



(d) Number of memory accesses LU contiguous

Figure 7.1: Impact of separation shared and private data

Because the applications are mapped on one processor the cache coherence operations can and will be disabled.

Both applications will be executed in three configurations, in the first we assume no information about the memory map *(WT+WT)*, and because of sharing we must keep the entire memory in write-through caches (and on a synchronization the entire cache should be cleaned and invalidated, but this effect will not be considered in this experiment). The second configuration shows the situation which was proposed in [29], where private data is (write-back) cached, but shared data is left uncached *(WB+NC)*. The third configuration shows our proposal, where we keep private data in write-back cache, and shared data in write-through cache *(WB+WT)*.

Figure 7.1(a) shows the execution times for FFT for the three configurations, and Figure 7.1(c) shows the three execution times for LU contiguous. We observe the highest execution times for the FFT and LU contiguous when these are executed in configuration *(WB+NC)*, in which the shared data is left uncached. Clearly, caching shared data can have a major impact on the performance, and this will even become more important if the ratio shared-to-private memory accesses becomes larger, or if the memory access latency is increased.

Surprisingly, the impact of caching private data in a write-through cache, or in a write-back cache does not have a significant impact on the execution time for these example applications. The execution times for *(WT+WT)* and *(WB+WT)* are almost identical. The main difference between these two configurations is that in *(WT+WT)* all writes to cacheable private locations are propagated to the memory through a write buffer. Apparently this write buffer succeeds well in its task to hide write latency.

Figure 7.1(b) shows the number of memory accesses for FFT, and Figure 7.1(d) shows the number of memory accesses for LU contiguous. A column in this figure represents the total number of memory accesses, the top part of a column represents the number of reads, the bottom part represents the number of writes. The configuration, *(WT+WT)* has the highest number of memory accesses, which are mainly writes, and we expect that these are mostly writes to the private stack. The number of reads in *(WT+WT)* is comparable to the number of reads in *(WB+WT)*, and is significantly less than in *(WB+NC)*. From this we may conclude that, as expected, caching shared data lowers the number of memory accesses. Apparently shared data is mainly read, as the number of reads shows the largest difference between *(WB+NC)* and *(WB+WT)*.

The number of writes is comparable in *(WB+WT)* and *(WB+NC)*, and it is significantly less than in *(WT+WT)*. Clearly, the high number of writes in *(WT+WT)* is caused by writes to private data, and because of memory bandwidth requirements it is preferable to keep the private data in a write-back region.

7.3 Speedup SPLASH2 applications

In this experiment we will discuss the speedup of the SPLASH2 applications on our MPSoC. The speedup is relative, as it is calculated using Equation 7.1, in which S is the speedup, T_1 is the execution time of the parallel algorithm in 2 threads on one processor without cache coherence operations, and T_p is the execution time on one processor if the algorithm is executed in parallel on p processors, each executing one thread.

$$S = \frac{T_1}{T_p} \tag{7.1}$$

The applications are partitioned and executed in two threads. In the single processor instance both threads are mapped on one single processor, and cache coherence operations are disabled. Private data is put in a write-back cache, and shared data is put in a write-through cache.

In the parallel instance, the two threads are mapped on two different processors, private data is put in a write-back cache, and shared data is put in a write-through cache. In addition, the cache coherence operations are enabled, and the *selectivity* of the operations will be varied in three instances. In *entire*, the entire cache will be cleaned and invalidated on each synchronization; in *way* only way 1 will be cleaned and invalidated; and in *MVA* the cache coherence operation will loop through the entire shared region. The size of the shared region is 4 MB for all applications.

Figure 7.2 shows the relative speedup for the SPLASH2 applications with standard problem sizes. For each application three columns are shown; these are, from left to right, the speedup for cache maintenance operations *entire*, *way*, and *MVA*. It is important to note that *only* for the instance where way 1 will be cleaned and invalidated (way), a certain way for shared data is forced. In the other instances shared data can be put in any of the four cache ways.

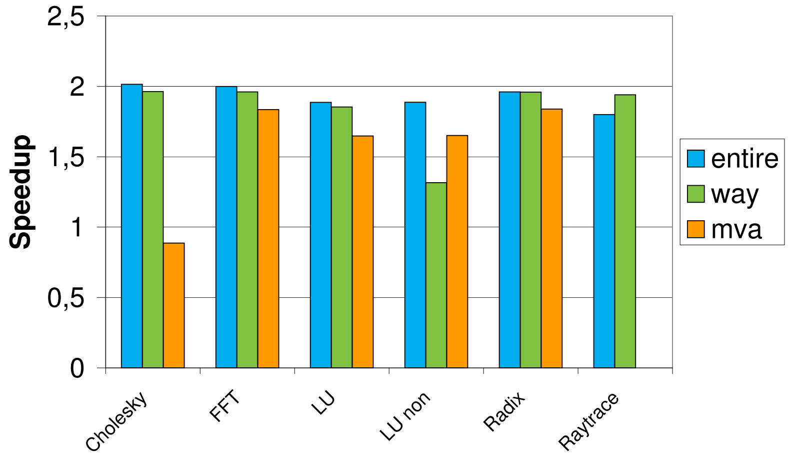


Figure 7.2: Relative speedup of SPLASH2 applications

For most applications the speedup is close to 2. For Cholesky we see a super linear speedup, this is most likely because of the increase in total cache size. Surprisingly is the speedup for the clean and invalidation of the entire cache on each synchronization in most applications better than for the other instances. We expect that this is mainly a result from the low number of synchronizations in one execution, technically speaking, the application has a low synchronization-to-computation ratio.

For most applications the second to best instance is by forcing shared data into way 1 and performing cache coherence operations only on this way. Performance drops if shared data is forced into a way, because the cache size for shared data is effectively smaller and the associativity is 1, which is likely to cause a high number of collisions. However, the time spent in cache maintenance operations is smaller for way-based cache maintenance compared to entire cache. Additionally less false invalidations occur, in comparison to entire cache.

The lowest speedup is observed when the cache is cleaned and invalidated by looping through all MVAs in the shared region of the memory. This is the direct result from the large number of cycles needed to actually perform the looping. If an efficient *clean and invalidate range* instruction would be implemented then the performance of this instance is expected to increase significantly.

The speedup of Cholesky is good, except for *MVA*. The execution time of the parallel implementation with cache coherence operations based on *MVA* is extremely large. We assumed a shared heap of 4 MB large for all applications, using MVA to loop through 4 MB takes a high number of cycles, this overhead is too much in case of Cholesky.

The difference between LU contiguous and LU non-contiguous is interesting. The performance of entire cache and MVA in LU contiguous and LU non-contiguous are comparable, but there is a huge difference when way-based cache-maintenance operations are used. As previously mentioned, the main difference between LU contiguous and LU non-contiguous is that the first is optimized for caches, and the second is not, the access pattern for LU

Application	**Accesses/cycle**
Cholesky	0.0141
FFT	0.0137
LU contiguous	0.0122
LU non-contiguous	0.0249
Radix	0.0023
Raytrace	0.0072

Table 7.2: Memory bandwidth requirements for SPLASH2 applications on a uniprocessor

contiguous is such that less cache misses occur than for LU non-contiguous. The increase in cache misses in the latter application has a large impact on the performance, because the smaller cache size and associativity results in a significant increase in the number of cache misses, mainly because of collisions.

Raytrace is interesting because in one execution, compared to the other applications, relatively a high number of locks are passed, which results in a high number of cache coherence operations. The execution of Raytrace with cache coherence operations using MVA takes too much time to yield any useful results. The instance that cleans and invalidates entire cache suffers from many *false invalidations* and additionally the time spent in cache coherence operations is larger than for the instance that uses way-based clean and invalidate. This results in the interesting outcome that, for Raytrace, forcing a way is better compared to clean and invalidate entire cache, for Raytrace, in terms of execution time.

The overhead of the software cache coherence protocol appears to be low for these applications. This can have several reasons, which are important to draw conclusions for other applications from these results. First of all, the SPLASH2 benchmark set is optimized to minimize communication (and thus synchronization), and we only perform cache coherence operations on synchronization points. Secondly, we use software floating point computations, which takes about a factor of four longer to complete, and this results in an even lower synchronization-to-computation ratio.

7.4 SPLASH2 memory bandwidth requirements

Speedup is not the only important metric, this section discusses the memory bandwidth requirements for the applications. This metric is important for scalability, as it gives information about the service that the application requests from the interconnect and the memory.

The standard applications are mapped to two processors, and the number of memory accesses observed on the interconnect is counted. Accesses to shared memory by a processor that is spinning on a lock are excluded. Each application is executed with three different cache maintenance operations; cleaning and invalidating *entire cache*, *way*, and using *MVA*. We have also determined the memory bandwidth requirements for the parallel applications mapped on one processor, without cache maintenance operations. The memory bandwidth requirement is calculated by dividing the *total number of memory accesses* by *the total number of cycles*. These memory bandwidth requirements are given in Table 7.2.

Figure 7.3 shows the memory bandwidth requirements for the three configurations of each

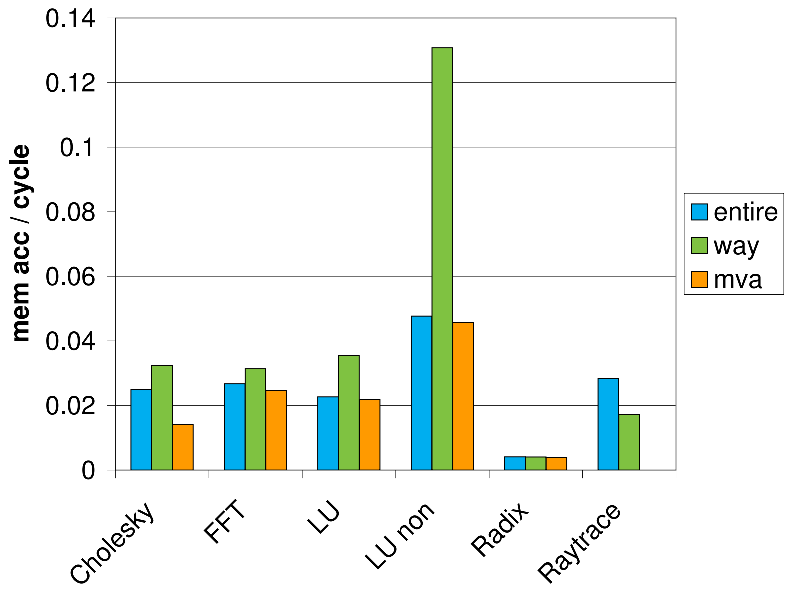


Figure 7.3: Memory bandwidth requirement for SPLASH2 applications

application. The columns represent, from left to right, the memory bandwidth requirements of the applications with cache maintenance operations *entire*, *way*, and *MVA*. The total number of memory accesses is determined for each application, and these applications are mapped on two processors, therefore we have determined the number of memory accesses on both processors.

Most applications pose only a small bandwidth requirement, which varies around 2%. It must be noted that increasing the execution speed, by for instance hardware floating point operations instead of software floating point operations, is expected to increase the bandwidth requirements.

The memory bandwidth requirements for SPLASH2 applications is approximately doubled when the applications is mapped to two processors, compared to the uniprocessor. This results from a speedup of almost factor 2, while the total number of memory accesses stays approximately the same as compared to the uniprocessor.

Forcing a way for shared data increases the memory bandwidth requirements, because a lot of cache misses, due to collisions, take place. However, Radix and Raytrace do not suffer significantly from these additional misses. The first is expected not to suffer from forcing a way, and invalidating the entire cache, as Radix has a fairly small problem size, consequently the number of accesses to shared data is low. The second, Raytrace, has a lower memory bandwidth requirement for forcing a way than invalidating the entire cache. This is most likely because of the high number of synchronizations, in other words, if the application cleans and invalidates the entire cache on each synchronization it suffers from more cache misses than compared to forcing a way as a result from false invalidations. Additionally, the time needed for performing a clean and invalidate entire cache takes many cycles, because of predictability and composability requirements.

Figure 7.3 shows the memory bandwidth requirements for the parallel implementations, but

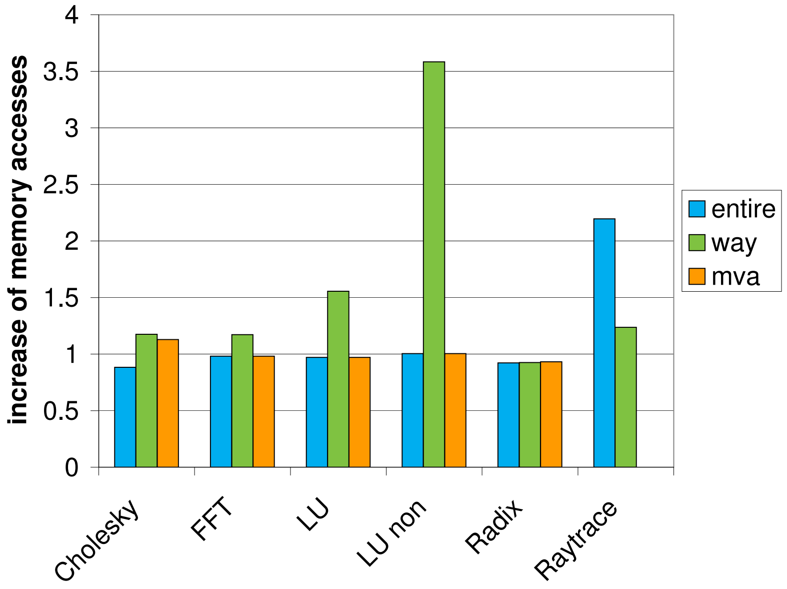


Figure 7.4: Relative increase in number of memory accesses for SPLASH2 applications

we would like to know how these values relate to the single processor execution. That is what we try to get insight in by calculating the increase in memory accesses for a parallel instance with cache coherence operations compared to the single processor instance.

The SPLASH2 applications are mapped on two processors, and on each processor the number of memory accesses is determined. The total number of memory accesses of each processor is summed to a total for the application, and this number is compared to the total number of memory accesses when the application is executed on one processor without cache maintenance operations. The difference between these two illustrates partially the overhead of the cache coherence operations. Figure 7.4 shows the increase, or decrease, in the number of memory accesses, relative to the number of memory accesses in the single processor execution, which is set to 1. The columns in Figure 7.4 represent, from left to right, the increase of memory accesses for cache maintenance operations *entire*, *way*, and *MVA*.

The shape of Figure 7.4 resembles the shape of Figure 7.3. Clearly forcing a way for shared data causes significantly more memory accesses, except for Radix, which shows a decrease in the total number of memory accesses in all three instances. The execution of Radix on two processors benefits from the increase in cache size. If then a way is forced, Radix still does not suffer from misses or collisions significantly. As a result, the total number of memory accesses for Radix decreases if the application is mapped to two processors. In the other applications is *way* a less attractive alternative, and in particular *LU non-contiguous* appears to suffer significantly from cache misses.

For several applications is the increase in cache size positive in terms of the number of memory accesses. Compared to the uniprocessor is the total number of memory accesses lower for several applications executed in parallel, even when cache maintenance operations on the entire cache are used. This is probably a result from the low number of synchronizations in combination with an increase in cache size. Surprisingly requires the parallel application with cache maintenance operations using *MVA* at least as many memory accesses as the uniprocessor implementation. We expect that this is caused by the large software loop, that is used to iterate through all MVAs in the shared region, which causes many stack accesses, that *pollute* the cache.

From Figure 7.4 it may be concluded that in cases where the number of synchronization is low, the preferable cache maintenance operation is on entire cache. If the number of synchronization becomes high, e.g., in Raytrace, then the need arises to *restrict* the cache operations. This can be done by either, forcing shared data into a way, but this has the drawback of a higher number of collisions, or, which is more promising, by adapting the programming model, e.g., FIFO communication, which enables an efficient software cache coherence protocol.

7.5 Impact on other threads

Apparently the cache coherence operations have a small impact on the performance of SPLASH2 applications itself, but of course this depends on the synchronization-to-computation ratio. In this experiment we try to estimate the impact of cache coherence operations by one thread on other threads.

FFT is executed in one thread, consequently without cache coherence operations, and a second thread is mapped to the same processor. This second thread performs one cache coherence operation on each preemption. The type of cache coherence operation (*entire cache*, *entire way*, *MVA (which operates on lines that are not used by FFT)*) is varied. The switch times are also varied, i.e., varying the synchronization-to-computation ratio.

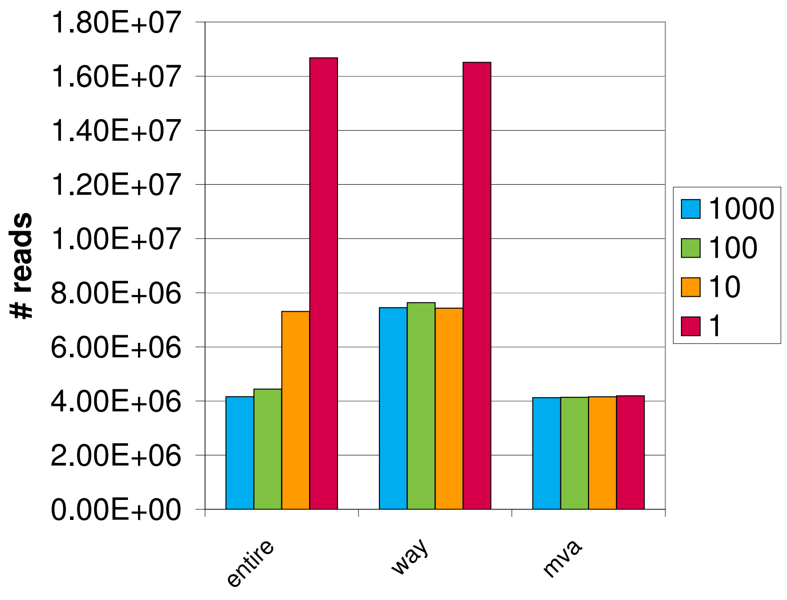


Figure 7.5: Impact of cache coherence operations on other threads

Figure 7.5 shows the number of reads of a thread executing FFT if it is interleaved with a different thread that is performing cache coherence operations on each preemption for a task

Instance	Interconnect	Stack	Shared data	T_1 (s)
(1)	Bus	SRAM	SRAM	24.56
(2)	NoC	SRAM	SRAM	28.52
(3)	NoC	SDRAM	SDRAM	30.36

Table 7.3: Execution time of FFT on different architectures

switch time in milliseconds. The columns correspond, from left to right, to a task switch time of 1000, 100, 10, or 1 milliseconds. If the switch times are large enough, the overhead of cache coherence operations is insignificant. Even in the case of invalidating entire cache the number of reads does not increase significantly with large switch times. Forcing shared data into a way has a higher impact than often invalidating the cache, as the cache coherence operations cause additional reads only for a small switch time. Invalidations based on MVA do not have a significant impact on the thread, because this cache coherence operation only invalidates lines specified by MVA that are not used by FFT. It is important to note that if FIFO communication, with the adapted programming model, would be used, then a thread should observe no overhead from other threads, like in MVA in this experiment.

For composability requirements the entire cache should be cleaned and invalidated upon each preemption. We have discarded this operation during the experiments, as it degrades the performance of the applications (especially if the switch times are small) and it prevents us from showing the impact of the synchronization-to-computation ratio in this experiment.

7.6 Impact of Æthereal NoC and SDRAM

In this experiment we will map the FFT application with the standard problem size to two different MPSoCs. In the first MPSoC are the processors directly connected to shared memory (SRAM), similar to the standard experimental setup (see Figure 4.1. In the second MPSoC are the processors connected through Æthereal NoC, and is the shared memory physically distributed as a SRAM, and a SDRAM (see Figure 4.2). Please note that because of ordering constraints the semaphores need to be put in the same physical memory as the shared data, otherwise these accesses can be reordered because of different latencies to the memories.

In this experiment we will only consider performance, and the impact of the architecture on this performance. The FFT application is executing in two threads mapped on one processor, and the cache maintenance operations are disabled. This enables us to illustrate the impact of the architecture, without the effect of cache coherence operations.

We compare three instances, see Table 7.3. The NoC increases the latency significantly, which is to a large extent because the current implementation of the NoC does not support pipelined reads. Fetching a cache lines therefore takes the same number of cycles as fetching 8 individual words. As a consequence the execution time of instance (2) is significantly higher than instance (1).

The differences between instances (2) and (3) are (i) the type of shared memory, in (2) a SRAM, and in (3) a SDRAM, and (ii) the arbiter on the memory. The arbiter for access to the SRAM is a TDM[1] arbiter, which wastes budget if a processor does not consume its

[1]Time Division Multiplexing: guarantees specific service to all processors, but wastes slice if a processor

slice, but the arbiter for the SDRAM is following a *round-robin*[2] scheme. This is expected to have a positive influence on the execution time, but the SDRAM will cause an increase in the execution time as well. From Table 7.3 is can be concluded that instance (3), with the NoC and SDRAM, indeed increases the execution time.

This experiment is used, together with the experiment in Section 7.8, as an example to show that the protocol is indeed suitable to be applied to an MPSoC with a NoC, but future work should focus on performance. As a result the MPSoC with a NoC and SDRAM will not be used in the experimental performance evaluation. The increase in the execution time, due to the increase in memory access latency, can have an impact on the protocol's performance and on the scalability. This is because a higher latency can increase the penalty for false invalidations.

7.7 Scalability

This section discusses the scalability of the software cache coherence protocol. Because the MPSoC only consists of two processors we will not perform an experiment to demonstrate the scalability, but we will discuss previous experiments that are relevant to scalability.

First of all, we will discuss scalability relative to the hardware protocols. In most of the hardware protocols threads can impact the execution of threads on other processors, and this is in particular in the case of false sharing. In this situation the scalability of an application with our software cache coherence protocol is expected to be better, because threads do not impact the execution of threads on other processors.

From experiments we can conclude that the speedup for the SPLASH2 applications on a two processor MPSoC is good, and in addition to this the memory bandwidth requirements are low. Another experiment demonstrated that the cache coherence operations do not have a significant impact on the number of memory accesses, if the synchronization-to-computation ratio is low. As a consequence, it is expected that the software cache coherence protocol does not have a negative impact on the scalability.

It is important to note that if the cache is cleaned and invalidated using MVA, which is used in FIFO communication, the scalability is expected to be good. Experiments demonstrated that, even if the synchronization-to-computation ratio is high, cleaning and invalidating based on MVA does not impact the number of memory accesses. As a consequence, we expect that FIFO communication can be used in a large MPSoC, with only small overhead. It is important to keep in mind that the efficiency of FIFO communication relies, to a large extent, on the size of the data that is being shared through the FIFO. If the size of shared data is large, then cleaning and invalidating using MVA is not anymore attractive. As a consequence we are forced to use either cache operations on the entire cache, or on an entire way, which can result in a high number of false invalidations.

False invalidations can impact the scalability. If our MPSoC is scaled, the latency to the shared memory also increases. This increased latency exposes a large penalty to cache misses, and these cache misses are, to some extent, caused by cache coherence operations.

does not consume its slice

[2]Round-robin: guarantees specific service to all processors, but if a processor does not consume its slice, then the next processor will be served

If an application suffers from a lot of false invalidation, and consequently a lot of cache misses it will not scale that well if the latency increases.

7.8 FIFO case study: MP3 decoder

In this experiment an MP3 decoder is mapped to one processor, and the decoded samples are sent to a second processor through a FIFO. The second processor executes the *playout* tasks, which sends the samples to the digital-to-analog converter. An overview of the mapping on the multiprocessor system is shown in Figure 7.6. The MPSoC, in which processors are connected through the Æthereal NoC to an SDRAM (see Figure 4.2), is used in this experiment.

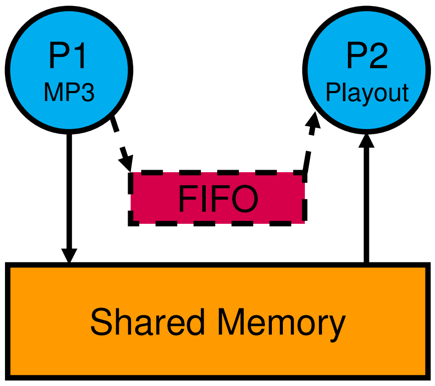


Figure 7.6: A multiprocessor MP3 decoder

The MP3 decoder produces samples, that are sent to the other processor through a FIFO. The FIFO communicates 576 words of data, which are written by the producer, and read by the consumer. As a consequence, the producer has to write and clean 72 cache lines on each iteration (if the FIFO is in a write-back cache), and the consumer has to invalidate and read 72 cache lines on each iteration. The invalidate operation using MVA takes approximately 1300 cycles for 72 cache lines. If the entire cache or an entire way had to be invalidated on each synchronization, i.e., after producing 576 samples, it would respectively take around 8000 and 2200 cycles.

This illustrates that FIFO communication can be used in our software cache coherence protocol, and that for some applications it may be beneficial to exploit FIFO communication. Additionally cleaning and invalidating using MVA has only impact on the shared data and no private data is cleaned or invalidated, which should improve the performance, compared to operating on the entire cache, or on a way.

However, it is very important to keep in mind that the tradeoff is between the number of cycles used to perform the action, and the number of false invalidations. Although cache coherence operations on entire cache, and on a way, can cause a high number of false invalidations, it can still be an attractive solution if the shared range is large, as looping through a lot of MVAs takes a high number of cycles. Fortunately, we expect that for many applications, and streaming applications in particular, the shared data can be put in separate FIFO buffers, which can improve the efficiency of the software cache coherence protocol [40].

8 Suggestions to improve the cache controller

In Section 7 we have shown that the software cache coherence protocol performance for the SPLASH2 applications can be good. However, during the project we have discovered some interesting properties that our MPSoC currently does not have. This section discusses those properties, which we would like to add to our MPSoC, and to the cache controller in particular.

In Section 6 we have discussed predictability, and it appeared that the standard cache maintenance operations on the entire cache in the ARM9 are not interruptible, which is a problem for predictability. To avoid this problem we had to use software loops to perform cache maintenance operations on lines, until the entire cache was cleaned/invalidated, which required quite some cycles. This large number of cycles can pose a large overhead on the application, especially if the synchronization-to-computation ratio is large. Ideally the ARM9 should provide an interruptible instruction to clean and invalidate the entire cache. We gradually propose several feasible solutions to provide a more efficient instruction to maintain cache coherence.

Currently we have implemented *clean and invalidate entire cache* by a software loop to clean and invalidate all lines in each way. This loop needs about 3 to 4 times more cycles than cleaning and invalidating one entire way. A first improvement could be to add (or change) an operation to clean and invalidate a line in each way in parallel which can be done in hardware (the *test clean and invalidate* already performs the cache operations in parallel according to the technical reference manual [25]). Invalidation can be performed in parallel without major difficulties, whereas cleaning in parallel may cause difficulties. Cleaning requires writing data into the write buffer, and this may be problematic to execute in parallel. An option would be performing the test to clean a line in parallel, and then performing the cleaning of the lines sequentially. Consequently cleaning and invalidating entire cache would take the same time as cleaning and invalidating one way.

ARM9 provides *(clean and) invalidate entire cache* as a single instruction. We expect this instruction to perform line based clean and invalidate operations by looping in hardware. The issue with this instruction is that when the instruction is interrupted it is unknown where the instruction will continue. ARM9 stores the address of the instruction that launched the cache operation + 4 in R14, which allows the software to restart the operation. This potentially repeatedly restarts the operation. We propose extending the instruction with a register, which represents a line (i.e., a set number for all four ways) where the instruction should (re)start. This register should be read/write, and thus can be saved and restored on a task switch, if the instruction is interrupted. It is expected that the hardware loop will significantly lower the required number of cycles.

For some applications and MPSoC architectures it might be a drawback to put shared regions in *write-through* caches. If there is no information about where shared and private data is put, then the entire memory needs to be in *write-through* caches. This problem can be avoided by clean and invalidate operations on a different granularity than cache lines, which could be for instance on *words* or even *bytes*. This improvement requires changes in the cache controller and the memory hierarchy, and we believe that in many cases this improvement will not have a significant impact on performance. Separation of shared and private data can already have a large positive impact on performance, and this separation is easy to achieve.

The last optimization is also included in ARMv6, but not in ARMv5, and it could have a significant impact on the performance. This optimization is by exploiting the prefetch operation. The prefetch operation fetches data from the shared memory and puts it in the cache parallel with the processor performing computations, and thus does not stall. This operation can be used to successfully lower the number of cycles required due to stalling because of cache misses. However, prefetching can only be exploited when it is explicit in the software which data is to be accessed soon. This is hard to achieve in the Pthreads standard, but can be quite easily done in case of FIFO communication. On an *acquire* of a place in the FIFO buffer the range for that place can be prefetched, which is expected to lower the number of cycles due to stalling on the first read. Prefetching could also be beneficial to Pthreads if the programming model is extended by relating accesses to synchronization points.

9 Conclusions

This thesis presented a tuneable software cache coherence protocol that is highly suitable for heterogeneous multiprocessor system-on-chip (MPSoC) architectures with a network-on-chip (NoC). The main advantage of this software cache coherence protocol is that it can be applied in a broad range of MPSoCs with only little effort. The software cache coherence protocol ensures that caches are coherent on synchronization points, which is sufficient to support Release Consistency, on top of which we have implemented a part of the standard Pthreads communication library. The protocol is applicable to an MPSoC with many off-the-shelf embedded processors and caches, and it is irrelevant whether those caches support hardware cache coherence.

The software cache coherence protocol requires software to contain explicit synchronization. In particular, the caches are guaranteed to be coherent on synchronization points. This poses the restriction that our MPSoC is limited to executing software with explicit synchronization, but we expect that this does not significantly restrict the applicability of the software cache coherence protocol, as most parallel programs rely on synchronization to guarantee correct behavior.

The software cache coherence protocol also poses some constraints on the hardware. Processors with caches need to be able to control the contents of their cache through clean and invalidate instructions. Currently we exploit software locks, and both the Æthereal NoC and memory ensure that the order between memory accesses and memory accesses to the software locks are not reordered. This last constraint can be relaxed, but then all writes have to be acknowledged to ensure that the semaphores are updated in the shared memory before (shared) reads and writes are issued, and vice versa.

The software cache coherence protocol is designed to be applicable in a *predictable* and *composable* MPSoC. This requires that cache coherence operations are interruptible, and that cache coherence operations are local to a thread, in other words, cache coherence operations in a thread do not impact the execution of threads on other processors.

In addition to the software cache coherence protocol we have identified several optimizations to increase the performance of the protocol. It is important to provide separate address ranges for private and shared data, which enables efficient cache utilization. Furthermore, cache coherence operations can be limited to the shared address range by, e.g., performing cache coherence operations on a specific cache way, or using modified virtual addresses

(MVA) to loop through the entire shared address space. Furthermore, for some applications it may be beneficial to require a specific programming model which improves the efficiency of the software cache coherence protocol. A suitable adapted programming model restricts interprocessor communication to sharing data through First-In-First-Out (FIFO) buffers.

The software cache coherence protocol is evaluated in an ARM9 MPSoC which is mapped on a Xilinx Virtex 4 FPGA. Several applications from the SPLASH2 benchmark set [41] are executed in parallel on the MPSoC. From experiments we have concluded that it is important to provide a separate address range for shared data. As a consequence private data can be *efficiently* cached, namely private data can be put in a write-back cache, because writes to private data do not have to become visible to other processors. Shared data can be put in a write-through cache, which avoids the sharing problem in software cache coherence protocols, while caching of shared data can lead to a performance gain.

SPLASH2 applications have been used for the experimental performance evaluation. In this evaluation we have tried to estimate the overhead of the cache coherence protocol, in terms of execution time, and the number of memory accesses. Furthermore, we have estimated the impact of the software cache coherence protocol on scalability, and we have identified several optimizations to tune the protocol. In most experiments we have compared the performance of cache coherence operations on (i) entire cache, (ii) a specific way, (iii) shared address range using MVA.

The overhead of the protocol for the SPLASH2 applications is surprisingly low, because the speedup for most SPLASH2 applications is between 1.89 and 2.01 on a two processor MPSoC. The cache coherence operation on the entire cache had the best performance, in terms of speedup, in most cases. This is most likely a result from the low synchronization-to-computation ratio and the disappointing performance of the other alternatives: clean and invalidate way, and looping through the entire shared address space using MVA.

The memory bandwidth requirements for the SPLASH2 applications were low, around 2% when the applications were executed in parallel on two processors. In addition, we have compared the total number of memory accesses of the application executed on one processor, to the total number of memory accesses of the multiprocessor implementation. The total number of accesses of the multiprocessor implementation was comparable to the uniprocessor execution, except when a way was forced for shared data, or when the application had a high synchronization-to-computation ratio (Raytrace). In the latter two cases the number of memory accesses showed a significant increase.

In experiments we have shown that, if no measures for maintaining composability are taken, the cache coherence operations can impact the execution of other threads on the same processor. The impact of the cache coherence operation is highly dependent on the synchronization-to-computation ratio. It was shown that cache coherence operations on the entire cache had a small impact on the number of memory accesses if there was only a small synchronization-to-computation ratio, as this ratio became larger, the impact also increased, i.e., causing more reads due to cache misses. It is important to note that cache coherence operations using MVA, that is likely to be used in FIFO communication, has almost no impact on the execution of other threads, as long as the lines that are to be invalidated are not used by the other threads.

We have also discussed the scalability of the software cache coherence protocol. The protocol is expected to be scalable, because cache coherence operations are local to a thread, and do never impact the execution of threads on other processors. The impact of a thread on threads on the same processor can be eliminated by taking measures for composability, e.g., clean and invalidate entire cache on each task switch. Another metric that is important for scalability is the memory bandwidth requirement, for the SPLASH2 applications this requirement was low, which can result in good scalability. In addition to this, we have demonstrated that the memory bandwidth requirement does not significantly increase, as long as the synchronization-to-computation ratio is low. In cases where the synchronization-to-computation ratio is high, it may be beneficial to exploit FIFO communication.

FIFO communication was presented as an optimization of our software cache coherence protocol, and illustrated using a multiprocessor MP3 decoder. In FIFO communication, it is known which memory locations are going to be read in a critical section, and the cache coherence operations can be limited to this range. As a result, there are only few false invalidations. In addition, in FIFO communication it is sufficient to only clean the range if the thread writes to the buffer, and only invalidate the range if the thread is reading from the buffer.

10 Future work

The different variants of our software cache coherence protocol have been implemented on the ARM9 MPSoC. This section will discuss directions for future work, some of these directions will consider improvements of the cache coherence protocol, and others will be in the direction of extending the MPSoC.

The experimental performance evaluation can be improved to draw more solid conclusions from the evaluation. Currently we have only evaluated the performance using Splash2 applications, that apparently have a low synchronization-to-computation ratio. It could be desirable to use applications from, e.g., the ALPBench benchmark suite to improve the performance evaluation.

The FPGA implementation of the ARM9 also requires some additional work. The current implementation was significantly extended and improved during this project, but only little attention has been paid to performance. In future work this may become important, e.g., the implementation of Æthereal network can be improved by supporting pipelined memory accesses, and maybe the ARM9 could be extended with hardware floating point support.

The current implementation of the MPSoC does not support hardware locks. As a consequence, a software lock algorithm for mutual exclusion is used for synchronization. Furthermore, the administration of FIFO buffers is maintained in software. The variables needed for the software locks and the FIFO administration are located in an noncacheable range of the shared memory. As a consequence, threads spinning on a lock cause many memory accesses. This can significantly increase the memory bandwidth requirements, and can be an issue for scalability. Future work should look into methods for predictable and composable locking, without significantly increasing the memory bandwidth requirements.

In future work it may be interesting to evaluate the scalability by applying the protocol in a large scale MPSoC. Unfortunately, only two ARM9 processors can be mapped to the Xilinx Virtex 4 FPGA. It can be an option to map a large number of MicroBlaze soft core processors on the FPGA. Evaluating scalability can be related to load balancing; in this thesis we have briefly discussed load balancing, and we have illustrated that the current software platform and software cache coherence protocol can support load balancing. However, it is not yet clear how to decide whether tasks need to be migrated.

The software cache coherence protocol is suitable for composable MPSoCS. However, to eliminate the impact of one thread on other threads executing on the same processor, it is required to clean and invalidate the entire cache on each task switch. Unfortunately, this can significantly degrade the performance, and performance becomes dependent of the task switch time. Future work should focus on more efficient ways to ensure composability.

References

[1] The posix threads standard. *ISO/IEC standard 9945-1:1996, also known as ANSI/IEEE POSIX 1003.1-1995.*

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide.* http://www.intel.com/products/processor/manuals/, 1989.

[3] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of hardware and software cache coherence schemes. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 298–308, 1991.

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

[5] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268, New York, NY, USA, 2005. ACM.

[6] D. Borodin and B.H.H. Juurlink. A low-cost cache coherence verification method for snooping systems. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 219–227, Sept. 2008.

[7] Celoxica. www.celoxica.com.

[8] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *Computers, IEEE Transactions on*, C-27(12):1112–1118, Dec. 1978.

[9] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. *SIGARCH Comput. Archit. News*, 35(2):278–289, 2007.

[10] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1999.

[11] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[12] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *SIGARCH Comput. Archit. News*, 14(2):434–442, 1986.

[13] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *EMSOFT '08: Proceedings of the 7th ACM international conference on Embedded software*, pages 255–264, New York, NY, USA, 2008. ACM.

[14] Marcus Ekerhult. Compose, design and implementation of a composable and slack-aware operating system targeting a multi-processor system-on-chip in the signal processing domain, 2008.

[15] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[16] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[17] J.R. Goodman. *Cache Consistency and Sequential Consistency.* Tech. Report no. 61, University of Wisconsin-Madison, Computer Science Dept., 1989.

[18] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1–24, 2009.

[19] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2006.

[20] Chris Holt, Mark Heinrich, Jaswinder P Singh, Edward Rothberg, and John Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical report, Stanford, CA, USA, 1995.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[22] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[23] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*, pages 241–251, 1997.

[24] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, Mar 1992.

[25] ARM Limited. Arm technical reference manual. 2005.

[26] Kenneth L. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 179–195, London, UK, 2001. Springer-Verlag.

[27] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, 1995.

[28] Gary L. Peterson. A new solution to lamport's concurrent programming problem using small shared variables. *ACM Trans. Program. Lang. Syst.*, 5(1):56–65, 1983.

[29] F. Petrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in noc based shared memory multiprocessor soc architectures. *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 53–60, 0-0 2006.

[30] Fong Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *Parallel and Distributed Systems, IEEE Transactions on*, 6(8):773–787, Aug 1995.

[31] E. Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization.* PhD thesis, Stanford University, 1992.

[32] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *ISCA '87: Proceedings of the 14th annual international symposium on Computer architecture*, pages 234–243, New York, NY, USA, 1987. ACM.

[33] C. E. Scheurich. *Access ordering and coherence in shared memory multiprocessors.* PhD thesis, University of Southern California, Los Angeles, CA, USA, 1989.

[34] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *Computer*, 27(7):45–55, 1994.

[35] Ulrich Stern and David L. Dill. Automatic verification of the sci cache coherence protocol. In *CHARME '95: Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, London, UK, 1995. Springer-Verlag.

[36] T. Suh, D.M. Blough, and H.-H.S. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1150–1155 Vol.2, Feb. 2004.

[37] I. Tartalja and V. Milutinovic. An approach to dynamic software cache consistency maintenance based on conditional invalidation. *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, i:457–466 vol.1, Jan 1992.

[38] I. Tartalja and V. Milutinovic. A survey of software solutions for maintenance of cache consistency in shared memory multiprocessors. *Hawaii International Conference on System Sciences*, 0:272, 1995.

[39] Igor Tartalja and Veljko Milutinovic. Classifying software-based cache coherence solutions. *IEEE Softw.*, 14(3):90–101, 1997.

[40] J.W. van den Brand and M. Bekooij. Streaming consistency: a model for efficient mpsoc design. *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 27–34, 2007.

[41] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*, pages 24–36, Jun 1995.

[42] Steven C Woo, Jaswinder P Singh, and John L. Hennessy. The performance advantages of integrating message passing in cache-coherent multiprocessors. Technical report, Stanford, CA, USA, 1993.

[43] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 219–229, New York, NY, USA, 1994. ACM.

[44] F. Zandvelt. On caches for a (rt) multiprocessor environment. *Unpublished*, prepared for the VSI/OCB working group, Apr. 1998.