CHAPTER 4

STRUCTURAL HYBRID MEMORY MODEL

The Hybrid memory consistency model considers not only read, write operation but it also consider synchronization of processes. Hybrid memory consistency model is dependent to process’s activity. So in this model the synchronization of process is required. In this chapter we will discuss about hybrid memory models in detail. For this we have designed structural hybrid memory model on unified framework for all memory consistency models belonging to hybrid memory models. All consistency models of hybrid memory models are described on the basis of this unified framework.

4.1 Structural Hybrid Memory Model – Design

A hybrid memory models consider read, write as well as synchronization operation. Processes want to restrict the order on which memory operation should be performed. Using this fact, hybrid memory models guarantee that processors only have a consistent view of the memory at synchronization time. This allows a great overlapping of basic memory accesses that can potentially lead to considerable performance gain.
Figure 4.1 Unified Structure of Hybrid memory models

Defining the Hybrid memory models is more complex than the uniform model because of memory operation, order of operation & relate to operation. We have taken following memory consistency model of distributed shared memory for our structural hybrid memory model:

vi) Weak consistency model (WC)

vii) Release consistency model (RC)
    a) Eager release consistency (ERC)
    b) Lazy release consistency (LRC)

viii) Entry consistency (EC)

ix) Scope consistency (ScC)

x) View based consistency (VC)
The proposed structural hybrid memory model is transparent and based on location transparency, migration transparency and parallelism transparency. With Location Transparency, the user can not know where the most recent value is, with Migration Transparency, the user will not feel the existence of other users in the system using the same shared object and finally, Parallelism Transparency: can only be achieved in read operation.

A unified structure of hybrid memory models is shown in Figure 4.1 which shows the execution history of hybrid memory models. The execution history $H$ of a process $P_i$ is an ordered sequence of memory operation issued by the process $P_i$. The execution history of a system is the union of execution history of all process i.e.

$$H = H_{P_1} \cup H_{P_2} \cup H_{P_3} \cup \cdots \cup H_{P_n} \text{ i.e.}$$

$$H = \bigcup_{i=1}^{n} H_{P_i}$$

Figure 4.2 shows an execution history of process $P_1$ and $P_2$, the notation $W(x)v$ represents the instant, where write of value $v$ on memory position $x$ was issued and the notation $R(x)v$ represents the instant where read of value $v$ on memory position $x$ was performed.

<table>
<thead>
<tr>
<th></th>
<th>$W(x)2$</th>
<th>$W(x)1$</th>
<th>$W(y)1$</th>
<th>$R(y)1$</th>
<th>$R(x)2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2 Execution History of $P_1$ and $P_2$

An event ordering or program ordering is to be maintained if:

i) More than one operation is issued by the same processor or

ii) More than one processor issued the operations at same time.
The event order or program order used in the design of structural hybrid memory model is the total ordering used to define the execution history of the system i.e. $H_{P_i}$. The decision of selecting the order is depend upon the memory consistency model. The arrow in the Figure 4.1 shows the ordering of memory models. The ordering of memory goes weaker from top to down. The execution history provides efficient implementation of synchronization of event ordering of consistency model. To make the structure of hybrid memory models unified gives the idea of conflict access. The unified structure removes the conflict access by maintaining the event ordering among process.

Defining hybrid memory models is more complex than defining uniform ones as 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. For the order operations, we use a relaxation of program order called comes-before order (cb) which can be defined as

**Definition:** An operation $O_1(x)v$ comes before operation $O_2(x)v$ i.e. $O_1(x)v \xrightarrow{cb} O_2(x)v$ if

i. $(\text{Type}(O_1) = \text{sync and Type}(O_2) \neq \text{sync})$ or $(\text{Type}(O_1) \neq \text{sync and Type}(O_2) = \text{sync})$ and $O_1 \xrightarrow{po} O_2$. 


ii. \( \text{Type}(O_1) = \text{sync} \) and \( \text{Type}(O_2) = \text{sync} \) and \( \text{address}(O_1) = \text{address}(O_2) \) and \( O_1 \xrightarrow{\text{PO}} O_2 \) and \( \exists O_3 \) such that \( O_1 \xrightarrow{\text{PO}} O_2 \xrightarrow{\text{PO}} O_3 \) where \( \text{Type}(O_3) = \text{sync} \). 

By this definition, only synchronization operations that precede basic operations in the program code and vice-versa are ordered. To preserve intraprocessor dependencies, we also order 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.

### 4.2 Defining Hybrid Memory Models

We have taken synchronization operation and execution history of a process to define different consistency models of hybrid memory models in the thesis. As per unified framework the hybrid memory models defines an order relation on a set of execution history \( (H_{\Pi}) \) of process \( P \). Hybrid memory models distinguish operations according to their access. Following access mechanism have been taken to define all the models.

i) **Shared Access**: As the system taken for the framework is DSM system so obviously all the access described in the thesis is shared access.

ii) **Competing Access**: If two accesses from different processor have at least one write then this access is competing access.

iii) **Synchronizing Access**: Access used for synchronizing the process is synchronizing access.
iv) **Acquire & Release**: the synchronization access divide into access to acquire locks and access to release locks.

Operation can be classified as special operation. Special operation considers synchronization behavior of process. The hybrid memory models can also be described by the ordinary operation which considers non synchronization operation also. But in this thesis we have taken synchronization operation and execution history of a process to define different consistency models of hybrid memory models.

### 4.2.1 Weak Consistency (WC)

Rather than requiring an update to be propagated and executed at other processors immediately, Weak Consistency requires that all previously-generated updates to be propagated and executed at all processors before a synchronization primitive is allowed to be executed. Thus propagation of updates can be postponed until a synchronization primitive is to be executed. Weak Consistency can achieve use of time by propagating updates to other processors only at synchronization time, rather than at every update time. With use of time, updates can be accumulated and only the final results are propagated in batches at synchronization time [12, 13, 14, 16].

**Definition**: On unified framework a Weak consistency is depending upon execution history \((\text{Hpi}+w+\text{sync})\) of write \((w)\) and synchronization \((\text{sync})\) operation of process \(P_i\). So for operation \(O_1\) and \(O_2\), a weak consistency can be defined as
1. \( \forall O_1, O_2 : \text{if } \text{type}(O_1) = \text{type}(O_2) = \text{sync} \text{ and } \exists \text{ Hpi+w+sync} \text{ between } O_1 \text{ and } O_2 \text{ where } O_1 \xrightarrow{\text{Hpi+w+sync}} O_2 \text{ then } O_1 \xrightarrow{\text{WC}} O_2. \)

2. \( \forall O_1, O_2 : \text{if } O_1 \xrightarrow{\text{ch}} O_2 \text{ then } O_1 \xrightarrow{\text{WC}} O_2 \)

3. \( \forall O_1, O_2 : \text{if } \text{processor}(O_1) = \text{processor}(O_2) = P_i \text{ and } O_1 \xrightarrow{\text{PO}} O_2 \)
   \text{ then } O_1 \xrightarrow{\text{WC}} O_2.

4. \( \forall O_1, O_2, O_3 : \text{if } O_1 \xrightarrow{\text{WC}} O_2 \text{ and } O_2 \xrightarrow{\text{WC}} O_3 \text{ then } O_1 \xrightarrow{\text{WC}} O_3 \)

**Sufficient condition for WC**

i) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and

ii) before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed and

iii) Synchronization accesses are sequentially consistent with respect to one another.

**Limitation of WC**

i) Accesses to synchronization variables are sequentially consistent

ii) No access to a synchronization variable is issued in a processor before all previous data accesses have been performed and

iii) No access is issued by a processor before a previous access to a synchronization variable has been performed. Here the meaning of “previous” is well-defined because it refers to program order.
At the time when a synchronizing access performs, all previous accesses by that processor are guaranteed to have performed and all future accesses by that processor are guaranteed not to have performed. The synchronization model corresponding to these access order constraints is relatively simple.

### 4.2.2 Release Consistency (RC)

Release consistency is one of the popular hybrid memory models. It is a relaxation of weak ordering. It is a refinement of weak consistency in the sense that synchronization accesses are divided into acquire, release [1, 4, 15]. As in release consistency the synchronization operation play important role so it is necessary to define synchronization order. So an operation $O_1(x)v$ is ordered before $O_2(x)v$ by the synchronization order $O_1(x)v \xrightarrow{so} O_2(x)v$ if

i. $\text{Type}(O_1) = \text{Type}(O_2) = \text{sync}$ and

ii. $\text{Gt}(\text{perform}(O_1)) = \text{Gt}(\text{perform}(O_2))$.

**Sufficient condition for RC**

i) Before an ordinary **LOAD** or **STORE** access is allowed to perform with respect to any other processor, all previous acquires must be performed.

ii) Before a release is allowed to perform with respect to any other processor, all previous ordinary **LOAD** or **STORE** access must be performed.
iii) **Special accesses** are sequentially consistent with respect to one another.

So the release consistency depending upon execution history \((\mathbf{Hpi+w+acq+rel})\) of write \((w)\) and synchronization \((\text{sync})\) operation of process \(P_i\) may be defined as:

**Definition:** On unified framework a history \(H\) is *release consistent* if there is a legal linear sequence of \((\mathbf{Hpi+w+acq+rel})\) that respects the order \(\xrightarrow{\text{RC}}\) which is defined for each processor \(p_i\) as follows:

i. \(\forall O_1,O_2,O_3 : \text{If } O_1 \xrightarrow{\text{so}} O_2 \xrightarrow{\text{eb}} O_3 \text{ on } H \text{ and } \text{subtype}(O_1) = \text{release} \text{ and } \text{subtype}(O_1) = \text{acquire} \text{ and } \text{Type}(O_3) \in \{r,w\} \text{ then } O_1 \xrightarrow{\text{RC}} O_3 \text{ and }\)

ii. \(\forall O_1,O_2 : \text{if } O_1 \xrightarrow{\text{eb}} O_2 \text{ on } (\mathbf{Hpi+w+acq+rel}) \text{ and } \text{Type}(O_1) \in \{r,w\} \text{ and } \text{subtype}(O_2) = \text{release} \text{ then } O_1 \xrightarrow{\text{RC}} O_2 \)

iii. \(\forall O_1,O_2 : \text{if processor } (O_1) = \text{processor } (O_2) = P_i \text{ and } \)

\(O_1 \xrightarrow{\text{po}} O_2 \text{ then } O_1 \xrightarrow{\text{RC}} O_2 \)

An execution history is valid on a Memory Consistency Model if there is at least one valid interleaving of memory operations for each processor that compose the system. As RC is a relaxed memory model, processors must only see their own operations and all write and release operations issued by the other processors (history \(\mathbf{Hpi+w+release}\)).
4.2.2.1 Eager Release Consistency (ERC)

The Eager Release Consistency (ERC) model improves WC by removing the update propagation at acquire time. It requires that all previously-generated updates must be propagated to and executed at all processors before a release is allowed to be executed. So update propagation can be postponed until a release is to be executed. ERC takes use of time one step further than the WC model by distinguishing two different synchronization primitives: acquire and release, which are the entry and exit of a critical region respectively. ERC requires that updates be propagated to other processors only at release time [6, 10].

In other words, ERC is more time-selective than the WC model by propagating updates only at the exit of a critical region, instead of at both the entry and exit of a critical region as in the WC model, thus further reducing the number of messages in the system. ERC has the same programmer interface as WC; though it removes update propagation at acquire time.

4.2.2.2 Lazy Release Consistency (LRC)

The Lazy Release Consistency (LRC) model does not require the update propagation at release time. It postpones the update propagation until a processor calls an acquire, at which time it knows which processor is the next one to need the updates. So LRC requires that before any access after an acquire is allowed to be executed, all previously-generated updates must be propagated to and executed at the processor executing the acquire [7, 9].
The Lazy Release Consistency (LRC) model improves the ERC model by performing both use of time and use of processor. LRC can achieve use of time similar to ERC, except the update propagation is further postponed until another processor has successfully executed an acquire [5]. LRC can achieve use of processor by postponing the update propagation until acquire time. At successful acquires, the DSM system is able to know precisely which processor is the next one to access the shared data objects, so updates can be propagated only to that particular processor (or no propagation at all if the next processor is the current processor). By sending updates only to the processor that has just entered a critical region; more messages can be reduced in the LRC model.

4.2.3 Entry Consistency (EC)

The entry consistency [EC] model is even weaker than RC. However, it imposes more restrictions on the programming model. EC is like RC except that every shared variable needs to be associated with a synchronization variable. A synchronizing variable is either a lock or a barrier. The association between a variable and its synchronization variable can change dynamically under program control. Note that this, like slow memory, is a location relative weakening of a consistency model [7, 11]. This has the effect that accesses to different critical sections can precede concurrently, which would not be possible under RC. As in entry consistency synchronization order and data guard comes before order plays most important role so both two new order must be defined.
So an operation $O_1(x)v$ is ordered before $O_2(x)v$ by the synchronization order $(SC_{RC})$ $O_1(x)v \overset{so_{RC}}{\rightarrow} O_2(x)v$ if

i. $O_1(x)v \overset{so}{\rightarrow} O_2(x)v$ and

ii. address $(O_1) = address (O_2)$

An operation $O_1(x)v$ is ordered before $O_2(x)v$ by the data guard comes before order $(dgbc)$ $O_1(x)v \overset{dgbc}{\rightarrow} O_2(x)v$ if

i. $O_1(x)v \overset{cb}{\rightarrow} O_2(x)v$ and subtype$(O_1) = acquire$ and type$(O_2) \in \{r,w\}$ and address$(O_2) \in data\ guard(address(O_1))$ or

ii. type$(O_1) \in \{r,w\}$ and subtype$(O_2) = release$ and address$(O_1) \in data\ guard(address(O_2))$.

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:** On unified framework a history $H$ is entry consistent if there is a legal linear sequence of $(Hpi+w+rel)$ that respects the order $\overset{EC}{\rightarrow}$ which is defined for each processor $pi$ as follows:

i. \( \forall O_1, O_2, O_3 : If \ O_1 \overset{so_{RC}}{\rightarrow} O_2 \overset{dgcb}{\rightarrow} O_3 \) on $H$ 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. \( \forall O_1, O_2 : \text{if } O_1 \xrightarrow{\text{dgc}_{b}} O_2 \text{ on } (Hpi+w+rel) \text{ and Type}(O_1) \in \{r,w\} \) and \( \text{subtype}(O_2) = \text{release} \) then \( O_1 \xrightarrow{\text{EC}} O_2 \)

iii. \( \forall O_1, O_2, O_3 : \text{if } O_1 \xrightarrow{\text{EC}} O_2 \text{ and } O_2 \xrightarrow{\text{EC}} O_3 \) then \( O_1 \xrightarrow{\text{Ec}} O_3 \)

iv. \( \forall O_1, O_2 : \text{if processor } (O_1) = \text{processor } (O_2) = P_i \) and
\( O_1 \xrightarrow{\text{PO}} O_2 \) then \( O_1 \xrightarrow{\text{EC}} O_2 \)

Here the Entry Consistency (EC) model tries to remove the propagation of useless updates in LRC by requiring the programmer to annotate association between ordinary data objects and synchronization data objects. When a processor acquires a synchronization data object, only the updates of the data objects that are associated with the synchronization data object are propagated to the processor. EC achieves the same use of data and use of processor as LRC, since updates are propagated only to the next processor calling an acquire.

EC achieves use of data by only propagating updates of data objects that are associated with a synchronization data object. The association, provided by the programmer, helps the EC model remove the propagation of some updates useless to a processor. With additional use of data, EC can be more efficient than LRC. In addition to requiring a program to be data-race free and properly labeled, EC requires the programmer to annotate the association between ordinary data objects and synchronization data objects in the program.
4.2.4 Scope Consistency (ScC)

The Scope Consistency (ScC) model is very similar to EC, except it can partially automate the association between ordinary data objects and synchronization data objects by introducing the concept of consistency scope. Scope Consistency orders only synchronization and data accesses that are related to the same synchronization variable [2, 11]. Informally, a system is scope consistent if

i) before a new section of a consistency scope is allowed to open a process P, any write previously performed with respect to that consistency scope must be performed with respect to P; and

ii) 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.

In order to define scope consistency formally, we will define a new scope comes-before (ScopCB) related order. An operation $O_1(x)v$ is ordered before $O_2(x)v$ by the scope comes before order $O_1(x)v \xrightarrow{\text{ScopCB}} O_2(x)v$ if $\exists O_3$ such that $O_3 \xrightarrow{\text{cb}} O_1 \xrightarrow{\text{cb}} O_2$ and $\text{subtype}(O_3) = \text{acquire}$ and $\text{type}(O_1) \in \{r,w\}$ and $\text{subtype}(O_3) = \text{release}$ and $\text{address}(O_3) = \text{address}(O_2)$

**Definition:** On unified framework a history $H$ is scope consistent if there is a legal linear sequence of $(Hpi+w+rel)$ that respects the order $\xrightarrow{\text{ScopCB}}$ which is defined for each processor $pi$ as follows:
ScC requires that before any access after an acquire is allowed to be executed, all previously-generated updates of data objects that belong to the corresponding scope, must be propagated to and executed by the processor executing the acquire. Like EC, ScC only propagates the updates of data objects that are in the current consistency scope. The difference is that a consistency scope can automatically establish the association between critical regions and data objects. For non-critical regions, however, scopes have to be explicitly annotated by the programmer. ScC achieves the same use of time, use of processor and use of data as EC. So it can offer the same performance advantages as EC if the scopes are well detected or annotated in the program. ScC improves the programmer interface of EC by requiring programmers to associate scopes with code sections,
instead of data. For critical regions, scopes can be automatically associated; but the programmer has to annotate the scopes explicitly for non-critical regions.

4.2.5 View-Based Consistency (VC)

The View-based Consistency (VC) model is proposed to achieve use of data transparently without programmer annotation. A view is a set of ordinary data objects that a processor has the right to access in a data-race-free program. A processor’s view changes when it moves from one region to another by calling acquire and release. VC requires that before a processor is allowed to enter a critical region or a non-critical region, all previously-generated updates of data objects that belong to the corresponding view, must be propagated to and executed at the processor. The views can be classified in two ways **Critical Region Views (CRVs)** and **Non-critical Region Views (NRVs)**. A CRV is the view of a processor while it executes a critical region, and an NRV is the view of a processor while it executes a non-critical region. In data-race-free programs, a CRV consists of the data objects accessed in the critical region, and an NRV consists of the data objects accessed in the noncritical regions [7, 17, 18].

Conditions for View-based Consistency

i. Before a processor $p_i$ is allowed to enter a critical or non-critical region, all previous write accesses to the ordinary data objects of the CRV or NRV must be performed with respect to $p_i$ according to their causal order.
ii. Before a processor \( \textbf{pi} \) is allowed to pass a barrier primitive, all previous write accesses must be performed with respect to \( \textbf{pi} \) according to their causal order.

iii. The sequential consistency of synchronization data objects must be guaranteed by the implementation of the system primitives such as acquire, release, and barrier.

To selectively update data objects, VC uses view, while EC uses guarded shared data and ScC scope. However, the view in VC is different from in EC and the scope in ScC. Both VC and scope are static and fixed with a particular synchronization data object or a critical region. Even if some data objects are not accessed by a processor in a critical region, they are updated simply because they are associated with the lock or the critical region. The view in VC is dynamic and may be different from region to region. Even for the regions protected by the same lock, the views in them are different and depend on the data objects actually accessed by the processor in the regions. VC achieves the same use of time and use of processor as LRC. It can be more selective than EC and ScC in terms of use of data. VC has the same programmer interface as LRC, ERC, and WC.

4.3 Relating Hybrid Memory Models

Defining hybrid memory consistency models formally makes it easier to compare and relate them. The Figure 4.3 shows the relationship among the different memory consistency models of structural hybrid memory model in \textit{Venn diagram}-like representation. Each Oval shows the possible
results that can be produced under the unified framework. By the fig we can easily see that among the hybrid memory models, weak consistency 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 access are performed. Release consistency is a model that divides the unique synchronization access of weak consistency into two distinct accesses: release & acquire.

The first condition of weak consistency refers to only release access in release consistency while the second one refers only to acquire accesses. That’s why weak consistency is strictly stronger than release consistency. Lazy release consistency tracks the causal dependencies between writes, acquires and release, allowing it to propagate writes lazily only when they are needed. While both entry and scope comparable. Consistency is strictly weaker than release consistency. Scope consistency offers most of the potential performance advantages of entry consistency, without requiring explicit binding of data to synchronization variable. Scope consistency works as a bridge between release consistency and

Figure 4.3. Relation b/w models of hybrid memory models
entry consistency. Entry consistency is weaker consistency than release consistency. When using entry consistency, processes synchronize using locks and barriers; the memory only becomes consistent on entry to the critical section. Entry consistency is described as taking advantage of the relationship between specific synchronization variables which protect critical sections and the shared data accessed within those critical sections. Compared with other DSM hybrid consistency models, view based consistency model can achieve use of data without user annotation and reduce more false sharing effect by only updating data objects in the view of a processor.

### 4.3.1 Weak versus Release Consistency

As we know that in weak consistency before an ordinary read or write access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and Before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary read or write accesses must be performed, and Synchronization accesses are sequentially consistent with respect to one another.

\[
\text{Computation } C_1: \begin{align*}
  p_1 &: w(x) \rightarrow w(x)2S \\
  p_2 &: r(x)1r(x)2S \\
  p_3 &: r(x)2r(x)1S
\end{align*}
\]

satisfies weak consistency because all operation precedes synchronization.

As we know that in release consistency Before an ordinary read or write access is allowed to perform with respect to any other
processor, all previous *acquires* must be performed and before a *release* is allowed to perform with respect to any other processor, all previous ordinary *read* or *write* access must be performed.

\[
\text{Computation } C_2 \begin{cases} 
    p_1 : \text{acq}(L) \, w(x) \, 1 \, w(x) \, 2 \, \text{rel}(L) \\
    p_2 : \text{acq}(L) \, r(x) \, 2 \, r(x) \, 1 \, \text{rel}(L) \\
    p_3 : r(x) \, 1
\end{cases}
\]

satisfies release consistency because all acquire is performed before read & write and all read & write are performed before all release.

### 4.3.2 Release versus Scope Consistency

In figure 4.4 \( P_0 \) updates \( X \) first and then \( Y \). \( Y \) is update in critical section guaranteed by \( L_1 \) lock, while \( X \) is updated outside it. \( P_1 \) acquires \( L_1 \) after \( P_0 \) has released it.

\[
P_1
\begin{align*}
    & X=1 \\
    & \text{acquire}(L_1) \\
    & \text{Release}(L_1)
\end{align*}
\]

\[
P_2
\begin{align*}
    & \text{Acquire}(L_1) \\
    & a=Y \\
    & b=X
\end{align*}
\]

Guaranteed \( y=1 \) (LRC & ScC)

Guaranteed \( x=1 \) (LRC)

**Figure 4.4.** Scope Consistency versus Lazy Release Consistency

In *LRC* after acquiring \( L_1 \), \( P_1 \) is guaranteed to see all writes which occurred at \( P_0 \) before the \( L_1 \) release, so the new value of both \( X \) and \( Y \) are guaranteed to be visible. Whereas *ScC* guarantees only that
P₁ sees the new value of Y, become the release event ensured memory consistency over are consistency scope only.

### 4.3.3 Entry Consistency versus Scope Consistency

Both Entry Consistency and Scope Consistency scheme relax the MCM by taking advantages of the relationship between data and synchronization. Both schemes are particularly effective for application based on extensive use of point to point (lock) synchronization. EC binds data explicitly to locks while the ScC binds dynamic memory references implicitly to locks. The communication volume may be higher in ScC then in EC due to page fragmentation, the page size transfer in ScC allows prefetching to occur.

### 4.4 Verification of Hybrid Memory Models

As we know that the hybrid memory models depends upon read, write and synchronization operation. So for the proper verification of hybrid memory models program order, processor order, execution order, global order and synchronization order is required. All these have been described in previous section. Here WE am going to describe weak program order \(<_{wpo}\) for \(O_1\) and \(O_2\).

**Definition:** Weak Program Order: Given two different operations \(O_1\) and \(O_2\) in the same processor, we say that \(O_1\) is before \(O_2\) in weak program order \(O_1<_{wpo}O_2\) iff

1. \(O_1<_{po}O_2\)
ii. \( O_1 \) and \( O_2 \) are in same data item

iii. \( O_1 \) and \( O_2 \) are synchronized operation

iv. For any ‘\( O \)’ \( O_1 <_{\text{wpo}} O \) and \( O <_{\text{wpo}} O_2 \)

4.4.1 Verifying Weak Consistency (VWC)

**INSTANCES:** \( O \) be the set of all the operations of a computation \( C \) of a system \((P, D)\), \( C \) satisfies a valid total order \((O, <)\) such that \( \{O \cup O <_{\text{wpo}} O, s <_{p}\} \).

**TO VERIFY:** \( C \) is Weak consistent

**VERIFICATION:** Let \((o_1, o_2) \in O\) and \((w_1, w_2) \in W\). It has given that \( C \) satisfies valid total order so it must satisfy the following:

\[
\{O \cup O <_{\text{wpo}} O, s <_{p}\} \subseteq \{O \cup O <_{\text{wpo}} O, s <_{p}\} \quad (Vh.1)
\]

\[
\forall q \in p, \{O <_{p}\} = \{O <_{q}\} \quad (Vh.2)
\]

So by Vh.1 and Vh.2 it is clear that computation \( C \) satisfies program order, processor order, execution order, synchronization order and weak program order. Hence \( C \) satisfies Weak Consistency.

4.4.2 Verifying Release Consistency (VRC)

**INSTANCES:** \( O \) be the set of all the operations of a computation \( C \) of a system \((P, D)\), \( C \) satisfies a valid total order \((O, <)\) such that \( \{O \cup O <_{\text{wpo}} O, s <_{p}\} \).
TO VERIFY: C is Release consistent

VERIFICATION: Let \((o_1, o_2) \in O\) and \((w_1, w_2) \in W\). It has given that C satisfies valid total order and we know that for

RC\(\forall o_1, o_2, o_3 : O_1 \xrightarrow{so} O_2 \xrightarrow{cb} O_3\) on H and subtype\((O_1) = release\) and subtype\((O_1) = acquire\) and Type\((O_3) \in \{r,w\}\), so it must satisfy the following:-

\[
\left( O_1^p \cup O_1^w \cup O_1^s <_{acq} \right) \vee \left( O_1^p \cup O_1^w \cup O_1^s >_{rel} \right) \tag{Vh.3}
\]

\[
\forall q \in p, (O_1^s < p) = (O_1^s < q) \tag{Vh.4}
\]

So by Vh.3 and Vh.4 it is clear that computation C satisfies program order, processor order, execution order, synchronization order and weak program order. Hence C satisfies Release Consistency.

4.4.3 Verifying Entry Consistency (VEC)

INSTANCES: O be the set of all the operations of a computation C of a system \((P, D)\), C satisfies a valid total order \((O, <)\) such that\(\left( O_1^p \cup O_1^w \cup O_1^s <_{SORC,dgeb} \right)\).

TO VERIFY: C is entry consistent

VERIFICATION: Let \((o_1, o_2) \in O\) and \((w_1, w_2) \in W\). it has given that C satisfies valid total order and as we know that for EC

RC\(\forall o_1, o_2, o_3 : O_1 \xrightarrow{sorc} O_2 \xrightarrow{dgeb} O_3\) on H and subtype\((O_1) = release\) and subtype\((O_2) = acquire\) and Type\((O_3) \in \{r,w\}\), so it must satisfy the following:-
\[ \forall O_1, O_2 \left( \exists \text{Sorb} O_1, O_2 \land \text{Acq} O_2, O_3 \right) \Rightarrow O_1 <_{\text{rel}} O_2 <_{\text{acq}} O_3 \in \{r, w\} \quad (\text{Vh.5}) \]

\[ \forall q \in p, (O |_s <_p) = (O |_s <_q) \quad (\text{Vh.6}) \]

So by Vh.5 and Vh.6 it is clear that computation C satisfies program order, processor order, execution order, synchronization order and weak program order. Hence C satisfies Entry Consistency.

### 4.4.4 Verifying Scope Consistency (VScC)

**INSTANCEs:** \( O \) be the set of all the operations of a computation \( C \) of a system \((P, D)\), \( C \) satisfies a valid total order \((O, <)\) such that \((O |_p \cup O |_w \cup O |_{\text{Sorb}, \text{ScopCB}} <_p)\).

**TO VERIFY:** \( C \) is entry consistent

**VERIFICATION:** Let \((o_1, o_2) \in O\) and \((w_1, w_2) \in W\). it has given that \( C \) satisfies valid total order and as we know that for ScC
\[ \forall O_1, O_2, O_3 : \text{If } O_1 \xrightarrow{\text{Sorb}} O_2 \xrightarrow{\text{ScopCB}} O_3 \text{ on } H \text{ and } \text{subtype}(O_1) = \text{release} \text{ and } \text{subtype}(O_2) = \text{acquire} \text{ and } \text{Type}(O_3) \in \{r, w\}, \text{ so it must satisfy the following:-} \]

\[ \forall O_1, O_2 \left( \exists \text{ScopCB} \right) \Rightarrow O_1 \in \{r, w\} \land O_2 <_{\text{rel}} \quad (\text{Vh.6}) \]

\[ \forall q \in p, (O |_s <_p) = (O |_s <_q) \quad (\text{Vh.7}) \]

So by Vh.6 and Vh.7 it is clear that computation C satisfies program order, processor order, execution order, synchronization order and weak program order. Hence C satisfies Scope Consistency.
REFERENCE


