A tuneable software cache coherence protocol for heterogeneous MPSoCs

Citation for published version (APA):
Ophelders, F. E. B., Bekooij, M. J. G., & Corporaal, H. (2009). A tuneable software cache coherence protocol for heterogeneous MPSoCs. In Embedded Systems Week 2009, ESWEEK 2009 - 7th IEEE/ACM International Conference on Hardware/Software-Co-Design and System Synthesis, CODES+ISSS 2009, 11 October - 16 October 2009, Grenoble (pp. 383-392). New York: Association for Computing Machinery, Inc. DOI: 10.1145/1629435.1629488

DOI:
10.1145/1629435.1629488

Document status and date:
Published: 01/01/2009

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

A Tuneable Software Cache Coherence Protocol for Heterogeneous MPSoCs

Frank Ophelders[1] Marco J.G. Bekooij[2,3] Henk Corporaal[1]
[1]Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands
[2]NXP Semiconductors, Eindhoven, The Netherlands
[3]Department of EEMCS, University of Twente, The Netherlands
frank.ophelders@gmail.com, marco.bekooij@nxp.com, h.corporaal@tue.nl

ABSTRACT

In a multiprocessor system-on-chip (MPSoC) private caches introduce the cache coherence problem. Here, we target at heterogeneous MPSoCs with a network-on-chip (NoC). Existing hardware cache coherence protocols are less suitable for MPSoCs because many off-the-shelf processors used in MPSoCs do not support these protocols. Furthermore, these protocols typically rely on global visibility and serialization of writes which does not match well with the parallel point-to-point communication provided by a NoC. Therefore, we propose a software cache coherence protocol, which can be applied in a heterogeneous MPSoC with a NoC. The software cache coherence protocol relies on explicit synchronization in the software. More specifically, caches are guaranteed to be coherent according to the Release Consistency model, on top of which we have implemented the standard Pthreads communication library. Heterogeneous MPSoCs with off-the-shelf processors can easily be supported, because processors are only required to provide *cache control operations*, e.g., clean and invalidate. All cache coherence operations are interruptible and do not impact the execution of tasks on other processors, therefore this protocol is suitable for predictable MPSoCs. Our software cache coherence protocol is implemented on an ARM926EJ-S MPSoC which is mapped on an FPGA. From experiments we conclude that the protocol overhead is low for the applications taken from the SPLASH-2 benchmark set. For these applications we observed a speedup between 1.89 and 2.01 on the two processor MPSoC.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*

General Terms

Design, Performance, Reliability

1. INTRODUCTION

In this paper, we consider heterogeneous multiprocessor systems on chip (MPSoC) with a network-on-chip (NoC). The processors in the MPSoC have private caches and processors communicate through reading and modifying the shared memory. An example MPSoC with n processors is shown in Figure 1.

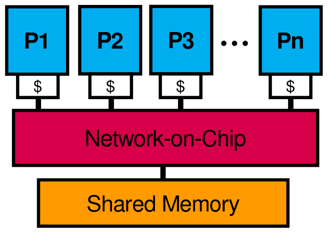


Figure 1: MPSoC with a NoC and shared memory

MPSoCs in which processors with private caches communicate through shared memory require a *cache coherence* protocol and a *memory consistency* model. A *cache coherence* protocol ensures that processors observe the most recent data, either in their cache or in the shared memory. A *memory consistency* model defines constraints on *when* writes become visible to processors with respect to other writes, therefore memory consistency and cache coherence are related. Cache coherence protocols can roughly be classified in a hardware and a software class.

A hardware cache coherence protocol, called *snooping based*, relies on all caches to "snoop" the interconnect and take appropriate actions based on transactions on the interconnect [4]. 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 [4], to function correctly the MPSoC needs to support two properties [4]. First of all, all memory accesses should be observable by all processors (*write propagation*). Secondly, all memory accesses to a location should be observed by every processor in the same order (*write serialization*). These properties are easily supported in an MPSoC with a bus, because of the nature of the bus. However, in an MPSoC with a NoC it is difficult to support these properties efficiently. A NoC handles, for performance reasons, memory accesses as point-to-point transactions in parallel. Therefore these transactions are usually not observable by all processors. In addition, processors can observe different latencies to memories, which makes it difficult to guarantee one single order of writes (write serialization) being observed by all processors.

A second hardware cache coherence protocol, called *directory based*, exploits a directory that stores information about the status of caches for each location in the shared memory. The idea is as follows; a directory is consulted before writing to a shared location X. 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.

A directory-based cache coherence protocol is intended for MPSoCs with a NoC. However, according to [10] 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. Furthermore, 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 [13]. An example of a hardware/software methodology to ensure cache coherence in heterogeneous MPSoC is proposed in [13], but it can only be applied when *all* caches support hardware cache coherence.

We are not aware of efficient hardware cache coherence protocols for heterogeneous MPSoCs with a NoC. In contrast to this, software cache coherence protocols have the potential to be a scalable cache coherence protocol that is suitable for heterogeneous MPSoCs with a NoC. It is essential to understand that software cache coherence protocols typically do not ensure cache coherence on the granularity of individual memory accesses, but on groups of memory accesses. Therefore, software cache coherence protocols usually guarantee a relaxed memory consistency model. The protocols rely on processor instructions to control the contents of their cache. Important for a NoC based MPSoC; software cache coherence protocols do not require global visibility of writes.

Typically software cache coherence protocols rely on explicit synchronization. In particular, the caches are guaranteed to be coherent on synchronization operations. 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 our software cache coherence protocol, as many parallel programs rely on synchronization to guarantee correct behavior [5]. Typically, in the embedded systems domain applications belonging to this class can be selected at design-time.

Transactional Memory is intended to improve parallel programming and it attempts to solve cache coherence and memory consistency issues. Transactional Memory groups memory accesses in transactions, and these transactions are executed in parallel. Memory accesses are not visible to other processors, until a transaction *commits*. The *commit* operation checks whether any other transaction modified any location that has been read or modified by the currently committing transaction. If so, one of the transactions is aborted and restarted. Besides complex hardware support for rolling back and comparing transactions, we see an additional issue. This issue concerns *predictability*, which is important in the design of MPSoCs, but predictable speculative execution appears to be challenging [6].

Concluding, we observe several issues in the design of cache coherence protocols for MPSoCs with a NoC. Current hardware cache coherence protocols are not well suited for a NoC based MPSoC, because write serialization is difficult to guarantee efficiently. In addition to this, designing a hardware cache coherence protocol for heterogeneous MPSoCs is a challenging task. Alternatively, existing software cache coherence protocols are also not always applicable as these usually require a specific programming model, instead of a widely used programming model like Pthreads [1].

Contributions of this paper: This paper presents a software cache coherence protocol that is highly suitable for heterogeneous MPSoCs with a NoC. The protocol is applicable to off-the-shelf processors. These processors are not required to support hardware cache coherence. The software cache coherence protocol ensures that caches are coherent on synchronizations, which is sufficient to support Release Consistency [7], on top of which standard communication libraries, e.g., POSIX threads (Pthreads) and OpenMP can be implemented. More specifically, we have embedded the protocol in Pthreads.

The software cache coherence protocol is designed to be applicable in a predictable MPSoC. All cache coherence operations are interruptible and local to a processor. Therefore cache coherence operations do not impact the execution of tasks on another processor. This is different from many hardware cache coherence protocols, where caches respond to, e.g., invalidation requests from other processors.

The protocol is evaluated in an ARM926EJ-S MPSoC which is mapped on an FPGA. Several applications from the SPLASH-2 benchmark set [17] are executed in parallel on the MPSoC. The overhead of the protocol is low for the evaluated SPLASH-2 applications, because the speedup observed 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, for private data and shared data different and potentially more efficient cache policies can be applied. Furthermore, 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. An attractive programming model would be one that restricts interprocessor communication through First-In-First-Out (FIFO) buffers.

Organization of this paper: The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we give a brief introduction to cache coherence and memory consistency. Two popular memory consistency models are discussed, and we discuss why Sequential Consistency [8] is not well suited for MPSoCs with a NoC. In Section 4, we present our tuneable software cache coherence protocol. In addition we discuss tradeoffs, identify several optimizations, and discuss how to embed the protocol in Pthreads. In Section 5 we discuss characteristics of the MPSoC on which we have implemented the software cache coherence protocol. Then, in Section 6 we evaluate the performance of the protocol. Lastly, Section 7 gives concluding remarks and presents directions for future work.

2. RELATED WORK

This section discusses related work, and considers software oriented solutions to cache coherence.

In [14] and [12] several software cache coherence protocols are proposed. These protocols ensure that caches are coherent on explicit synchronization points in the software. More specifically, the shared address range is divided in segments and shared segments are only accessed in a critical region. The required cache coherence operations are performed on the entry and exit procedures of a critical region.

Both [14] and [12] require explicit coupling of a critical region and accesses to specific parts of shared data. This coupling is, to the best of our knowledge, not explicit in widely used programming models. In addition to this, the mapping of shared data on the memory is known. Furthermore, both protocols require a software administration which is used to determine whether a processor can have a valid copy of the shared data.

Our software cache coherence protocol has three major differences. Firstly, our protocol does not require a coupling between synchronization operations and shared data. However, we do acknowledge that coupling shared data and synchronization operations can improve the protocol efficiency and we propose a specific programming model, *FIFO communication*, as an optimization. Secondly, in our protocol shared data can be scattered throughout the entire address range. Thirdly, our protocol does not require a software administration to guarantee correct behavior.

Cache coherence issues for NoC based MPSoCs are discussed in [11]. They propose to provide separate address ranges for shared and for private data. Private data can be cached, because it does not have the cache coherence problem. Shared data is put in a noncacheable region, consequently avoiding the cache coherence problem. However, putting shared data in a noncacheable region is very likely to result in performance degradation, and is therefore not always favorable. We see the separation of private and shared data as an optimization. We provide means to keep shared data in a cacheable region, while still ensuring cache coherence.

In [16] a novel memory consistency model, Streaming Consistency, is proposed. Streaming Consistency targets at the streaming application domain. In [16] also an efficient software cache coherence protocol is proposed. The protocol relies on using FIFO buffers for all interprocessor communication. Hence, the software cache coherence protocol requires a specific programming model.

In our software cache coherence protocol we propose FIFO communication as an optimization, and we discuss an efficient software cache coherence protocol for FIFO buffers. Our protocol does not require all interprocessor communication through FIFO buffers, and can therefore be applied in combination with our basic software cache coherence protocol, while still improving the performance.

3. CACHE COHERENCE AND MEMORY CONSISTENCY

In this section we give a short introduction in cache coherence and memory consistency. We give examples of cache coherence and memory consistency issues and we discuss two memory consistency models.

3.1 Cache coherence

This section discusses *cache coherence* in MPSoCs [4]. Intuitively a memory with cache hierarchy holds values, and on a read, a memory returns the last value written to it. Unfortunately if in an MPSoC processors have a private cache, without taking precautions, danger exists that one may see stale values in its cache. The cache coherence problem is illustrated in Figure 2.

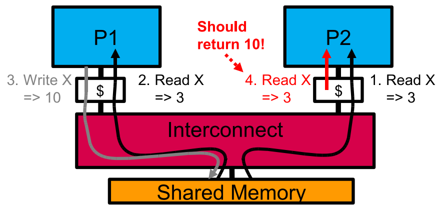


Figure 2: Example cache coherence problem

Cache coherence is essential if data is being shared between processors. Cache coherence has two properties: *write propagation* and *write serialization*. *Write propagation* means that writes become visible to other processors. *Write serialization* means that all writes to a single location (from one or multiple processors) are observed in the same order by all processors.

3.2 Memory consistency

A memory consistency model is required, because it enables the programmer to reason about outcome of the program. Often in parallel programs a read returns the value of a particular write; therefore there should be a known *order* between memory accesses to different locations. However, cache coherence only ensures a certain order among accesses to a single location. A memory consistency model specifies constraints on the order in which memory operations must appear to be performed (i.e., become visible to other processors) with respect to one another.

Consider the example in Algorithm 1, in which three processors read and modify shared variables. The programmers intention is that processor *P3* prints value 1, written by *P1*. Unfortunately, this is not necessarily true, as memory accesses can be reordered by the compiler, memory controller, NoC, and the processor; e.g., reads can overtake writes that are put in a write buffer.

Memory consistency and cache coherence are related in the sense that memory consistency subsumes cache coherence [4], because it specifies constraints on the order of memory accesses to a single and to different locations, which may be issued by multiple processors. Cache coherence only ensures that eventually writes become visible, which is required

Algorithm 1 Importance of write atomicity

Require: A and B are initially 0

```
P1              P2                  P3

A ← 1           while A = 0 do      while B = 0 do
                    skip                skip
                end while           end while
                B ← 1               print A
```

for interprocessor communication. A cache coherence protocol, in cooperation with the hardware and the compiler, supports a certain memory consistency model, which is required by the programmer to reason about outcome of software executed on the MPSoC.

The remainder of this section is organized as follows. First we will discuss the most strict memory consistency model, Sequential Consistency, followed by a memory consistency model, Release Consistency, which allows more reordering of memory accesses, but poses some requirements to the software. Other memory consistency models are not discussed in this paper, but more information can for instance be found in [2, 4].

3.2.1 *Sequential consistency*

Lamport formalized a strict model, called Sequential Consistency [8]. Sequential Consistency requires memory accesses to be completed in program order. Every process appears to issue and complete memory accesses one at a time and atomically in program order [4]. Writes issued by a processor are observed by all processors in one single order; which is called *write atomicity*. This is intuitive for the programmer, but poses restrictions on hardware and compiler optimizations [2].

Especially *write atomicity* poses restrictions on the hardware. We are targeting at MPSoCs with a NoC, and processors can observe different memory latencies in a NoC (see Figure 3). Therefore it is difficult to efficiently guarantee Sequential Consistency in a NoC.

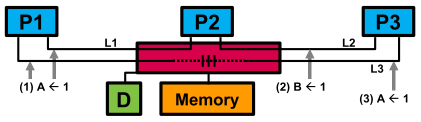


Figure 3: Write atomicity issues in a network-on-chip

Imagine Algorithm 1 is executed on the MPSoC shown in Figure 3. Assume all processors propagate, with different latencies, all writes to other processors, furthermore $L3 > L1 + L2$. Then, it is possible that $P3$ receives the write to A (3) by $P1$ later than it receives the write to B (2) by $P2$. Consequently, one single order of writes is not maintained and write atomicity is violated.

A, likely inefficient, solution to ensure *write atomicity* in a NoC is by adding a separate arbiter, called D, which serializes accesses to memory locations. However, this increases the memory access latency and causes additional traffic. An example of an MPSoC which relies on an arbiter to provide write atomicity is SGI Origin according to [4, 9].

3.2.2 *Release consistency*

Release Consistency is introduced in [7]. This model relies on distinction between types of shared memory accesses, and different ordering requirements can apply to different types of memory accesses. Two accesses are *conflicting* if they are to the same memory location, and at least one of the accesses is a write. A conflicting access is *competing* if two conflicting accesses are not ordered, they may be issued simultaneously and thus causing a race condition.

A conflicting access can be made non-competing by using synchronization. Release Consistency specifies two types of synchronization accesses; *acquire* and *release*.

A set of sufficient conditions for Release Consistency is as follows (taken from [7]):

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

Acquire and release accesses are processor consistent [4], this means that each processor issues these accesses in program order, but acquire and release accesses issued by different processors may be observed in a different order by different processors.

The conditions for Release Consistency give ordering requirements between non-competing and competing accesses, and between competing accesses. These ordering constraints are shown in an example program in Figure 4. There are no ordering requirements between non-competing accesses. Consequently caches are only guaranteed to be coherent on the competing accesses, e.g., acquire and release accesses.

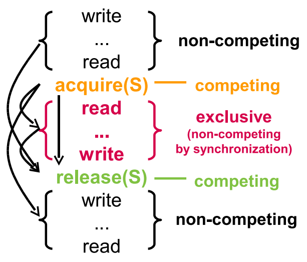


Figure 4: Ordering constraints for Release Consistency

According to Release Consistency, it is sufficient to perform cache coherence operations only on synchronizations and enforce an order between memory accesses and these synchronization accesses, e.g., acquire and release. This is in contrast to Sequential Consistency, which requires cache coherence operations on each individual write access. In the following section we will discuss a cache coherence protocol that ensures coherent caches, and provides Release Consistency. The Release Consistency model is sufficient to support Pthreads. In addition we present what is required to embed the protocol in a POSIX thread library.

4. TUNEABLE SOFTWARE CACHE COHERENCE PROTOCOL

This section presents our software cache coherence protocol, which is suitable for heterogeneous MPSoCs with a NoC. First, we will describe the basic software protocol, which poses minimal requirements to the software and hardware. Following, we will discuss several optimizations to increase the performance of the cache coherence protocol.

The protocol can be implemented on off-the-shelf processors, as long as these processors support *cache control operations*. *Cache control operations* are for instance, (i) *clean*, which copies modified cache lines back to the shared memory, (ii) *invalidate*, which causes a cache miss the next time a specific cache line is accessed. If a memory location is read, a preceding invalidate causes the read to return values from the shared memory, instead of values from the cache. Many processors like ARM, MIPS, MicroBlaze, and TriMedia support these operations.

The protocol relies on explicit synchronization operations in the software. Cache control operations are added to these synchronization points, according to the Release Consistency model. These cache control operations will be called *cache coherence operations* in the remainder of this paper because these operations are applied to guarantee cache coherence.

We will classify data as *private* or *shared* data. *Private* data is all data that is local to a process, and consequently writes to private data do not need to become visible to other processes. *Shared* data is all data that is communicated between processes, and includes the global variables.

4.1 Basic software cache coherence protocol

The basic software cache coherence protocol presented in this paper does not require a specific mapping of private and shared data in the memory. As a result shared data can be scattered throughout the address range and mixed with private data. Consequently all cache control operations should operate on the entire cache.

Assume all caches in the MPSoC are in write-through mode. Then, the basic software cache coherence protocol works as follows. The software cache coherence protocol relates cache coherence operations to synchronization operations. Two synchronization operations are presented, these are *acquire* and *release* and these correspond to synchronization accesses in Release Consistency. It is important to note that cache coherence operations are only performed on synchronization operations. Consequently caches are only ensured to be coherent with the shared memory on these synchronization operations. Acquire and release operations can be used to construct a critical section, therefore *acquire* obtains a lock, and *release* releases the lock.

After performing an *acquire* it is guaranteed that the most recent data will be accessed. In our protocol this means that acquire should invalidate all cache lines that can hold shared data, because the shared memory is up-to-date if no process is currently executing a critical section. In the basic protocol all cache lines can hold shared data, consequently the cache coherence operations operate on the entire cache.

A *release* ensures that all previous issued writes become visible to other processors. We have assumed that all caches in the MPSoC are in write-through mode, thus the release only has to ensure that all previous issued writes have updated the shared memory before the release completes.

Figure 5 shows which actions are initiated by the protocol if two processors execute an example program like shown in Figure 4.

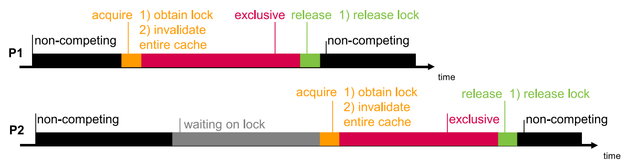


Figure 5: Example software cache coherence protocol with only write-through cacheable regions

Keeping all data in write-through cacheable regions can result in a high memory bandwidth requirement. Therefore, it is desirable to also support a write-back cache policy. However, using write-back cacheable regions results in several issues, such as the *sharing* problem and ensuring that writes are not lost on an acquire.

A write-back cacheable region introduces the *sharing* problem in software cache coherence protocols. This problem is caused by the granularity of cache coherence operations; which usually operate on entire cache lines. The sharing problem can arise if different processors modify memory locations that map to the same cache line, thus these locations share a cache line. Assume two processors *P1* and *P2* read two memory locations *A* and *B*, consequently holding identical values in their caches. Then *P1* modifies location *A*, while *P2* modifies location *B*. In this situation both lines are partially up-to-date. If both processors clean the cache line, in any order, one of the two modified locations will be overwritten by its old value. Clearly this can lead to incorrect behavior.

Different approaches are used to solve the sharing problem for private and shared data. Private data covers the private stack for each process and private data allocated on the heap. If we ensure that stacks for processes are *cache line aligned*, in other words, it is not allowed to map parts of two stacks to the same cache line, the sharing problem is nonexistent. Similarly, each process has a private local heap and these heaps are mapped on the address range such that no two heaps share a cache line.

Ensuring cache line alignment for shared data is likely to be inefficient and might be challenging to achieve in comparison to private data. Therefore solving the sharing problem for shared data requires a different approach. One way is to adapt the granularity of cache coherence operations by *sub-blocking*, which uses a *bit per byte* in the cache to identify the modified bytes, and only these bytes are updated in the memory. This is for instance supported by TriMedia processors [15]. Unfortunately this solution is not supported by all processors, e.g., ARM926EJ-S processors do not support sub-blocking.

A different solution to avoid the sharing problem for shared data is to use a *write-through cacheable region* for shared data. In a write-through cacheable region all writes are propagated to the memory, and only the bytes that are written are updated. This is likely to be a suitable solution in many cases, but it can be inefficient in combination with an SDRAM that works on a page access granularity.

A second issue with write-back caches is guaranteeing that writes issued preceding an acquire are never lost. In the basic protocol an acquire only performs an invalidate operation, but if a region of the memory is write-back cacheable, an additional cache coherence operation is required. This operation cleans all modified cache lines, before any cache line is invalidated on an acquire. This guarantees that writes to a write-back region are not lost.

4.2 Optimizations

This section presents several optimizations to increase the performance of the basic software cache coherence protocol. The first optimization considers the separation of private and shared data, which enables exploitation of different cache policies for different regions of the memory. The second optimization lowers the impact of cache coherence operations on private data. The third optimization can increase the performance of the cache coherence protocol, but requires a specific programming model.

Separation of private and shared data. In the basic software cache coherence protocol private and shared data are potentially scattered throughout the entire address range. As a result, cache coherence operations are required to operate on the entire cache, which potentially operates on mainly private data that is not required to be cache coherent. Additionally to avoid the sharing problem, the entire memory has to be put in write-through cacheable regions. This is because shared data can be stored at any location in the memory, consequently causing a high memory bandwidth requirement.

Separating dynamic data in private and shared data requires two heaps, one heap is used for private data, and a second for shared data. Two different *malloc* operations are provided to the programmer to allocate memory on one of the two heaps.

An important optimization that can increase performance exploits the separation of the address range in a private range and a shared range. This separation enables putting private data in a write-back cacheable region, because writes to private data are not required to become visible to other processors. Keeping private writes in a write-back cache successfully lowers the memory bandwidth requirement in many applications. Remember that the sharing problem for private data can be solved by ensuring cache line alignment of the private stacks and heaps.

Writes to shared data need to become visible to other processors, and the sharing problem can be solved by putting shared data in a write-through cacheable region. It is important to note that the standard ARM926EJ-S processor supports to apply different cache policies to different regions of the memory. Off course, an acquire is still required to perform an invalidate operation on shared data.

Furthermore, the separation of private and shared data enables *selective* cache coherence operations, because these operations are only required to be performed on shared data. A *selective* cache coherence operation only operates on the shared regions of the memory, instead of operating on the entire cache.

Put shared data in a certain cache way. We have implemented our software cache coherence protocol on an ARM926EJ-S processor, but unfortunately our ARM9 does not support instructions to operate on only a range of the entire address range. Therefore, we have implemented an operation that guarantees to keep the caches coherent, while it lowers the impact on private data in comparison to operating on the entire cache.

We propose a mixed hardware and software approach to force shared data into a certain cache way. The ARM926EJ-S cache consists of four cache ways. The cache controller is adapted to put all shared data, that is mapped to a certain region of the memory, in a specific cache way (see Figure 6). In software it is guaranteed that all shared data is mapped on the shared region of the memory. Private data, on the other hand, can be put in any of the four cache ways.

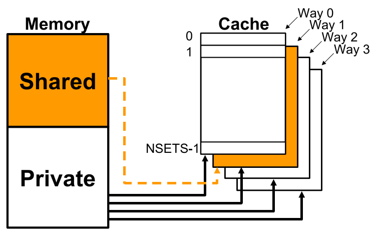


Figure 6: Put shared data in a certain cache way

If shared data is forced in one cache way, then the cache coherence operations only need to operate on this specific cache way. This has the advantage that less private data is invalidated on an acquire, because private data is also put in the other three cache ways, and these ways are not invalidated on an acquire.

A disadvantage of forcing a way for shared data is that the cache size and associativity have decreased for shared data. This can, for some applications, lead to performance degradation because a higher number of cache misses and collisions occur. Allowing more cache ways to be used for shared data increases the impact of invalidation operations on private data, which is therefore not considered as a solution.

FIFO communication. Our software cache coherence protocol for FIFO communication is different from the protocol presented in [16]. In our protocol it is not required to have *all* interprocessor communication through FIFO buffers, consequently our protocol for FIFO communication can be applied simultaneously with other variants of our software cache coherence protocol.

In FIFO communication, shared data is communicated in critical regions through FIFO buffers which are explicitly coupled to these regions. Therefore, clean and invalidate operations can easily be added to entry and exit procedures of the critical regions. More specifically, a FIFO buffer has one producer and one or more consumers. A producer only needs to perform a clean operation after the process has written to the buffer. A consumer only needs to perform an invalidate operation before reading from the buffer. Clearly this results in less protocol overhead in comparison to the basic software cache coherence protocol, where all processes were required to perform clean and invalidate operations on synchronizations.

The software cache coherence protocol for FIFO communication is more selective than the basic software cache coherence protocol. On entry of a critical region it is known which memory locations will be accessed. Only the cache lines holding copies of these locations are required to be coherent.

The producer acquires mutual exclusive access to a token, before it writes to that token in the buffer. After modifying the token, all cache lines holding a copy of the token will be cleaned. Note that the clean operation is only required if the buffer is put in a write-back cacheable region. Following the clean operation the lock for the token will be released, allowing the consumer to read the token.

The consumer first obtains mutual exclusive access to the next token to read. Then, preceding the read access all lines holding a copy of the memory location belonging to the token will be invalidated. This invalidate operation ensures that the read will return values from the shared memory, which holds the most recent values. Afterwards the lock is released, allowing the producer to modify the token.

Besides putting FIFO buffers in a write-through cacheable region, the sharing problem can also be avoided by mapping only one buffer to a cache line. Therefore FIFO buffers can easily be put in a write-back cacheable region, which saves memory bandwidth and is attractive in combination with an SDRAM.

4.3 Embedding the protocol in POSIX threads

This section presents what is required to embed the software cache coherence protocol in the POSIX threads standard. POSIX threads, also known as Pthreads, provides a standard for an API for creating, manipulating, and managing threads [1]. Pthreads is implemented on our MPSoC because, firstly, it is widely used and accepted for writing general purpose multi-threaded programs, and secondly, the SPLASH-2 benchmark set [17] relies on Pthreads calls, and several SPLASH-2 applications are used in the experimental performance evaluation.

Pthreads intentionally avoids stating a memory consistency model [3]. But in an attempt to give a clear and concise description [3, 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(), ...

The programmer adds synchronization operations to ensure mutual exclusive access to shared variables. The Pthreads function pthread_mutex_lock() resembles an *acquire*. Therefore we have extended the pthread_mutex_lock() with a clean and invalidate operation.

In Pthreads a critical region is exited by calling pthread_mutex_unlock(), which should ensure that all previous issued writes become visible. Therefore we have embedded a clean operation in this Pthreads function.

4.4 Tradeoff in cache coherence operations

In previous sections, we have explained when cache coherence operations are required to be performed, without discussing which cache control operation is most suitable to be used. In our MPSoC, we can choose between operating on the entire cache, an entire way, and lines based on modified virtual address (MVA). The latter might need some clarification; the cache controller is instructed to clean and invalidate several lines when using MVA. In this case, the cache controller first checks whether the cache holds a copy of a certain MVA. If so, the operation is only performed on the line that holds the copy. This is in contrast to operating on the entire cache or an entire way, which both unconditionally perform cache control operations.

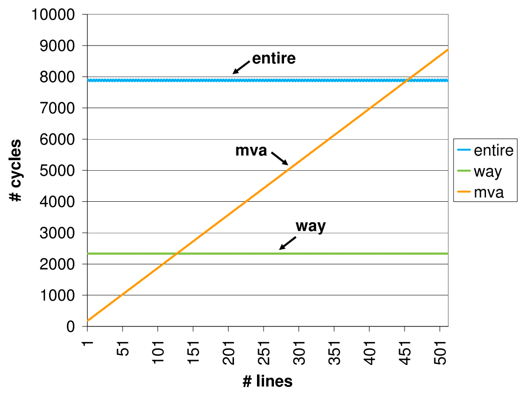


Figure 7: Cycles required for performing invalidate operation

In case shared data is scattered throughout the entire address range, the only option is operating on the entire cache. But, in case shared and private data are separated, there is a tradeoff in deciding which cache control operations to use.

We identify two different types of costs for the cache control operations. The first cost is *execution time*, which denotes the number of cycles that are required to operate on a certain number of cache lines. The second cost is *false invalidations*. A false invalidation is the invalidation of a cache line that is not required to be invalidated, for instance, a cache line holds private data, which never requires to be invalidated.

Figure 7 shows the number of cycles versus a number of cache lines to be invalidated. This figure only displays the execution time, which considers the number of cycles required to invalidate a certain number of cache lines. The time spent in executing the cache control operations was determined on an ARM9 processor.

By using MVA, only a specific number of cache lines is invalidated. Consequently no false invalidations occur. Unfortunately, using MVA has the drawback that it is only efficient if a small number of cache lines need to be invalidated. Cache coherence operations using MVA are implemented as a software loop, which loops through all MVAs in the range that needs to be invalidated. Clearly, if the range to be invalidated is large, this operation will take a high number of cycles. For FIFO communication with small tokens usually only a few cache lines have to be cleaned or invalidated, therefore MVA is suitable for the FIFO software cache coherence protocol.

A second option is to invalidate the entire cache on each acquire. As depicted in Figure 7, invalidating the entire cache requires a fixed number of cycles. However, this operation has the disadvantage that a high number of false invalidations can occur. In some applications the cache may be mainly holding private data, e.g., stack and private heap, and consequently upon an invalidate entire cache a high number of false invalidations occurs.

A third option is to invalidate an entire cache way, which requires a hardware change in the standard ARM926EJ-S cache controller. This operation requires a fixed number of cycles to perform and, even in the worst case, it causes fewer false invalidations than operating on the entire cache. The entire ARM926EJ-S cache consists of 4 cache ways, and this operation only invalidates one of these ways. Although invalidating an entire cache way requires fewer cycles than operating on the entire cache, and it is likely that fewer false invalidations occur in comparison to entire cache, it can still result in lower performance compared to operating on the entire cache. This is mainly caused by forcing all shared data in a specific way, which results in a decrease in cache size and associativity for shared data. This can result in a high number of collisions and consequently a high number of cache misses.

These cache coherence operations are implemented such that the operations are interruptible. Being interruptible is important for our predictability requirement, as we do not allow task switches being postponed due to cache coherence operations. The requirement for interruptible operations forced us to use slightly different cache operations than provided by ARM. The standard ARM9 processor provides operations to invalidate an entire cache, but unfortunately these operations are not interruptible, or may restart once being interrupted. These properties have a negative influence on predictability, consequently we have implemented an interruptible software loop to (clean and) invalidate all cache lines in all cache ways.

Processors can interfere with others in hardware cache coherence protocols, e.g., invalidate requests are issued, whereas our software cache coherence protocol has bounded interference. All cache coherence operations are local to a processor, and do therefore not impact the execution of tasks on other processors. Local cache coherence operations enable worst-case execution time (WCET) analysis of the tasks with classical single processor WCET analysis tools.

5. HARDWARE PLATFORM

This section discusses the hardware platform on top of which our software cache coherence protocol is implemented. This hardware platform is also used in the experimental performance evaluation.

Figure 8 shows the ARM926EJ-S MPSoC which is implemented on a Xilinx Virtex 4 FPGA. In addition to this we have implemented an ARM926EJ-S MPSoC in which the processors were connected through an Æthereal NoC, which has been used in preliminary experiments and will be used in further research. The ARM926EJ-S processors have a four way associative 16Kb cache, and different cache policies, noncacheable, write-through cacheable, and write-back cacheable, can be applied to different regions of the memory.

It is important to understand several specific characteristics of the MPSoC, which can have an impact on the experiments.

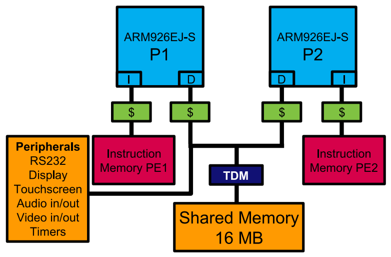


Figure 8: Multiprocessor system-on-chip architecture

Access to the shared memory is controlled by a time-division multiplexing (TDM) arbiter, which provides equal service to both processors. If a processor does not consume its slice, the slice is wasted. Because each processor is granted a certain service, synchronization by spinning on synchronization variables in shared memory does not influence execution of other processors.

The latency to shared memory is only 4 cycles for a word. The number of cycles to fetch 8 single words is equal to fetching one cache line of 8 words. This means that a cache line is not fetched in a pipelined fashion. The memory controller waits 2 cycles for each word, before it is transferred to the processor.

Memory accesses are not reordered in the memory hierarchy, not even in the Æthereal NoC. Furthermore, the memory controller handles memory requests in order.

Software is executed in tasks that are managed by a small preemptive operating system kernel which uses a TDM scheduler. Each task is given a slice of a certain length, and consequently a particular minimum service is guaranteed to each task. The next task that is scheduled is determined following a round-robin scheme.

We have added a *yield* signal to the kernel in order to minimize the impact of spinning idly on locks. This *yield* signal is initiated after an unsuccessful acquire of a lock and forces a task switch. Consequently, a task blocking on a lock does not consume its slice spinning idly, but instead it initiates a task switch.

6. EXPERIMENTS

This section discusses the experiments that have been executed on the MPSoC described in Section 5.

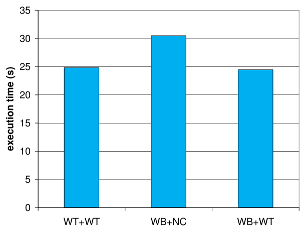
Several applications from the SPLASH-2 benchmark set [17] are used in the experimental performance evaluation. These applications are executed with the standard problem instances, which are given in Table 1. This table also gives the number of locks and the execution time of the application when executed on one processor without cache coherence operations.

The SPLASH-2 applications are executed on the MPSoC with a shared address range of 4 MB. Using MVA to perform the cache operations requires to loop through the entire shared address range, which equals 131,072 MVAs. The SPLASH-2 applications already distinguish between memory allocation for shared and for private data, which enables the separation of private and shared data.

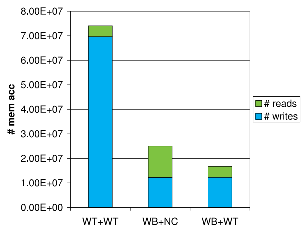
We have stated that separating private and shared data is an optimization. In the first experiment we compare three instances of FFT, which is executed on one pro-

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 1: SPLASH-2 applications



(a) Execution times FFT



(b) Number of memory accesses FFT

Figure 9: Impact of separation shared and private data

cessor without cache coherence operations. The instances are *WT+WT*, which means both private and shared data write-through cacheable, *WB+NC*, which means private date write-back cacheable and shared data noncacheable (as proposed in [11]), and our proposal, *WB+WT*, where private data is write-back cacheable and shared data is write-through cacheable. Figure 9(a) shows the execution time for the three instances, and Figure 9(b) shows the total number of memory accesses for the three instances.

Leaving shared data noncacheable (*WB+NC*) results in the worst execution time compared to the other two instances. This is most likely a result from a high number of read misses in the shared address range. The other two instances *WT+WT* and *WB+WT* do not differ much in terms of execution time.

Putting private data in a write-through cache results in a high number of memory accesses. These accesses are mainly writes, which are presumably writes to the stack. In our proposal *WB+WT* we observe a significant lower number of memory accesses. The difference in memory accesses does not result in a significant difference in execution time, because the write buffer successfully hides write latency.

A second experiment considers the speedup observed when comparing the execution time of the applications executed on a single processor without cache coherence operations with the applications executed on two processors with three different cache coherence operations. In all cases private data is put in a write-back cacheable region, and shared data is put in a write-through cacheable region.

The speedup observed is shown in Figure 10 and the best speedup is between 1.89 and 2.01 for the SPLASH-2 applications. The performance is dependent on the synchronization-to-computation ratio, which affects the number of synchronizations in a time interval.

Cache coherence operations using MVA to loop through the shared address range demand a high number of cycles. Consequently, the performance for MVA is low compared to the other instances. For Raytrace, it was not even possible to retrieve any useful results, which is a direct result from the high number of synchronizations.

For many applications, cache coherence operations on a specific way show a reasonable speedup, comparable to operating on the entire cache. However, this does not hold for the LU non-contiguous application, which apparently suffers from a high number of cache misses. LU non-contiguous performs the same computation as LU contiguous, but the first is not optimized for caches and consequently has a higher cache miss rate than LU contiguous.

Although forcing a way for shared data may degrade the performance in some applications, it can be beneficial. Raytrace with cache coherence operations on a way performs better than operating on the entire cache. This results from the high number of synchronizations that cause a higher number of false invalidations and additionally requires more time to perform the cache coherence operation.

Cache coherence operations on the entire cache usually perform best, but this is highly dependent on the synchronization-to-computation ratio. The reason that operating on the entire cache performs better than on a specific way is a result from the performance degradation by forcing shared data in a way, which can cause a higher number of cache misses.

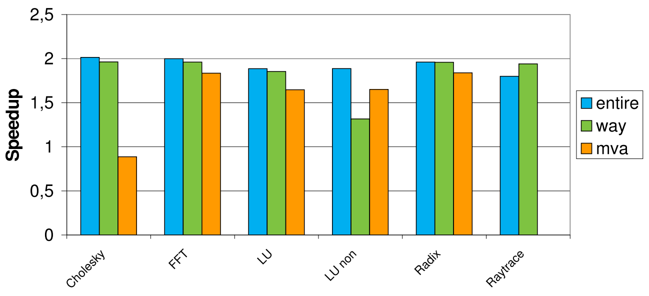


Figure 10: Relative speedup of SPLASH-2 applications

Another important metric for performance and assessing scalability is the memory bandwidth requirement. We have determined this requirement for the SPLASH-2 applications executing with the three different cache coherence operations. In most cases the memory bandwidth requirement was around 2%, consequently the scalability is expected to be good for an MPSoC with up to 10 processors. For larger MPSoCs it can be desirable to increase the memory bandwidth by exploiting a shared level 2 cache and by applying multiple memories in the address space.

More important is the increase in the number of memory accesses as a result of cache coherence operations. We have compared the number of memory accesses for a single processor without cache coherence operations with the total number of memory accesses for the two processor instances with cache coherence operations enabled. The increase in the number of accesses is shown in Figure 11, in which the number of memory accesses for the single processor is set at one.

From Figure 11 it can be concluded that the cache coherence operations do not significantly increase the number of memory accesses. Surprisingly, for some applications the total number of memory accesses in the two processor instance is lower than the number of memory accesses in the single processor instance. This is a positive consequence from the increase in cache size, because the total cache size in the two processor MPSoC doubled.

Clearly, if shared data is forced in a specific cache way the SPLASH-2 applications suffer from cache misses. For all applications, except for Radix, the total number of accesses for way-based cache coherence operations is higher than for the single processor instance. For Radix it is different because it only shares a little amount of data, and it has a low synchronization-to-computation ratio.

An application with a high synchronization-to-computation ratio can benefit from way-based cache coherence operations. An example is Raytrace, which shows that forcing a way only increases the total number of memory access by a small amount, compared to operating on the entire cache. The latter results in a high number of cache misses due to false invalidations.

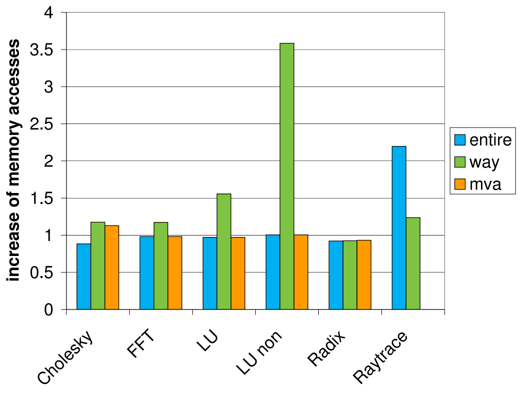


Figure 11: Relative increase in number of memory accesses

In future work it may be interesting to study the impact of processors that exploit instruction level parallelism. As this has an impact on the synchronization-to-computation ratio, which is important for the performance of our software cache coherence protocol.

7. CONCLUDING REMARKS

We presented a tuneable software cache coherence protocol. The protocol has several interesting properties. Firstly, it is applicable to heterogeneous multiprocessor systems-on-chip (MPSoC) with a network-on-chip and with off-the-shelf processors. Secondly, the protocol supports Release Consistency [7], on top of which the standard Pthreads [1] communication library has been implemented. Thirdly, all cache coherence operations are local to a processor and the cache coherence operations are interruptible, which makes the protocol to be suitable for predictable MPSoCs.

The protocol overhead is dependent on application characteristics. The protocol performance is higher if the application has a low synchronization-to-computation ratio. The performance can be improved by providing separate address ranges for private and shared data. Protocol performance can also be improved by putting shared data in a specific cache way, and by restricting interprocessor communication to FIFO communication. The fact that the protocol performance is dependent on application characteristics does not need to be problematic because selection at design-time of applications with specific characteristics is often a valid option for embedded systems.

The software cache coherence protocol has been evaluated in an ARM926EJ-S MPSoC which has been mapped on a Xilinx Virtex 4 FPGA. Several applications from the SPLASH-2 benchmark set [17] have been used in this evaluation.

From experiments we have concluded that the protocol overhead is low for the evaluated SPLASH-2 benchmark applications, because we observe a speedup between 1.89 and 2.01. In addition we have demonstrated that the total number of memory accesses does not increase significantly due to the cache coherence protocol. For applications with a low synchronization-to-computation ratio we observe a lower number of memory accesses, up to a 12% decrease, in the two processor MPSoC compared to a single processor.

8. REFERENCES

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

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

[3] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. PLDI*, pages 261–268, New York, NY, USA, 2005. ACM.

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

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

[6] S. F. Fahmy, B. Ravidran, and E. Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *Proc. DATE*, 2009.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

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

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

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

[11] F. Petrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures. *Proc. DSD*, pages 53–60, 2006.

[12] H. Sandhu, B. Gamsa, and S. Zhou. The shared regions approach to software cache coherence on multiprocessors. *ACM SIGPLAN Notices*, 28(7):229–238, 1993.

[13] T. Suh, D. Blough, and H.-H. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Proc. DATE*, volume 2, pages 1150–1155 Vol.2, Feb. 2004.

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

[15] J.-W. van de Waerdt, S. Vassiliadis, J.-P. van Itegem, and H. van Antwerpen. The TM3270 media-processor data cache. In *Proc. Computer Design: VLSI in Computers and Processors, ICCD*, pages 334–341, Oct. 2005.

[16] J. van den Brand and M. Bekooij. Streaming consistency: a model for efficient MPSoC design. *Proc. DSD*, pages 27–34, 2007.

[17] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Jun 1995.