UvA-DARE (Digital Academic Repository)

The verification of the on-chip COMA cache coherence protocol

Vu, T.D.; Zhang, L.; Jesshope, C.

DOI

[10.1007/978-3-540-79980-1_31](https://doi.org/10.1007/978-3-540-79980-1_31)

Publication date

2008

Document Version

Submitted manuscript

Published in

Algebraic Methodology and Software Technology

[Link to publication](#)

Citation for published version (APA):

Vu, T. D., Zhang, L., & Jesshope, C. (2008). The verification of the on-chip COMA cache coherence protocol. In J. Meseguer, & G. Roşu (Eds.), *Algebraic Methodology and Software Technology: 12th International Conference, AMAST 2008 Urbana, IL, USA, July 28-31, 2008 : proceedings* (pp. 413-429). (Lecture Notes in Computer Science; Vol. 5140). Springer. https://doi.org/10.1007/978-3-540-79980-1_31

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

The verification of the on-chip COMA cache coherence protocol

Thuy Duong Vu¹, Li Zhang², and Chris Jesshope²
{tdvu,zhangli,jesshope}@science.uva.nl

¹ Sectie Software Engineering,

² Computer Systems Architecture Group,
University of Amsterdam,
The Netherlands

Abstract. This paper gives a correctness proof for the on-chip COMA cache coherence protocol that supports the Microgrid of microthreaded architecture, a multi-core architecture capable of integrating hundreds to hundreds of thousands of processors on single silicon chip. We use the Abstract State Machine (ASM) as a theoretical framework for the specification of the on-chip COMA cache coherence protocol. We show that the protocol obeys the Location Consistency model proposed by Gao and Sakar.

Keywords: on-chip COMA cache coherence protocol, verification, location consistency, Abstract State Machine

1 Introduction

A number of computer system architecture and implementation issues as long wire delay, heat dissipation, memory synchronization, etc., have driven the computer architecture to an inevitable transition from single-core to multicore processor design. The Microgrid of microthreaded [13, 3, 4, 12] architecture is designed to possess thousands of on-chip simple processing cores, while providing the scalable throughput both on and off chip. The microthreaded architecture could perform explicit context switch during long latency operations as memory accesses without wasting the processor time.

The shift from off-chip to on-chip multiprocessing allows the cache coherence to operate at a higher clock rate. In addition, the capability of tolerating long memory access latency in the microthreaded architecture helps us revive a paradigm used in earlier parallel computers, such as the Kendal Square KSR1 [5]. We introduce a Cache Only Memory Architecture (COMA) [8] for the on-chip cache system. In COMA, all the memory modules can be considered as large caches, called Attraction Memory (AM). Data is stored by cacheline but the line has no fixed location to find. Similar to COMA, in on-chip COMA, a certain piece of data can be replicated and migrated dynamically between caches. A main difference between the on-chip COMA and traditional COMA is that the traditional COMA system will hold all data in the system without a backing store, while the on-chip COMA has a backing store for data off chip, where an

interface is provided for storing incoming data. The readers are referred to [16] for more detail.

Although the on-chip COMA has some similar property and structure as the traditional COMA, the underlying consistency models and supporting cache coherence protocol are largely different. To balance the programming complexity and execution efficiency, a number of memory consistency models [11] had been proposed before. The most commonly used memory consistency model is the *sequential consistency* (SC) model given by Lamport [15]. In this model, memory operations performed by the processors are *serialized*. Since SC requires all the processors to observe the write requests in some unique order, thus the atomic broadcast communication is generally required for implementing SC, which may severely impair the cache throughput and execution efficiency. Furthermore, the assumption of the universal order poses fundamental obstacles to defining a scalable and efficient view of the memory consistency in computer system. A number of more relaxed consistency models of the SC model such as *release consistency*, *lazy release consistency*, *entry consistency* and *dag consistency* have been proposed in [8, 14, 1, 2]. *Location Consistency* (LC) proposed by Gao and Sakar in [6, 7], is considered the weakest memory model to date. In the LC model, memory operations performed by processors need not be seen in the same order by all processors, and therefore, there can be many multiple legal values for a memory location at the same time. The on-chip COMA cache coherence protocol is designed to obey this consistency model. In its cache system, multiple legal values of a memory location are stored in different caches. This reduces the consistency-related traffic in the network of the cache coherence protocol significantly, since a read operation can read a legal value from a local cache or from the main memory (in the case that there are no legal values available in the caches).

In this paper, we give a correctness proof for the design of the on-chip COMA cache coherence protocol. We show that our protocol does not rely on the memory coherence assumption, and therefore, it does not satisfy the SC and SC-derived models. However, it obeys the LC model of Gao and Sakar. Indeed, our protocol is strictly stronger than the LC model. We will use the *Abstract State Machine* (ASM) [9, 10] as a theoretical framework for the specification and verification of our protocol.

2 Location consistency

In this section, we follow Gao and Sakar [6, 7] to define the location consistency with respect to the microthreaded architecture.

2.1 Programming model

Our programming model consists of two memory operations and two synchronization operations whose descriptions are as follows:

Memory read : If thread T_i needs to read a value from memory location L , it performs a $\text{read}(T_i, L)$ operation, which is also represented by the notation $\text{read } L$ in thread T_i 's instruction sequence.

Memory write : If thread T_i needs to write the value v on location L then it must wait for all read operations issued by T_i and its subthreads on location L to be complete and then performs a $\text{write}(T_i, v, L)$ operation, which is also represented by the notation $L := v$ in thread T_i 's instruction sequence.

Thread creation : If thread T_i needs to create a family of threads then it must wait for all write operations issued by T_i and its subthreads to be complete and then performs a $\text{create}(T_i, \mathcal{F})$ operation where $\mathcal{F}$ is a sequence of threads. This operation is represented by the notation $\text{create}(\mathcal{F})$ in T_i 's instruction sequence. We note that every thread T_j of $\mathcal{F}$ is a *subthread* of T_i , and all subthreads of T_j are also subthreads of T_i .

Barrier synchronization : If thread T_i needs to identify the termination of a specified family of threads, it performs a $\text{sync}(T_i, \mathcal{F})$ operation where $\mathcal{F}$ is the specified family. This operation is represented by the notation $\text{sync}(\mathcal{F})$ in thread T_i 's instruction sequence. The subsequent instructions after $\text{sync}(\mathcal{F})$ in thread T_i must wait until all write operations of the threads in $\mathcal{F}$ and their subthreads are complete.

2.2 State update for a memory location

In the LC model, the state of a memory location is a partial ordered set of memory and synchronization operations. Given a memory location L , the state of L is a *partially ordered multiset* (pomset) $\text{state}(L) = (S, \prec)$, where S is a multiset and $\prec$ is a partial order on S . Each element of S is a memory operation or a synchronization involving location L . Two elements in multiset S can have the same value, however, they can be distinguished by the partial order. For two operations $e_1, e_2 \in S$ such that $(e_1, e_2) \in \prec$, we say that e_1 is a *predecessor* of e_2 . Initially, the state of a memory location is the empty set. For an operation e , we denote $\text{thread}(e)$ as the thread involved in operation e , i.e. $\text{thread}(e) = T_i$ where $e \in \{\text{write}(T_i, v, L), \text{read}(T_i, v, L), \text{create}(T_i, \mathcal{F}), \text{sync}(T_i, \mathcal{F})\}$.

The state of a memory location L is updated when a memory operation on the location L or a synchronization operation is performed. This new operation is inserted to the current multiset of the state. The *precedence relation* (the partial order $\prec$) is updated by the following rules:

1. All operations in the multiset from the same thread with the new operation are considered as the predecessors of that new operation.
2. The thread creation operation creating the thread containing the new operation is a predecessor of that new operation.
3. If this new operation is a barrier synchronization operation then all operations issued by the threads involved in the barrier synchronization operation are predecessors of that new operation.

Let L be a memory location with the current state $(S, \prec)$. The state update of L with operation e is defined as follows. $S := S \cup \{e\}$; and moreover:

$$\begin{aligned} \prec := & \text{trans}(\prec \cup \{(e', e) \mid e' \in S \wedge e' \neq e : \text{thread}(e') = \text{thread}(e)\} \\ & \cup \{(e', e) \mid e' \in S : e' = \text{create}(T_i, \mathcal{F}') \wedge \text{thread}(e) \in \mathcal{F}'\} \\ & \cup \{(e', e) \mid e = \text{sync}(T_j, \mathcal{F}) \wedge e' \in S : \text{thread}(e') \in \mathcal{F}\}) \end{aligned}$$

The function **trans** is to maintain the transitive property of the precedence relation $\prec$: $\text{trans}(\prec) = \prec \cup \{(e_1, e_2) \mid \exists e_1, e_2, e' \in S : (e_1, e') \in \prec \wedge (e', e_2) \in \prec\}$.

2.3 State observability for a memory location

The state of a memory location in the LC model can be observed via read operations. Let L be a memory location with $\text{state}(L) = (S, \prec)$, and $r \in S$ a read operation on L . The **most recent predecessor write** with respect to r is a write operation $w \prec r$ such that there is no other write operation $w' \in S$ satisfying $w \prec w' \prec r$. The read operation r reads a **legal** value v if there is a write operation w such that $w = \text{write}(T, v, L)$, and

1. w is the most recent predecessor write with respect to r , or
2. r and w are unordered, i.e. $(w, r) \notin \prec$.

The set $V(r)$ is the set of all legal values returned by r .

We now recall the definition of the Location Consistency from [7] as follows: A multiprocessor system is **location consistent** if, for any execution of a program on the system, the operations of the execution are the same as those for some location consistent execution of the program; and moreover, for any read operation R with target location L of any execution of a program on the system, R always returns a legal value.

3 The on-chip COMA cache coherence protocol

This section briefly introduces the on-chip COMA cache coherence protocol that supports our programming model. Threads are distributed to processors for their execution. One or more threads can be executed on a processor. Each processor may cache values for many memory locations in a cache consisting of a number of cachelines. The value of a memory location is cached in a cacheline. Note that the values of a memory location stored in different caches can be different, since they are not updated at the same time. For simplicity, we assume that a cache is connected to one processor, and a cacheline stores the value of one memory location only.

Thus in our protocol, caches are connected in a directed ring network which has a *directory* to hold the information about all the data available on the ring. The design of the ring networks is based on the properties of the microthreaded architecture. Due to its capability of tolerating long latency operations, the microgrid has less requirement on latency and higher requirement

Fig. 1. The on-chip COMA cache coherence protocol.

on bandwidth. The low cost ring structure, which can easily broadcast data and invalidation requests, meets both the requirement of COMA and the microgrid CMP. Details can be found in [16]. Only the directory has the access to the main memory. Hence, any loading data from or writing back to the main memory must be handled through this node. The on-chip COMA cache coherence protocol is based on MOSI variations in which a cacheline have four main states **MODIFIED**, **OWNER**, **SHARED** and **INVALID**, and three temporary states **READ PENDING**, **READ PENDING I** and **WRITE PENDING** (see Fig. 1) whose descriptions are given below:

- **INVALID**: If a cacheline is in an **INVALID** state then it has no valid data;
- **MODIFIED**: If a cacheline is in a **MODIFIED** state, it has the exclusiveness of the data.
- **OWNER**: If a cacheline is in an **OWNER** state, it has the ownership of the data, and there can be another cacheline in the system that has a valid data;
- **SHARED**: If a cacheline is in a **SHARED** state, it has a valid data but no ownership of the data.
- **READ PENDING**: If a cacheline is in a **READ PENDING** state, it is waiting for a valid data to be loaded;
- **READ PENDING I**: If a cacheline is in a **READ PENDING I** state, it has received an invalidation request while waiting for a valid data to be loaded. When it gets a valid data, its state will become **INVALID**;
- **WRITE PENDING**: If a cacheline is in a **WRITE PENDING** state, it is waiting for the exclusiveness of the data.

A cache can handle two kinds of requests: local requests and network requests. Local requests are memory operations issued by processors, while network requests (or messages) occur during the communication between caches, and have

a higher priority to be considered than local ones. When a cache receives a local request to read or write a data, it will check whether all cache entries are occupied or not. If yes, it has to eject a cacheline for loading or writing a new data. We note that since caches are connected in a directed ring, network requests can only be sent or passed to the next cache on the ring. The types of requests of the on-chip COMA cache coherence protocol are:

- **LR** (Local Read): the type of a read operation issued by a processor;
- **LW** (Local Write): the type of a write operation issued by a processor;
- **RS** (Remote Read to **SHARED** state): issued by a cache to ask for a valid data when it receives a **LR** request but it has no valid data;
- **SR** (to **SHARED** state) Read Reply: issued by a cache when it receives a **RS** request and has a valid data;
- **IV** (InValidation): issued by a cache when it receives a **LW** request. It wants to become the ownership of the data, and therefore, tries to invalidate all other data on the ring network;
- **WB** (Write Back to main memory): issued by a cache to write back a dirty data (whose cacheline is in a **MODIFIED** or **OWNER** state) to the main memory before the ejection of the cacheline;
- **eject**: issued by a cache to notice the directory that one of its cachelines in a **SHARED** state has been ejected.

The next section will describe the on-chip COMA cache coherence protocol in more detail.

4 The specification of the protocol

This section specifies the on-chip COMA cache coherence protocol in the Abstract State Machine (ASM) framework [9, 10]. The protocol is considered as an ASM whose transition rules represent the behavior of the protocol.

4.1 Vocabulary

We assume the existence of a fixed set **Thread** of threads, a fixed set **Processor** of processors, a fixed set **Location** of memory locations, a fixed set **Operation** of operations, a fixed set **Message** of messages, and a fixed set **Data** of data values. The undefined value or attribute of an object is specified as **undef**.

For a thread T , there is an attribute $\text{proc} \in \text{Processor}$ to characterize the processor where T is distributed to. Let $\text{Type} = \{\text{LR}, \text{LW}, \text{CRE}, \text{SYNC}, \text{RS}, \text{SR}, \text{IV}, \text{WB}, \text{eject}\}$. An operation $e \in \text{Operation}$ has four attributes $\text{type} \in \text{Type}$, $\text{thread} \in \text{Thread}$, $\text{val} \in \text{Data}$ and $\text{loc} \in \text{Location}$ to characterize the type, the thread, the data and the memory location involved in the operation. For instance, for a write operation $e = \text{write}(T, v, L)$, $e.\text{type} = \text{LW}$, $e.\text{thread} = T$, $e.\text{val} = v$ and $e.\text{loc} = L$.

By the assumption, for a processor there is only one cache, and vice versa. We can assume that messages are issued by processors as well. Hence, a message $m \in \text{Message}$ has three attributes $\text{type} \in \text{Type}$, $\text{val} \in \text{Data}$ and $\text{loc} \in \text{Location}$.

to characterize the type, the data and the memory location involved in the message. Moreover, it has an attribute **source** $\in \text{Processor}$ to characterize the processor who originally sends out the request. We denote the empty message as **noMess**. For each processor P and for each location $l \in \text{Location}$, the pair (P, l) represents a unique cacheline whose description is given by the following functions:

- **state** $(P, l) \in \{\text{undef}, \text{INVALID}, \text{MODIFIED}, \text{OWNER}, \text{READ PENDING}, \text{READ PENDING I}, \text{WRITE PENDING}\}$ to indicate the current state of the cacheline (P, l) . Initially, **state** $(P, l) = \text{undef}$;
- **cacheValid?** (P, l) to indicate whether the cacheline (P, l) has a valid or not. Initially, **cacheValid?** $(P, l) = \text{false}$;
- **cacheVal** (P, l) to indicate the value stored in the cacheline (P, l) . Initially, **cacheVal** $(P, l) = \text{undef}$;
- **cacheDirty?** (P, l) to indicate whether the cacheline (P, l) holds a dirty data or not. Initially, **cacheDirty?** $(P, l) = \text{false}$;
- **Eject** $(P, l) = \{\text{state}(P, l) := \text{undef}, \text{cacheDirty?}(P, l) := \text{false}, \text{cacheValid?}(P, l) := \text{false}, \text{cacheVal}(P, l) := \text{undef}, \text{state}(P, P.\text{curOp}.\text{loc}) := \text{INVALID}\}$ to evict the cacheline (P, l) from the cache of P ;
- **Invalidate** $(P, l) = \{\text{state}(P, l) := \text{INVALID}, \text{cacheDirty?}(P, l) := \text{false}, \text{cacheValid?}(P, l) := \text{false}, \text{cacheVal}(P, l) := \text{undef}\}$ to invalidate the cacheline (P, l) .

A processor $P \in \text{Processor}$ has the following attributes:

- **cacheOccupied?** $\in \{\text{true}, \text{false}\}$ to determine whether all cache entries of P are occupied or not, monitored by the execution environment;
- **id** $\in \mathbb{N}$ to indicate the index of P ;
- **neighbor** $\in \text{Processor} \cup \{\text{dir}\}$ to indicate the next node of P on the ring network;
- **ejectee** $\in \text{Location}$ to indicate the location to be ejected when all cache entries of P are occupied, monitored by the execution environment satisfying the condition that **state** $(P, P.\text{ejectee}) \neq \text{undef}$;
- **Return** $\in \text{Data} \times \text{Location}$ to return a read value asked by a LR request;
- **curMess** $\in \text{Message}$ to indicate the current network request of P ;
- **curOp** $\in \text{Operation}$ to indicate the current operation performed by P , monitored by the execution environment;
- **nextOp** $\in \text{Operation}$ to indicate the next operation performed by P , monitored by the execution environment.

The notation **dir** denotes the directory which has the following attributes:

- **MMVal** $: \text{Location} \rightarrow \text{Data}$ to determine the value of a location stored in the main memory;
- **neighbor** $\in \text{Processor}$ to indicate the next node of **dir** on the ring network;
- **curMess** $\in \text{Message}$ to indicate the current network request of **dir**;


```

if curMess = noMess & curOp.type = LR then
  if state(self, curOp.loc) = undef then
    if cacheOccupied? then
      if cacheDirty?(self, ejectee) and neighbor.curMess = noMess then
        SendMess(self, WB, cacheVal(self, ejectee), ejectee)
        Eject(self, ejectee)
      if state(self, ejectee) = SHARED and neighbor.curMess = noMess then
        SendMess(self, eject, cacheVal(self, ejectee), ejectee)
        Eject(self, ejectee)
      if state(self, ejectee) = INVALID then
        Eject(self, ejectee)
    else state(self, curOp.loc) := INVALID
  else
    if cacheValid?(self, curOp.loc) then Return(cacheVal(self, curOp.loc), curOp.loc)
    if state(self, curOp.loc) = INVALID then
      if neighbor.curMess = noMess then
        SendMess(self, RS, -, curOp.loc)
        state(self, curOp.loc) := READ PENDING
        curOp := nextOp
      else curOp := nextOp

```

Table 1. Transition rule responding LR requests.

- $\text{cacheCounter} : \text{Location} \rightarrow \mathbb{N}$ to determine the numbers of valid caches for a memory location on the ring network. Initially, for all $l \in \text{Location}$, $\text{dir.cacheCounter}(l) = 0$. This counter is updated as follows. When the directory receives an IV request, meaning that someone wants to become the ownership of the data, the counter is set to 1. When the directory receives a RS (or SR) request from (or for) a processor whose cacheline is not in READ PENDING I state, meaning that someone wants to have a valid data, the counter is increased by 1. When the directory receives a WB (or eject) request, meaning that someone who has a valid data has been ejected, the counter is decreased by 1.

There are also two auxiliary functions needed for the specification of the protocol:

- $\text{SendMess}(P, \text{messType}, \text{val}, l) = \{\text{if messType} = \text{SR then}$
 $\text{mess.source} := P.\text{curMess.source}$ **else** $\text{mess.source} = \text{proc},$
 $\text{mess.type} := \text{messType}, \text{mess.val} := \text{val}, \text{mess.loc} := l,$
 $P.\text{neighbor.curMess} := \text{mess}, P.\text{curMess} := \text{noMess}\}$
 to send a message from node P to its next node ($P.\text{neighbor}$) on the ring;
- $\text{PassMess}(P) = \{P.\text{neighbor.curMess} = P.\text{curMess}, P.\text{curMess} := \text{noMess}\}$
 to pass the current message of node P to its next node ($P.\text{neighbor}$).

4.2 Transition rules

The behavior of the on-chip COMA protocol is represented as an ASM *module* whose *transition rules* are given in Table 1, Table 2, Table 3, Table 4, Table 5, Table 6 and Table 8. We will sometimes shorten macros such as **self.curMess** | **cacheOccupied?** | **ejectee** | **neighbor** | **curOp** | **nextOp** | **id** | **MMVal** | **cacheCounter** by **curMess**, **cacheOccupied?**, **ejectee**, **neighbor**, **curOp**, **nextOp**, **id**, **MMVal** and **cacheCounter**. With reference to Table 1, we first explain how a processor P reacts when it receives a LR (Local Read) request. As mentioned earlier, network

```

if curMess = noMess & curOp.type = LW then
  if state(self, curOp.loc) = undef then
    if cacheOccupied? then
      if cacheDirty?(self, ejectee) and neighbor.curMess = noMess then
        SendMess(self, WB, cacheVal(self, ejectee), ejectee)
        Eject(self, ejectee)
      if state(self, ejectee) = SHARED and neighbor.curMess = noMess then
        SendMess(self, eject, cacheVal(self, ejectee), ejectee)
        Eject(self, ejectee)
      if state(self, ejectee) = INVALID then
        Eject(self, ejectee)
      else state(self, curOp.loc) := INVALID
    else
      cacheVal(self, curOp.loc) := opVal
      cacheValid?(self, curOp.loc) := true
      if state(self, curOp.loc) = WRITE PENDING then curOp := nextOp
    else if neighbor.curMess = noMess then
      SendMess(self, IV, -, curOp.loc)
      state(self, curOp.loc) := WRITE PENDING
      curOp := nextOp

```

Table 2. Transition rule responding LW requests.

requests have higher priority to be considered than the local ones. Thus, this local read request is only considered in the case that there is no network request available for P , i.e. $P.curMess = noMess$. If there is no cache entry set up for the memory location involved in the request yet ($state(P, P.curOp.loc) = undef$), then P first checks whether all cache entries are occupied or not. If yes ($P.cacheOccupied? = true$), P has to eject a cacheline determined by the execution environment ($P.ejectee$ with $state(P, P.ejectee) \neq undef$). If the ejectee has a dirty data then this data is written back to the main memory by sending a WB (Write Back) message to the directory. If the ejectee has a valid (but not dirty) data then P also sends out a `eject` message to notice the directory. If the ejectee is in a pending state then the removal is also pending. P then removes the ejectee, and sets up another cacheline for the location concerned ($state(P, P.curOp.loc) = INVALID$). If this cacheline has a valid data, then P just sends back the value stored in the cacheline. The read operation is considered *complete*. If it has no valid data and its state is `INVALID`, then P sends a RS request to other processors to ask for a valid data.

In Table 2, we now explain how a processor P reacts when it receives a LW (Local Write) request. Similar to the previous case, this request is only considered by the processor and when no network request is available. Moreover, P also has to set up a cacheline for writing the new data as in Table 1 in the case that there is no place for that yet. If there is already an available cacheline, P just overwrites the data. If the cacheline is not waiting for the exclusiveness of the data (i.e. it is not in a `WRITE PENDING` state), P then sends out a IV request to other processors to ask for the exclusiveness of the data. The state of the cacheline becomes `WRITE PENDING`. We impose the following condition on all write operations to avoid *useless* reads for a thread:

```

if curMess.type = RS then
  if curMess.source = self and cacheValid?(self, curMess.loc) then
    Return(cacheVal(self, curMess.loc), curMess.loc)
    if state(self, curMess.loc) = READ PENDING then state(self, curMess.loc) := SHARED
    if state(self, curMess.loc) = READ PENDING I then Invalidate(self, curMess.loc)
    curMess := noMess
  else if neighbor.curMess = noMess then
    if self = dir then
      if cacheCounter(curMess.loc) = 0 then
        SendMess(self, SR, MMVal(curMess.loc), curMess.loc)
      else PassMess(curMess)
      if state(curMess.source, curMess.loc) = READ PENDING then
        cacheCounter(curMess.loc) := cacheCounter(curMess.loc) + 1
    else if cacheValid?(self, curMess.loc) then
      SendMess(self, SR, curMess.val, curMess.loc)
      if state(self, curMess.loc) = MODIFIED then state(self, curMess.loc) := OWNER
    else PassMess(curMess)

```

Table 3. Transition rule responding RS requests.

```

if curMess.type = SR then
  if curMess.source = self then Return(curMess.val, curMess.loc)
  if state(self, curMess.loc) = READ PENDING then
    state(self, curMess.loc) := SHARED
    cacheVal(self, curMess.loc) := curMess.val
    cacheValid?(self, curMess.loc) := true
  if state(self, curMess.loc) = READ PENDING I then Invalidate(self, curMess.loc)
  curMess := noMess
  else if neighbor.curMess = noMess then
    if self = dir and state(curMess.source, curMess.loc) = READ PENDING then
      cacheCounter(curMess.loc) := cacheCounter(curMess.loc) + 1
    else if state(self, curMess.loc) ∈ {READ PENDING, READ PENDING I} then
      cacheVal(self, curMess.loc) := curMess.val
      cacheValid?(self, curMess.loc) := true
    PassMess(curMess)

```

Table 4. Transition rule responding SR requests.

Condition 1 For every processor P , if the current operation of P is a write operation ($P.\text{curOp} = w$) then all the read operations performed by $w.\text{thread}$ and its subthreads on the same location ($w.\text{loc}$) must be complete.

Table 3 presents the transition rules for a node P on the ring network when it receives a RS (Remote Read to Shared state) request. If P originally sent out the request and has a valid data then P just sends back the value stored in its cache. If the cacheline concerned is currently in a READ PENDING state then its state becomes SHARED; otherwise it will be invalidated. All the read operations performed by P on the same location are considered *complete*. If P is the directory and there are no caches on the ring network having a valid data ($P.\text{cacheCounter}(\text{curMess.loc}) = 0$) then it sends out the value stored in the main memory together with the read reply SR. The counter of valid caches for the location involved in the message is increased by 1 if the cacheline who requested the data is in a READ PENDING state. If P is a processor who has a valid data then it sends out the data together with the reply SR. The state of the cacheline concerned is set to OWNER if it is MODIFIED in the current step. In the remaining cases, it just passes the message to the next node on the ring.

```

if curMess.type = IV then
  if curMess.source = self then
    if state(self, curMess.loc) = WRITE PENDING then
      state(self, curMess.loc) := MODIFIED
      cacheDirty?(self, curMess.loc) := true
      curMess := noMess
    else if neighbor.curMess = noMess then
      if state(self, curMess.loc) = WRITE PENDING and id < curMess.source.id then
        Invalidate(self, curMess.loc)
      else if state(self, curMess.loc) = READ PENDING then
        state(self, curMess.loc) = READ PENDING I
      else if state(self, curMess.loc) ∉ {undef, READ PENDING I} then
        Invalidate(self, curMess.loc)
      else if self = dir then cacheCounter(curMess.loc) := 1
      PassMess(curMess)

```

Table 5. Transition rule responding IV requests.

```

if curMess.type = WB then
  if self = dir then
    MMVal(curMess.loc) := curMess.val
    cacheCounter(curMess.loc) := cacheCounter(curMess.loc) - 1
    curMess := noMess
  else if neighbor.curMess = noMess then
    if state(self, curMess.loc) ∈ {READ PENDING, READ PENDING I} then
      cacheVal(self, curMess.loc) := curMess.val
      cacheValid?(self, curMess.loc) := true
    PassMess(curMess)

```

Table 6. Transition rule responding WB requests.

In Table 4, transition rules reacting SR (to Shared state Read Reply) requests for a node P are given. If P is waiting for this reply then it just sends back the data involved in the reply. The state of the cacheline concerned becomes **SHARED** if it was **READ PENDING**, and becomes **INVALID** otherwise. All the read operations performed by P on the same location are considered *complete*. If P is waiting for a valid data ($\text{state}(\text{self}, \text{curMess.loc}) \in \{\text{READ PENDING}, \text{READ PENDING I}\}$), then P just overwrites the data. If P is the directory and the cacheline requested this reply is in a **READ PENDING** state, then the counter of valid caches for the memory location involved in the message is increased by 1. P then passes the message to the next node on the ring network.

Table 5 presents transition rules reacting IV (Invalidation) requests for a node P . If P originally sent out the request and is waiting for the exclusiveness of the data then it has the exclusiveness of the data. The state of the cacheline concerned becomes **MODIFIED**. All the write operations performed by P on the same location are considered *complete*. If P did not sent out the request but it is also waiting for the exclusiveness of the data then a *racing situation* occurs. In this case, we compare the indexes of P and the processor who originally sent out the request. If P has a smaller index then it has to give up the exclusiveness of the data. The state of the cacheline concerned becomes **INVALID**. If P is waiting for a valid data, then the state of the cacheline concerned becomes **READ PENDING I**. In the remaining defined states, the state is reset to **INVALID**. Finally, if P is the directory then the counter of valid caches for the location involved in the message is reset to 1.

```

if curMess.type = eject then
  if self = dir then
    cacheCounter(curMess.loc) := cacheCounter(curMess.loc) - 1
    curMess := noMess
  else if neighbor.curMess = noMess then
    if state(self, curMess.loc) ∈ {READ PENDING, READ PENDING I} then
      cacheVal(self, curMess.loc) := curMess.val
      cacheValid?(self, curMess.loc) := true
    PassMess(curMess)

```

Table 7. Transition rule responding Eject messages.

```

if curMess = noMess & curOp = create( $\mathcal{F}$ ) then curOp := nextOp
if curMess = noMess & curOp = sync( $\mathcal{F}$ ) then curOp := nextOp

```

Table 8. Transition rules responding create and sync operations.

With reference to Table 6, we explain how a node P reacts when it receives a WB (Write Back to the main memory) request. If P is the directory, then P updates the memory value with the data involved in the message. The counter of valid caches for the location concerned is decreased by 1. If P is waiting for a valid data, then P also updates the cached value with the data value involved in the WB message. The state of the cacheline concerned remains the same. P then passes the message to the next node on the ring. Similarly, Table 7 presents transition rules for a node P when it receives a `eject` message. The only difference between the two tables is that in Table 7, P does not update the value stored in the main memory.

In Table 8, we provide transition rules for a processor P in the case that its current operation is a thread creation or a synchronization operation. We impose the following conditions to ensure that these operations are treated in the right order.

- Condition 2**
1. *For every processor P , if the current operation of P is a creation operation ($P.\text{curOp} = c$) then all the write operations performed by the creating thread $c.\text{thread}$ and its subthreads must be complete.*
 2. *For every processor P , if the current operation of P is a synchronization operation ($P.\text{curOp} = \text{sync}(\mathcal{F})$) then all the write operations performed by the threads in $\mathcal{F}$ and their subthreads must be complete.*

5 The on-chip COMA cache coherence protocol obeys LC

In this section, we show that the on-chip COMA cache coherence protocol obeys the LC model, i.e. a read operation r always returns a value belonging to the set $V(r)$ defined as in Section 2.

By the ASM Lipari Guide [10], we lose no generality by proving correctness of an arbitrary linearization of a run of a distributed ASM. Hence, let ρ be a linearization of an arbitrary distributed run of the on-chip COMA cache coherence protocol, and let $\prec_m$ be the linear order on the moves of ρ . We adapt the definition of state update for a memory location in Section 2.2 as follows. When

the current operation of a processor P concerning with a memory location L is updated by a move $P.\text{curOp} := P.\text{nextOp}$ in ρ , the state of memory location L is updated as $S := S \cup \{P.\text{curOp}\}$. We say that:

1. A processor P performs a read r at a move P_r if $P.\text{curOp} = r$;
2. A processor P completes a read r and reads value v for r at a move C_r if $\text{cacheValid?}(P, r.\text{loc})$ at P_r , $C_r = P_r$ and $v = \text{cacheVal}(P, r.\text{loc})$, or C_r is the first move after P_r at which $P.\text{curMess.source} = P$, $P.\text{curMess.loc} = r.\text{loc}$, $P.\text{curMess.type} = \text{RS}$ and $v = \text{cacheVal}(P, r.\text{loc})$; or $P.\text{curMess.type} = \text{SR}$ and $v = P.\text{curMess.val}$;
3. A processor P performs a write w at a move P_w if $P.\text{curOp} = w$;
4. A processor P completes any write w at a move C_w if it is the first move after P_w at which $P.\text{curMess.source} = P$, $P.\text{curMess.loc} = w.\text{loc}$ and $P.\text{curMess.type} = \text{IV}$;
5. A processor P performs a thread creation c at a move P_c if $P.\text{curOp} = c$;
6. A processor P performs a synchronization s at a move P_s if $P.\text{curOp} = s$.

Lemma 1. *Let w be a write operation, and o a memory operation on a location L satisfying that $w \prec o$, and w and o are not issued by the same thread. Then w must be complete before the execution of o , i.e. $C_w \prec_m P_o$.*

Proof. Since $w \prec o$, $w.\text{thread} \neq o.\text{thread}$ and by the definition of state update for a memory location in Section 2.2, we consider two possibilities:

1. There is an operation $c = \text{create}(T, \mathcal{F})$ such that $w.\text{thread} = T$ and $w \prec c \prec o$. Let P_w , P_c and P_o be the moves at which w , c and o are performed in ρ , respectively. Then $P_w \prec_m P_c \prec_m P_o$. It follows from Transition rule in Table 8 and Condition 2 that w must be complete before the next operation of c is performed. Hence, w must be complete before the move O .
2. There is an operation $s = \text{sync}(T, \mathcal{F})$ such that $w.\text{thread} \in \mathcal{F}$ and $w \prec s \prec o$. Again, it follows from Transition rule in Table 8 and Condition 2 that w must be complete before the execution of o .

Lemma 2. *In ρ , let C_r be a move at which a processor P completes a read r and reads value v for r . Then v is defined.*

Proof. We prove only for the case that $P.\text{curMess.type} = \text{RS}$ at C_r . The other cases are obvious. In this case, P had no valid data at P_r , and then it sent out a **RS** request for a valid data (by Transition rules in Table 1). Later, this request comes back to P at move C_r . We show that $\text{cacheValid?}(P, r.\text{loc})$ at C_r . It follows from Transition rules in Table 3 that there was no processor from P to **dir**, and there was at least one processor from **dir** to P having a valid data when the **RS** request arrived at **dir**. Since the **RS** request comes back to P , the processors having a valid data had been ejected when the **RS** request arrived to them. By Transition rules in Table 1 and Table 2, they had to send a **WB** or a **eject** message to notice the directory before their ejection. These messages can update the data stored in the cacheline $(P, r.\text{loc})$, according to Transition rules in Table 6 and Table 7. Hence there exists a move

M such that $P_r \prec_m M \prec_m C_r$ and $\text{cacheValid?}(P, r.\text{loc})$ at M . It follows from Transition rules in Table 2, Table 3 and Table 5 that $\text{state}(P, r.\text{loc}) \in \{\text{READ PENDING}, \text{READ PENDING I}, \text{WRITE PENDING}\}$ between moves P_r and C_r . This implies that $\text{cacheValid?}(P, r.\text{loc})$ at C_r .

Finally, we prove our main theorem as follows.

Theorem 1. *In ρ , let C_r be a move at which a processor P reads value v for a read operation r . Then v is a legal value returned by r .*

Proof. Let w be the write operation that writes the value v ($v = w.\text{val}$). There are two cases:

1. w and r are unordered, i.e. $(w, r) \notin \prec$. Then $v \in V(r)$.
2. w and r are ordered. It follows from Condition 1 that $w \prec r$. We show that w is the most recent predecessor write with respect to r , i.e. there does not exist a write operation w' such that $w \prec w' \prec r$. We prove by contradiction. Assume that there is a write operation w' satisfying $w \prec w' \prec r$. If w' and r are performed by the same thread, and therefore, by the same processor then $C_{w'} \prec_m P_r$, otherwise v would be written by w' , not w . If w' and r are performed by different threads then by Lemma 1, we also get that $C_{w'} \prec_m P_r$. With the transition rules in Table 1, Table 3 and Table 4, we consider the following subcases:
 - (a) P reads the value v from a cache. Since after the completion of w' , there is still processor P that has a valid data written by w , $C_{w'} \prec_m C_w$. This is a contradiction, since w must be complete before the move $P_{w'}$ in the case that w and w' are performed by different threads (by Lemma 1), or w must be complete before or at the same time as the completion of w' in the case that w and w' are performed by the same thread (by Transition rules in Table 5).
 - (b) P reads the value v from the main memory, i.e. w was written back to the main memory. Since $C_{w'} \prec P_r$, w' was also written back to the main memory when the RS request for the value v from P arrived at the directory dir . Otherwise, $\text{dir.cacheCounter}(r.\text{loc}) \geq 1$ (by Transition rules in Table 5), and therefore, P would not read the value stored in the main memory. Since $w \prec w'$, v is written by w' . This contradicts the assumption that v is written by w .

Hence, there does not exist a write operation w' satisfying $w \prec w' \prec r$, or w is the most recent predecessor write with respect to r . Thus, $v \in V(r)$.

6 Relation with the standard consistency models

6.1 Our consistency model is weaker than SC model

Sequential consistency requires all memory operations to be executed in some sequential order, and the operations with in a process to be executed in program

order. In the context of microthreaded architecture, adapting location consistency model, memory accesses to different locations do not conform to any order. To illustrate the model's discontentment of SC, we recall the standard example from [6] as follows.

Example 1. Let threads T_1 and T_2 be distributed to two different processors. T_1 first writes 1 to the shared variable x and then reads the value of the shared variable y . Symmetrically, T_2 writes 1 to the shared variable y and then read the value of x . Note that initially, $x = y = 0$.

```
X:=0; Y:=0; /* initial values */
T1:                                     T2:
w1: x := 1;                             w2: y := 1;
r1: read Y;                             r2: read x;
```

Under the SC and SC-derived models, the operations from T_1 and T_2 are seen in the same order by both processors. Hence, the case that both read operations r_1 and r_2 return 0 is prohibited by the SC and SC-derived models. However, this can happen according to the on-chip COMA cache coherence protocol. Here, r_1 and r_2 can just return the initialized values of x and y which are 0.

6.2 Our model is stronger than the strong LC model

After proving our on-chip COMA system complies with LC model, in this section, we show that our system is not strongly LC consistent [6]. Here, we consider the following example.

Example 2. Thread T_0 creates two separate thread families consisting of T_1 , T_2 and T_3 , T_4 accordingly. Threads T_1 and T_2 perform write operations on location L , and T_3 and T_4 perform read operations of L . We assume threads T_1 , T_2 , T_3 and T_4 are running on different processors.

```
T0:                                     T1:      T2:      T3:      T4:
create(T1,T2);   L:=1;   L:=2;   r0: read L;   r1: read L;
sync(T1,T2);
create(T3,T4);
sync(T3,T4);
```

Under the strong location consistency definition, r_0 and r_1 can return different values from any of the two indeterministic writes in T_1 and T_2 . However, in our system, after the synchronization on T_1 and T_2 , only one of the two values written by T_1 and T_2 will be alive. Thus, the r_0 and r_1 cannot observe different values left by the two write operations. Hence, we conclude that our memory system implementing location consistency is not strongly location consistent.

7 Concluding remarks

In this paper, the on-chip COMA cache coherence protocol has been formally specified in the ASM framework. We gave a correctness proof for the coherence protocol. Furthermore, we showed that our memory system is weaker than

the SC and SC-derived models. It complies with location consistency but it is not strongly location consistent. Our work is a part of a project investigating microthreading in a collaboration between the Computer Systems Architecture group and Sectie Software Engineering at the University of Amsterdam³.

References

1. B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proceedings of the IEEE COMPCON*, 1993.
2. R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson, and K.H. Randall. An analysis of DAG-consistent distributed shared-memory algorithms. In *Proceedings of the 18th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, 1996.
3. A. Bolychevsky, C.R. Jesshope, and V. Muchnick. Dynamic sheduling in rics architectures. In *IEE Proceedings Computers and Digital Techniques*, volume 143(5), pages 309–317, 1996.
4. K. Bousias, N.M. Hasasneh, and C.R. Jesshope. Instruction-level parallelism through Microthreading—a scalable Approach to chip multiprocessors. *The Computer Journal*, 49 (2):211–233, 2006.
5. Corporation. K.s.r.: Ksr1 technical summary. Technical report, 1992.
6. G.R. Gao and V. Sarkar. Location consistency: Stepping beyond memory coherence barrier. In *Int’l Conf. Parallel Processing*, volume 2, pages 73–76, 1995.
7. G.R. Gao and V. Sarkar. Location consistency - A new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8):798–813, 2000.
8. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. 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.
9. Y. Gurevich. Elvolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
10. Y. Gurevich. May 1997 draft of the asm guide. Technical Report CSE-TR-336-97, EECS Department, University of Michigan, 1997.
11. J.H. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. 2006.
12. C.R. Jesshope. Microthreading a model for distributed instruction-level concurrency. *Parallel Processing Letters*, 16 (2):209–228, 2006.
13. C.R. Jesshope and B. Luo. Micro-threading: A new approach to future risc. In *ACAC 2000*, pages 31–41. IEEE Computer Society Press, 2000.
14. P. Keleher, A.L. Cox, and W. Zwaenpoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 13–21, 1992.
15. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
16. L. Zhang and C.R. Jesshope. On-chip coma cache-coherence protocol for microgrids of microthreaded cores. In *Euro-Par 2007 Workshops: Parallel Processing*, volume Volume 4854/2008, pages 38–48. Springer Berlin / Heidelberg, 2008.

³ The work presented in this paper is supported by NWO (Netherlands Organisation for Scientific Research) in the "Foundations for Massively Parallel on-chip Architectures using Microthreading" project.