Defining Uniform and Hybrid Memory Consistency Models on a Unified Framework

Alba Cristina Magalhães Alves de Melo
Department of Computer Science - University of Brasilia (CIC/UnB) - Brazil
albamm@cic.unb.br

Abstract

The behavior of Distributed Shared Memory Systems is dictated by the Memory Consistency Model. Several Memory Consistency Models have been proposed in the literature and they fit basically in two categories: uniform and hybrid models. To provide a better understanding on the semantics of the memory models, researchers have proposed formalisms to define them. Unfortunately, most of the work has been done in the definition of uniform memory models. In this paper, we propose a general, unified and formal framework where uniform and hybrid memory consistency models can be defined. To prove the generality of the framework, we use it to define the following Memory Models: Atomic Consistency, Sequential Consistency, Causal Consistency, PRAM Consistency, Slow Memory, Weak Ordering, Release Consistency, Entry Consistency and Scope Consistency.

1. Introduction

In order to make shared memory programming possible in complex parallel architectures, we must create a shared memory abstraction that parallel processes can access. This abstraction is called Distributed Shared Memory (DSM). The first DSM systems tried to give parallel programmers the same guarantees they had when programming uniprocessors. It has been observed that providing such a strong memory model creates a huge coherence overhead, slowing down the parallel application and bringing frequently the system into a thrashing state [28] [27]. To alleviate this problem, researchers have proposed to relax some consistency conditions, thus creating new shared memory behaviors that are different from the traditional uniprocessor one. Many new memory consistency models[1] have been proposed in the literature but, by now, none of them has achieved the goal of providing both performance and ease of programming for a large set of parallel/distributed applications.

There are basically two categories of memory consistency models: uniform and hybrid models. A uniform model considers only read and write memory operations to define consistency conditions whereas a hybrid model considers also synchronization operations in this definition.

Memory consistency models, strong or relaxed ones, have not been originally formally defined. In most cases, the semantics of memory behavior have to be induced from the protocol that implements the memory model. The lack of a unique framework where memory models can be formally defined makes it difficult to compare and understand the memory model semantics. This fact was also observed by other researchers and some work was indeed done in the sense of formal memory model definitions. Unfortunately, much of the work has been done to define uniform models formally while little attention has been paid to hybrid models.

In this article, we describe a new formal framework that can be used to define uniform and hybrid memory consistency models. Basically, we identify the characteristics that are inherent to all memory consistency models and the characteristics that are model-specific. A simple and general definition of memory consistency model is proposed.

In order to prove the generality of our framework, we use it to define five uniform and four hybrid memory consistency models. The following uniform memory consistency models are defined: dynamic atomic consistency, sequential consistency, causal consistency, processor consistency and slow memory. The following hybrid memory models are defined: weak ordering, release consistency, entry consistency and scope consistency.

The framework described in this paper was used as a basis for the design of a multiple memory consistency model DSM system that was presented in [6].

The rest of this paper is organized as follows. Section 2 describes our system model and framework. Using this framework, we define some well-known uniform memory

[1] The terms "memory model" and "memory consistency model" are used interchangeably in this paper.

models in section 3. Section 4 presents the definition of some hybrid memory models. Related work in the area of formal memory consistency model definition is presented in section 5. Finally, conclusions and future work are presented in section 6.

2. System Model

To describe memory models in a formal way, we propose a history-based system model that is related to the models described in [2] and [19].

In our model, a *parallel program* is executed by a *system*. A *system* is a finite set of *processors*. Each *processor* executes a *process* that issues a set of *operations* on *the shared global memory* $\mathbb{M}$.

The *shared global memory* $\mathbb{M}$ is an abstract entity composed by all addresses that can be accessed by a program. Each *processor* p_i has its own *local memory* m_i. Each *local memory* m_i caches all memory addresses of $\mathbb{M}$.

A memory operation $o_{pi}(x)v$ is executed by processor p_i on memory address x with the value v. There are two basic types of operations on $\mathbb{M}$: read (*r*) and write (*w*). Also, there is a synchronization type (*sync*). A *sync* operation can be of three subtypes: *acquire, release* or *nsync.*

Table 1 shows the auxiliary functions related to memory operations. These functions will be used in some formal definitions.

Table 1. Functions Related to Memory Operations

1) processor($o_{pi}(x)v$) = p_i
2) address ($o_{pi}(x)v$) = x
3) type ($o_{pi}(x)v$) = o
4) subtype ($o_{pi}(x)v$) = acquire, release, nsync

Each memory operation is first *issued* and then *performed*, i. e., memory operations are non-atomic.

A memory operation $o_{pi}(x)v$ is *issued* when processor p_i executes the instruction $o(x)v$.

A read operation $r_{pi}(x)v$ is *performed* when a write operation on the same location x cannot modify the value v returned to p_i. Read operations of p_i are always done on the local memory m_i.

A write operation $w_{pi}(x)v$ is in fact a set of memory operations $\mathbb{C} = \sum_{a=0}^{n-1} w_{pa}(x)v$ where n is the number of processors. A write operation $w_{pi}(x)v$ is *performed with respect to processor* p_j when the value v is written to the address x on the local memory m_j of p_j. A write operation $w_{pi}(x)v$ is *performed* when it is performed with respect to all processors that compose the system.

At the beginning of the execution, we assume that all memory locations are initialized to 0. At the end of the execution, all memory operations must be performed. We assume that each processor executes only one process and that two processors cannot write the same value to the same memory position. This second assumption may appear rather restrictive but it is necessary in some order definitions. However, it can be easily overcome in a real system by adding timestamps to the value *v*.

Each process running on a processor p_i is described by a local execution history $\mathbb{H}_{pi}$, that is an ordered sequence of memory operations issued by p_i.

The execution history $\mathbb{H}$ is the union of all $\mathbb{H}_{pi}$. A graphical representation of $\mathbb{H}$ is shown in Figure 1.

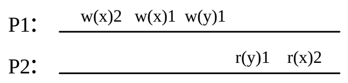

Figure 1. An Execution History

In Figure 1, the notation *w(x)v* represents the instant where the write of the value *v* on memory position *x* was issued and *r(x)v* represents the instant where the read of value *v* on memory position *x* was performed. Time corresponds to the horizontal axis and increases from left to right.

In our definitions, we use the notion of *linear sequences.* If $\mathbb{Q}$ is a history, a *linear sequence* of $\mathbb{Q}$ contains all operations in $\mathbb{Q}$ exactly once. A linear sequence is *legal* if all read operations *r(x)v* return the value written by the most recent write operation on the same address in the sequence.

In execution histories, we have some operation orderings that are allowed and some orderings that are forbidden . The decision of which orderings are valid is made by the *memory consistency model*. This observation leads to our memory consistency model definition:

Definition 1. Memory Consistency Model: A *memory consistency model* defines an order relation ($\overset{R}{\rightarrow}$) on a set of shared memory accesses ($\mathbb{H}_x$).

One execution history is valid on a memory consistency model if it respects the order relation defined by the model. In other words, a history $\mathbb{H}$ is valid on a memory consistency model if there is at least one legal linear sequence on $\mathbb{H}_x$ that respects $\overset{R}{\rightarrow}$.

An order relation that is used in the definition of all memory consistency models proposed until now in the literature is *program order* ($\overset{po}{\rightarrow}$).

Program-order (po). An operation o_1 is related to an operation o_2 by program-order ($o_1 \overset{po}{\rightarrow} o_2$) if:

a) both operations are issued by the same processor p_i and o_1 immediately precedes o_2 in the code of p_i or

b) $\exists\ o_3$ such that $o_1 \overset{po}{\rightarrow} o_3$ and $o_3 \overset{po}{\rightarrow} o_2$.

The program order used in our definitions is a total order on $\mathbb{H}_{pi}$ that relates operations according to the order in which they appear in the program code of each process.

3. Defining Uniform Models

In this section, five uniform models are defined using our framework and related to each other. The first two models are strong memory models because they impose an order relation on all shared memory accesses, i.e, they impose a total order on $\mathbb{H}$. The other three models are relaxed memory models because they impose an order relation only to a subset of the shared memory accesses. In the second case, every processor has its own view of the shared memory.

3.1 Dynamic Atomic Consistency

Atomic consistency is the strongest and the oldest memory consistency model. It defines a total order on all shared memory accesses ($\mathbb{H}$) and, besides, imposes that real time order must be preserved. In order to define what orderings are valid when more than one access occur in the same real time, many variations of atomic consistency have been proposed. Our definition refers to the model known as dynamic atomic consistency [23].

Before defining dynamic atomic consistency, we must define the global time function (g*t*):

Function gt: *The function gt(ξ) returns the value of a global clock when the event ξ occurs.*

Our formal definition of dynamic atomic consistency is derived from the definition in [29]:

Definition 2. Dynamic Atomic Consistency: A history H is *dynamic atomically consistent* if there is a legal linear sequence of $\mathbb{H}$ that respects the order $\overset{AT}{\rightarrow}$ which is defined as follows:

i) $\forall\ o_1, o_2$: if $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{AT}{\rightarrow} o_2$ and

ii) $\forall\ o_1, o_2$: if gt(perform(o_1)) < gt(issue(o_2)) then $o_1 \overset{AT}{\rightarrow} o_2$.

In this definition, we define a new order relation ($\overset{AT}{\rightarrow}$) where all processors must perceive the same execution order of <u>all</u> shared memory accesses (legal linear sequence of $\mathbb{H}$). In this order, all operations performed by a process must respect the program order $\overset{po}{\rightarrow}$ (i). Also, non-overlapping memory accesses must respect real-time order (ii). In this definition, issuing and performing memory accesses are seen as events.

We do not have examples of distributed shared memory systems that implement dynamic atomic consistency. Preserving real time order for every shared memory access is very time-consuming in systems where no global physical clocks exists.

3.2 Sequential Consistency

Sequential Consistency was proposed by [22] as a correctness criterion for shared memory multiprocessors. Many early distributed shared memory systems proposed in the literature implement sequential consistency. Ivy [25], Mirage [10] and KOAN [20] are examples of sequentially consistent distributed shared memory systems.

Our definition of sequential consistency is derived from the definition presented in [4]:

Definition 3. Sequential Consistency: A history $\mathbb{H}$ is *sequentially consistent* if there is a legal linear sequence of $\mathbb{H}$ that respects the order $\overset{SC}{\rightarrow}$ which is defined as follows:

i) $\forall\ o_1, o_2$: if $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{SC}{\rightarrow} o_2$.

For sequential consistency, we define a new order $\overset{SC}{\rightarrow}$. Like Dynamic Atomic Consistency, SC requires a total order on $\mathbb{H}$. The only difference between these two models is that preserving real-time order is no longer necessary in sequential consistency.

3.3 Causal Consistency

Causal consistency is a relaxed memory model that is based on the potential causal order relation defined by [21]. Causal consistency was implemented by the researchers that proposed it [3] and was also implemented in the system Clouds [17].

Since the potential causal order relation is defined to be the irreflexive transitive closure of two order relations (program order and read-by order) [3], we must define the read-by order before presenting the definition of Causal Consistency.

Read-by Order (rb). A write operation *w(x)v* is read by a read operation *r(x)v* ($w(x)v \overset{rb}{\rightarrow} r(x)v$) if the operation *r(x)v*, issued by processor p_i, reads the value written by the operation *w(x)v* issued by processor p_j and $i \neq j$.

Also, we must define a new execution history:

History $\mathbb{H}_{pi+w}$. Let $\mathbb{H}$ be the global execution history and p_i be a processor. The history $\mathbb{H}_{pi+w}$ is called *the history of writes with respect to processor p_i* and it is a sub-history of $\mathbb{H}$ that contains all operations issued by processor p_i and all write operations issued by the other processors.

For the formal definition of causal consistency, we define an order on $\mathbb{H}_{pi+w}$. Our definition is derived from the one proposed by [17]:

Definition 4. Causal Consistency: A history $\mathbb{H}$ is *causally consistent* if there is a legal linear sequence of $\mathbb{H}_{pi+w}$ that respects the order $\overset{CA}{\rightarrow}$ which is defined for each processor p_i as follows:

i) $\forall$ o_1, o_2: if $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{CA}{\rightarrow} o_2$ and

ii) $\forall$ o_1, o_2: if $o_1 \overset{rb}{\rightarrow} o_2$ then $o_1 \overset{CA}{\rightarrow} o_2$ and

iii) $\forall$ o_1, o_2, o_3: if $o_1 \overset{CA}{\rightarrow} o_2$ and $o_2 \overset{CA}{\rightarrow} o_3$ then $o_1 \overset{CA}{\rightarrow} o_3$.

By this definition, we can see that it is no longer necessary for all processors to agree on the order of all shared memory operations. Instead of using $\mathbb{H}$, we use a subset of it, $\mathbb{H}_{pi+w}$. Every processor has its own view ($\mathbb{H}_{pi+w}$) of the shared global memory $\mathbb{M}$ and the order $\overset{CA}{\rightarrow}$ is defined for $\mathbb{H}_{pi+w}$. In this view, the program order of $\mathbb{H}_{pi+w}$ must be respected (i), the order *read-by* must be respected (ii) and the transitivity of the new order $\overset{CA}{\rightarrow}$ must be preserved (iii).

3.4 Processor Consistency

Processor Consistency is perhaps the most clear example of the problems that can arise if we do not define consistency models in a formal way. In fact, processor consistency is a family of memory consistency models that are based upon the same idea but have small differences. These differences led to different memory behaviors and, consequently, to different memory consistency models. The basic idea of these memory consistency models is to relax some conditions imposed by sequential consistency and to require only that write operations issued by the *same processor* are observed by all processors in the order they were issued. In other words, among the operations issued by other processors, only write operations must obey the program order.

Many parallel machines implement processor-consistency related models, such as the VAX8800 and the SPARC V8. The most important models in the processor consistency family are PRAM Consistency [26] and Processor Consistency as defined by [12] (PCG). These two models differ basically because the latter requires that all processors must observe the writes to the same memory location on the same order[2] whereas the former does not. In this article, we present the formal definition of PRAM consistency. The formal definition of PCG can be found in [5].

Definition 5. PRAM Consistency: A history $\mathbb{H}$ is *PRAM consistent* if there is a legal linear sequence of $\mathbb{H}_{pi+w}$ that respects the order $\overset{PRAM}{\rightarrow}$ which is defined for each processor p_i as follows:

i) $\forall$ o_1, o_2: if $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{PRAM}{\rightarrow} o_2$.

By this definition, we can see that the PRAM order is defined on $\mathbb{H}_{pi+w}$. In this view, the program order must be respected. That means that, for each processor p_i, the program order of p_i and the program order of the writes issued by the other processors must be respected (i).

3.5 Slow Memory

Slow Memory was proposed by [15] and is one of the most relaxed memory consistency models. Slow Memory only requires that writes issued by the same processor on the same memory location must be seen by the other processors in program order. This is really a weak condition. Nearly all traditional mutual exclusion solutions fail on slow memory. In such a memory, the history presented in Figure 1 would be valid.

Definition 6. Slow Memory: A history $\mathbb{H}$ is *slow consistent* if there is a legal linear sequence of $\mathbb{H}_{pi+w}$ that respects the order $\overset{SL}{\rightarrow}$ which is defined for each processor p_i as follows:

[2] In the literature, this condition is often called cache coherence or data coherence.

i) $\forall$ o_1, o_2: if processor(o_1)=processor(o_2)=p_i and $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{SL}{\rightarrow} o_2$, and

ii) $\forall$ o_1, o_2: if processor(o_1) = processor (o_2) and address(o_1) = address(o_2) and $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{SL}{\rightarrow} o_2$.

By the above definition, we can see that only the local program order must be respected (i) and all processors must agree about the processor write order on the same memory location (ii). As in PRAM Consistency, all processors must eventually see all write operations issued by all processors since the order is defined on $\mathbb{H}_{pi+w}$. However, it is no longer necessary to fully respect program order on $\mathbb{H}_{pi+w}$.

3.6 Relating Uniform Memory Models

Defining Memory Consistency Models formally makes it easier to compare and relate them. Figure 2 shows the relation among the previously defined models. Each rectangle represents the possible results that can be produced under the named Memory Model.

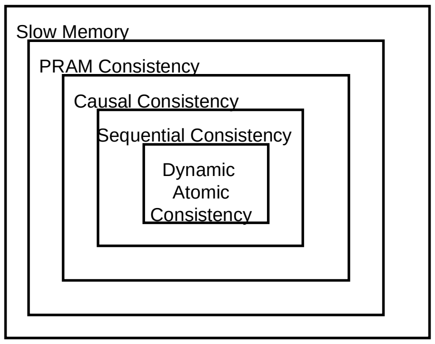

Figure 2. Relating Uniform Models

In Figure 2, the strongest memory model is Dynamic Atomic Consistency because it imposes a total order on $\mathbb{H}$ <u>and</u> requires real time order to be preserved. Sequential Consistency imposes only a total order on $\mathbb{H}$. Henceforth, Dynamic Atomic Consistency is strictly stronger than Sequential Consistency. That means that every history which obeys Dynamic Atomic Consistency also respects Sequential Consistency. Causal Consistency is strictly stronger than PRAM Consistency. Both Memory Consistency Models require that program order on $\mathbb{H}_{pi+w}$ must be preserved. However, Causal Memory requires <u>also</u> that all processors must respect the *read-by* order. PRAM Consistency is strictly stronger than Slow Memory since Slow Memory only requires program order on $\mathbb{H}_{pi+w}$ to be respected when some conditions are true.

By Figure 2, we can induce erroneously that memory models are always comparable. However, many memory models are incomparable. Causal Consistency and PCG are an example of incomparable memory models.

4. Defining Hybrid Memory Consistency Models

Even on uniprocessor systems, processes that use only the basic shared memory operations to communicate can produce unexpected results. For this reason, synchronization operations must be used every time processes want to restrict the order on which memory operations should be performed. Using this fact, hybrid Memory Consistency Models guarantee that processors only have a consistent view of the shared memory at synchronization time. This allows a great overlapping of basic memory accesses that can potentially lead to considerable performance gains.

Defining hybrid memory models is more complex than defining uniform ones. This is basically for three reasons. First, there are more types of operations that must be considered. There is at least one more type of memory operation: *sync* (synchronization type). Second, there are at least two different orders: one order that relates basic operations to synchronization operations and one order that relates synchronization operations exclusively. Third, and perhaps the most important reason, we do not consider program order anymore to relate operations. To order operations, we use a relaxation of program order called *comes-before order*.

Comes-Before Order (cb). An operation $o_1(x)v$ comes before an operation $o_2(x)v$ ($o_1(x)v \overset{cb}{\rightarrow} o_2(x)v$) if:

i) (type(o_1) = sync and type(o_2) ≠ sync) or (type(o_1) ≠ sync and type(o_2) = sync) and $o_1 \overset{po}{\rightarrow} o_2$; or

ii) type (o_1) = write and type(o_2) = write and address(o_1) = address(o_2) and $o_1 \overset{po}{\rightarrow} o_2$ and $\exists$ o_3 such that $o_1 \overset{po}{\rightarrow} o_2 \overset{po}{\rightarrow} o_3$ where type(o_3) = sync.

By this definition, only synchronization operations that precede basic operations in the program code and vice-versa are ordered (i). To preserve intraprocessor dependencies, we order also all write operations issued by the same processor on the same address according to program order if there is a synchronization operation that follows the write operations (ii).

In the rest of this section, the formal definitions of four hybrid memory models are presented.

4.1 Weak Ordering

Weak ordering was the first consistency model to make the distinction between basic and synchronization operations. It was originally defined by [9] and later redefined by [1]. In the present paper, we will use the original definition: *A system is weakly ordered if (1) accesses to global synchronizing variables are strongly ordered; (2) no access to a synchronizing variable is issued by a processor before all previous global data accesses have been globally performed and if (3) no access to global data is issued by a processor before a previous access to a synchronization variable has been globally performed.* In [9], it is stated that intraprocessor dependencies and local program order must be respected. We assume also that conditions (2) and (3) refer to all global data accesses issued by the same processor that issues a synchronizing access.

In weak ordering, there are three operation types: read (*r*), write (*w*) and synchronization (*sync*). There is also a new execution history:

History $\mathbb{H}_{pi+w+sync}$. Let $\mathbb{H}$ be the global execution history and p_i be a processor. The history $\mathbb{H}_{pi+w+sync}$ is called *the history of writes and synchronization operations with respect to processor p_i* and it is a sub-history of $\mathbb{H}$ that contains all operations issued by processor p_i and all write and sync operations issued by the other processors.

Definition 7. Weak Ordering: A history $\mathbb{H}$ is *weakly ordered* if there is a legal linear sequence of $\mathbb{H}_{pi+w+sync}$ that respects the order $\overset{WO}{\rightarrow}$ which is defined for each processor p_i as follows:

i) $\forall$ o_1, o_2: if type(o_1)=type(o_2)=*sync* and $\exists$ $\mathbb{H}_{pj+w+sync}$ where $o_1 \overset{H_{pj+w+sync}}{\longrightarrow} o_2$ then $o_1 \overset{WO}{\rightarrow} o_2$ and

ii) $\forall$ o_1, o_2: if $o_1 \overset{cb}{\rightarrow} o_2$ then $o_1 \overset{WO}{\rightarrow} o_2$ and

iii) $\forall$ o_1, o_2: if processor(o_1)=processor(o_2)=p_i and $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{WO}{\rightarrow} o_2$ and

iv) $\forall$ o_1, o_2, o_3: if $o_1 \overset{WO}{\rightarrow} o_2$ and $o_2 \overset{WO}{\rightarrow} o_3$ then $\overset{WO}{\rightarrow} o_3$.

In this definition, we state that synchronization operations must be seen on the same order by all processors, i.e, they must be sequentially consistent (i), the comes-before order must be respected (ii), the local program order must be respected (iii) and the transitivity of $\overset{WO}{\rightarrow}$ must be guaranteed (iv).

4.2 Release Consistency

Release consistency was defined by [11] and it is one of the most popular hybrid memory models. Release Consistency is a relaxation of weak ordering where competing accesses are called special accesses. Special accesses are divided into synchronization operations and non-synchronization operations. There are two subtypes of synchronization operations: *acquire* accesses and *release* accesses.

Informally, in release consistent systems, it must be guaranteed that: *before an ordinary access performs, all previous acquire accesses must be performed; and before a release performs with respect to any other processor, all previous ordinary accesses must be performed*. There is also a third condition that requires special accesses to be processor consistent. We find in the literature many systems that implement release consistency: Munin [8] and TreadMarks[18] are examples of release consistent distributed shared memory systems. DASH [24] is a release consistent parallel architecture.

In order to define release consistency formally, we must before define the synchronization order (proposed by [1]):

Synchronization Order (so). An operation $o_1(x)v$ is ordered before $o_2(x)v$ by the synchronization order ($o_1(x)v \overset{so}{\rightarrow} o_2(x)v$) if :
i) type(o_1) = type(o_2) = *sync* and
ii) *gt*(perform(o_1)) < *gt*(perform(o_2)).

Our definition of release consistency is derived from the one presented by [19]. In this definition, release accesses are seen as special write accesses. Similarly, acquire accesses are treated as special read accesses. For this reason, we use a new execution history:

History $\mathbb{H}_{pi+w+release}$. Let $\mathbb{H}$ be the global execution history and p_i be a processor. The history $\mathbb{H}_{pi+w+release}$ is called *the history of writes and release operations with respect to processor p_i* and it is a sub-history of $\mathbb{H}$ that contains all operations issued by processor p_i and all write and release operations issued by the other processors.

Definition 8. Release Consistency: A history $\mathbb{H}$ is *release consistent* if there is a legal linear sequence of $\mathbb{H}_{pi+w+release}$ that respects the order $\overset{RC}{\rightarrow}$ which is defined for each processor p_i as follows:

i) $\forall$ o_1, o_2, o_3: if $o_1 \overset{so}{\rightarrow} o_2 \overset{cb}{\rightarrow} o_3$ on $\mathbb{H}$ and subtype(o_1) = *release* and subtype(o_2) = *acquire* and type(o_3) $\in$ {r,w} then $o_1 \overset{RC}{\rightarrow} o_3$ and

ii) $\forall$ o_1, o_2: if $o_1 \overset{cb}{\rightarrow} o_2$ on $\mathbb{H}_{pi+w+release}$ and type(o_1) $\in$ {r,w} and subtype(o_2) = *release* then $o_1 \overset{RC}{\rightarrow} o_2$ and

iii) $\forall$ o_1, o_2: if processor(o_1)=processor(o_2)=p_i and $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{RC}{\rightarrow} o_2$.

In short, the definition of $\overset{RC}{\rightarrow}$ imposes that synchronization order $\overset{so}{\rightarrow}$ must be preserved. In addition, all basic memory operations that follow the acquire must be ordered after the acquire (i) and all basic memory operations must be performed before the release is issued (ii). Condition (iii) simply states that program order of p_i must be preserved in p_i's view.

4.3 Entry Consistency

Entry Consistency is a hybrid memory consistency model proposed by [7] that establishes a relation between synchronization variables used to enter a critical section and the shared variables guarded by this critical section. Under entry consistency, a consistent view of part of the memory is only guaranteed at the moment a processor acquires a lock and enters a critical section.

According to the informal definition, *a system is entry consistent if an acquire access on lock* s *is not allowed to perform with respect to processor p_i until all (previous) updates to the shared data guarded by* s *have been performed with respect to processor p_i.* Entry consistency defines the set D_s as the set of shared variables that are guarded by synchronization variable s. The function *associate(s, d)* includes the shared data *d* on the set D_s. Entry consistency allows a synchronization access to be exclusive or non-exclusive. In our definition, we will consider only exclusive-mode synchronization accesses.

For the formal definition of Entry Consistency, we must define the *data_guarded* function*:*

Function data_guarded: *data_guarded(x) = {y | y is an address guarded by x}*

Two new orders must also be defined: *synchronization-order-1* and a *data-guarded-comes-before order.*

Synchronization-Order-1 (so1). An operation $o_1(x)v$ is ordered before $o_2(x)v$ by the synchronization order-1 ($o_1(x)v \overset{so1}{\rightarrow} o_2(x)v$) if

i) $o_1(x)v \overset{so}{\rightarrow} o_2(x)v$ and
ii) address(o_1) = address (o_2).

Data-Guarded-Comes-Before (dgcb). An operation $o_1(x)v$ is ordered before $o_2(x)v$ by the data-guarded-comes-before order ($o_1(x)v \overset{dgcb}{\rightarrow} o_2(x)v$) if $o_1(x)v \overset{cb}{\rightarrow} o_2(x)v$ and

i) subtype(o_1) = acquire and type(o_2) $\in$ {r,w} and address(o_2) $\in$ data_guarded(address(o_1)) or
ii) type(o_1) $\in$ {r,w} and subtype(o_2) = release and address(o_1) $\in$ data_guarded(address(o_2));

By synchronization-order-1, only synchronization operations to the same address are related. Also, data-guarded-comes-before relates only synchronization operations to shared data operations that are guarded by them.

Definition 9. Entry Consistency: A history $\mathbb{H}$ is *entry consistent* if there is a legal linear sequence of $\mathbb{H}_{pi+w+release}$ that respects the order $\overset{EC}{\rightarrow}$ which is defined for each processor p_i as follows:

i) $\forall$ o_1, o_2, o_3: if $o_1 \overset{so1}{\rightarrow} o_2 \overset{dgcb}{\rightarrow} o_3$ in $\mathbb{H}$ and subtype(o_1) = *release* and subtype(o_2) = *acquire* and type(o_3) $\in$ {r,w} then $o_1 \overset{EC}{\rightarrow} o_3$ and

ii) $\forall$ o_1, o_2: if $o_1 \overset{dgcb}{\rightarrow} o_2$ on $\mathbb{H}_{pi+w+release}$ and type(o_1) $\in$ {r,w} and subtype(o_2) = *release* then $o_1 \overset{EC}{\rightarrow} o_2$ and

iii) $\forall$ o_1, o_2, o_3: if $o_1 \overset{EC}{\rightarrow} o_2$ and $o_2 \overset{EC}{\rightarrow} o_3$ then $o_1 \overset{EC}{\rightarrow} o_3$ and

iv) $\forall$ o_1, o_2: if processor(o_1)=processor(o_2)=p_i and $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{EC}{\rightarrow} o_2$.

The definition of $\overset{EC}{\rightarrow}$ states that (i) a release synchronization access on lock *x* must be ordered before all shared data accesses guarded by *x* that occurs after the next acquire to *x*. This is exactly what states the informal definition of Entry Consistency. However, it is also necessary (ii) to order accesses to shared data guarded by a lock *x* before the release on lock *x*. This relation and transitivity of EC (iii) guarantee that all previous accesses to shared data guarded by *x* will be performed before the next acquire on *x* is performed. Also, local program order must be preserved (iv).

4.4 Scope Consistency

The goal of Scope Consistency is to take advantage of the association between synchronization variables and ordinary shared variables they protect in a reasonably easy programming model. It was proposed by [16]. In scope consistency, executions are divided into consistency scopes that are defined in a per lock basis. Just like Entry Consistency, Scope Consistency orders only synchronization and data accesses that are related to the same synchronization variable. However, the association between shared data and the synchronization variable that guards them is implicit and depends on program order. Informally, a system is scope consistent if *(1) before a new section of a consistency scope is allowed to open at process P, any write previously performed with respect to that consistency scope must be performed with respect to P; and (2) A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened.*

As it can be easily seen, Scope Consistency and Entry Consistency are very similar memory models. The difference is that Scope Consistency considers that data inside a critical section guarded by lock *l* are automatically associated with *l* whereas Entry Consistency requires this association to be provided by the programmer.

In order to define scope consistency formally, we will define a new comes-before related order:

Scope-Comes-Before (scopcb). An operation $o_1(x)v$ is ordered before $o_2(x)v$ by the scope-comes-before order ($o_1(x)v \overset{scopcb}{\rightarrow} o_2(x)v$) if :

i) $\exists$ o_3 such that $o_3 \overset{cb}{\rightarrow} o_1 \overset{cb}{\rightarrow} o_2$ and subtype(o_3) = acquire and type(o_1) $\in$ {r,w} and subtype(o_2) = release and address(o_3) = address(o_2), or

ii) subtype(o_1) = acquire and type(o_2) $\in$ {r,w} and $\nexists$ o_3 such that $o_1 \overset{cb}{\rightarrow} o_3 \overset{cb}{\rightarrow} o_2$ where subtype(o_3) = release and address(o_1)= address(o_3).

Definition 10. Scope Consistency: A history $\mathbb{H}$ is *scope consistent* if there is a legal linear sequence of $H_{pi+w+release}$ that respects the order $\overset{SCOP}{\rightarrow}$ which is defined for each processor p_i as follows:

i) $\forall$ o_1, o_2, o_3: if $o_1 \overset{so1}{\rightarrow} o_2 \overset{scopcb}{\rightarrow} o_3$ in $\mathbb{H}$ and subtype(o_1) = *release* and subtype(o_2) = *acquire* and type(o_3) $\in$ {r,w} then $o_1 \overset{SCOP}{\rightarrow} o_3$ and

ii) $\forall$ o_1, o_2: if $o_1 \overset{scopcb}{\rightarrow} o_2$ on $\mathbb{H}_{pi+w+release}$ and type(o_1) $\in$ {r,w} and subtype(o_2) = *release* then $o_1 \overset{SCOP}{\rightarrow} o_2$ and

iii) $\forall$ o_1, o_2, o_3: if $o_1 \overset{SCOP}{\rightarrow} o_2$ and $o_2 \overset{SCOP}{\rightarrow} o_3$ then $o_1 \overset{SCOP}{\rightarrow} o_3$ and

iv) $\forall$ o_1, o_2: if processor(o_1)=processor(o_2)=p_i and $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{SCOP}{\rightarrow} o_2$.

As stated before, the only difference between Entry Consistency and Scope Consistency is the order used to establish the operations that are guarded by critical sections. In the case of Scope Consistency, we use the Scope-Comes-Before order while Data-Guarded-Comes-Before is used in Entry Consistency.

4.5 Relating Hybrid Models

The same Venn Diagram-like representation presented in section 3.6 is used in figure 3 in order to relate the previously defined hybrid memory consistency models.

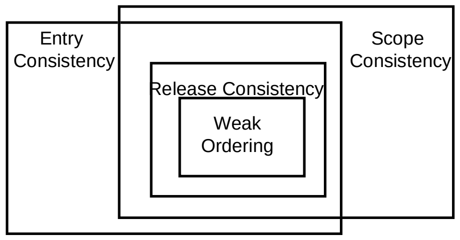

Figure 3 - Relation among hybrid models

By the formal definitions, it is quite clear that, among the hybrid models, weak ordering is the strongest one. It imposes that all shared memory accesses previous to a synchronization access must be performed before the synchronization access performs and that no shared data access must be issued until all previous synchronization accesses are performed. Release Consistency is a model that divides the unique synchronization access of weak ordering into two distinct accesses: release and acquire. The first condition of weak consistency refers only to release accesses in release consistency while the second one refers only to acquire accesses. That is why Weak Ordering is strictly stronger than Release Consistency. While both Entry Consistency and Scope Consistency are strictly weaker than Release Consistency, they are not comparable. Depending on the association made between locks and shared data by the programmer, Entry Consistency can produce histories that are not valid in Scope Consistency and vice-versa.

5 Related Work

Adve [2] has proposed a quite complete methodology to specify memory models. A great number of memory models were considered. The aim of her work, however, was to define relaxed models in terms of sequential consistency. The central hypothesis of this study is that all parallel programmers would prefer to reason as if they were programming a time-sharing uniprocessor.

A set of formal definitions was also proposed in [29]. The objective of this study was to understand memory models and compare them. The following memory models were defined using the proposed notation: atomic consistency, sequential consistency, causal consistency and PRAM consistency.

Kohli et al. [19] also proposed a formal framework to relate memory models where sequential consistency, TSO[3], processor consistency and release consistency were defined.

Heddaya and Sinha [13] proposed a formalism and a system - Mermera - that permits multiple memory models in a single execution. Shared memory is accessed by read and write primitives. Reads are always local. The following memory models were formally defined: sequential consistency, PRAM consistency, slow memory and local consistency.

Recently, Higham et al. [14] proposed a new framework to describe memory consistency models. The authors defined atomic consistency, sequential consistency, data coherence, PRAM consistency, processor consistency, weak ordering, TSO and PSO.

Our work has some similarities with [14] and [29]. However, we propose more generic function-based definitions that permit very relaxed models to be described in a simple way. As far as we know, our work is the first one to propose a unified framework where per-lock basis relaxed hybrid memory models are formally defined.

6 Conclusion and Future Work

In this article, we presented a unified framework to describe memory consistency models in a formal way. A large set of well-known uniform and hybrid Memory Consistency Models were defined. We also related memory models to each other. We claim that this task is much easier when memory models are defined formally.

As future work, we intend to propose a unified framework where memory coherence protocols can be specified. Having memory consistency models and memory coherence protocols definitions on unified frameworks, we will investigate the semi-automatic generation of DSM systems when a pair (memory consistency model, memory coherence protocol) is chosen.

[3] TSO and PSO are memory models where ordinary accesses are ordered by processor-consistent related models and synchronization operations are similar to the sync operation on weak ordering.

7 References

[1] S. V. Adve, M. Hill, "Weak Ordering - A New Definition", *Proc. 17th Annual International Symposium on Computer Architecture*, pages 2-11, 1990.

[2] S. V. Adve, "Designing Multiple Memory Consistency Models for Shared-Memory Multiprocessors", PhD dissertation, University of Wisconsin-Madison, 1993.

[3] M. Ahamad, P. Hutto, R. John, "Implementing and Programming Causal Distributed Shared Memory", Technical Report GIT-CC-90/49, Georgia Institute of Technology, 1990.

[4] M. Ahamad et al., "The Power of Processor Consistency", Technical Report GIT-CC-92/34, Georgia Institute of Technology, 1992.

[5] A. Balaniuk, "Conception d'un Système Supportant des Modèles de Cohérence Multiples pour les Machines Parallèles à Mémoire Virtuelle Partagée", PhD dissertation, Institut National Polytechnique de Grenoble, France, 1996.

[6] A. Balaniuk, "Multiple Memory Consistency Models on a SVM Parallel Programming Environment", *Proc Int. Conf. on Principles of Distributed Systems (OPODIS)*, December, 1997, pages 249-260.

[7] B. Bershad and M. Zekauskas, "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors", Tech Report CMU-CS-91170, CMU, 1991.

[8] J. Carter, "Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency", PhD dissertation, Rice University, 1993.

[9] M. Dubois, C. Scheurich and F. Briggs, "Memory Access Buffering in Multiprocessors", *Proc 13th Annual International Symposium on Computer Architecture*, pages 434-442, 1986.

[10] B. Fleisch and G. Popek, "Mirage: a Coherent Distributed Shared Memory Design", *Proc 14th ACM Symposium on Operating Systems Principles*, pages 211-221, 1989.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proc. Int. Symp. On Computer Architecture*, May, 1990, p15-24.

[12] J. Goodman, "Cache Consistency and Sequential Consistency", Technical Report 61, IEEE Scalable Coherent Interface Working Group, March, 1989.

[13] A. Heddaya and H. Sinha, "An Implementation of Mermera: a Shared Memory System that Mixes Coherence with Non-Coherence", Tech Report BU-CS-93-006, Boston University, 1993.

[14] L. Higham, J.Kawash and N.Verwaal, "Defining and Comparing Memory Consistency Models", *Proc. ACM Int Conf on Parallel and Distributed Computing Systems (PDCS)*, October, 1997.

[15] P. Hutto and M. Ahamad, "Slow Memory: Weakening Consistency to Enhance Concurrency on Distributed Shared Memories", *Proc 10th Int. Conf. on Distributed Computing Systems (DCS)*, 1990, p.302-309.

[16] Iftode, L., Singh, J. P., and Li, K. "Scope Consistency: A Bridge between Release Consistency and Entry Consistency" . *Proc 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277-287, June 1996.

[17] R. John and M. Ahamad, "Causal memory: Implementation, Programming Support and Experiences", Technical Report GIT-CC-93/10, Georgia Institute of Technology, 1993.

[18] P. Keleher, S. Dwarkadas, A. Cox, W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", *Proc. of the Winter 94 Usenix Conference*, pages 115-131, January, 1994.

[19] P. Kohli, G. Neiger, M. Ahamad, "A Characterization of Scalable Shared Memories", Technical Report GIT-CC-93/04, Georgia Institute of Technology, 1993.

[20] Z. Lahjomri and T. Priol, "KOAN: a Shared Virtual Memory for the iPSC/2 Hypercube", *Lecture Notes on Computer Science 634*, pages 442-452, 1992.

[21] L. Lamport, "Time, Clocks and Ordering of Events in a Distributed System", *Communications of the ACM*, pages 558-565, 1978.

[22] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Transactions on Computers*, pages 690-691, 1979.

[23] L. Lamport, *Distributed Computing*, Chapter: On Interprocess Communication; Part I - Basic Formalism, pages 77-85, 1986.

[24] D. Lenosky et al. "The DASH Prototype: Logic Overhead and Performance", *IEEE Transactions on Parallel and Distributed Systems*, January, 1993.

[25] K. Li, "Shared Virtual Memory on Loosely Coupled Architectures", PhD dissertation, Yale University, 1986.

[26] R. Lipton and J. Sandberg, "PRAM: A Scalable Shared Memory", Technical Report 180-88, Princeton University, September, 1988.

[27] D. Mosberger, "Memory Consistency Models", *Operating Systems Review*, pages 18-26, 1993.

[28] B. Nitzberg and V. Lo. "Distributed Shared Memory: A Survey of Issues and Algorithms", *IEEE Computer*, pages 52-60, 1991.

[29] M. Raynal and A. Schiper, "A Suite of Formal Definitions for Consistency Criteria in Distributed Shared Memories", Technical Report PI-968, IRISA, France, 1995.