CHAPTER 3

STRUCTURAL UNIFORM MEMORY MODEL

The uniform memory consistency model considers only read and write operation. Uniform memory consistency models are independent to process’s activity. So in this model the synchronization of process is not required. In this chapter we will discuss about uniform memory models in detail. For this we have designed a structural uniform memory model for all memory consistency models belonging to uniform memory model on unified framework. All consistency models of uniform memory model are described on the basis of this unified framework.

3.1 Structural Uniform Memory Model – Design

The main components of our structural uniform memory model are the consistency models which considers only read & write memory operation to define consistency condition. We identify the characteristics that are inherent to all memory consistency models and the characteristics that are model-specific. The structural uniform memory model is the combination of strong & relaxed memory model which proposes a simple and general definition of memory consistency models. The proposed structural uniform memory model is based on the models which come in to the
category of uniform memory model. The unified framework is categorized by four properties, order of access; concurrency; scope and atomicity. The order of access defines the sequence in which accesses are seen by interested parties. The concurrency of access defines if nodes can concurrently access the data and the modes in which they can access it. The scope determines the set of data that is to be kept consistent and atomicity defines whether the propagation of updates is done on per access basis or whether several local updates can be done before a batched update is sent out.

We have taken Atomic consistency (AC), Sequential consistency (SC), Causal consistency (CC), Processor consistency (PC), PRAM, Cache consistency and Slow memory consistency models for our structural uniform memory model. The first two AC & SC is the strong

![Figure 3.1 Structural Uniform Memory Model](image-url)
consistency whereas the other one are relaxed consistency. Atomic Consistency is the strict consistency among all the models of framework.

In structural uniform memory model shown in fig. 3.1, if we follow the path from Top to Bottom, we find that the sequential consistency is evolved from the atomic consistency so it inherits some property of atomic consistency. The processor consistency and the causal consistency i.e. defined by the sequential consistency. The PRAM model is evolved by combining the processor consistency and causal consistency. The cache consistency is defined by processor consistency and based on cache coherency, so it is also called as coherence consistency. Slow memory model is the combination of PRAM and Cache Consistency. But if follow the path from Bottom to Top, the Sequential Consistency is the combination of Processor and Causal consistency and Processor Consistency is the combination of Pram and Cache consistency.

### 3.2 Defining Uniform Memory Models

As our proposed structural uniform memory model is designed on unified framework, so to describe memory models in unified way, we propose a history-based system model that is related to unified framework. 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 distributed shared global memory $M$. The distributed shared global memory $M$ 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 $M$. A
memory operation $O_{P_i}(x)v$ is executed by processor $P_i$ on memory address $x$ with the value $v$. There are two basic types of operations on $M$: Read ($r$) and Write ($w$). An execution history $H$ of a process $P_i$ is an ordered sequence of memory operation issued by the process $P_i$. Figure 3.2 shows the execution history of processor $P_1$ and $P_2$.

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

**Figure 3.2** Execution History of $P_1$ and $P_2$

A read operation $R_{P_i}(x)v$ 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 $W_{P_i}(x)v$ is in fact a set of memory operations $S = \sum_{i=0}^{n-1} W_{P_i}(x)v$ where $n$ is the number of processors. A write operation $W_{P_i}(x)v$ is performed with respect to processor $P_i$ when the value $v$ is written to the address $x$ on the local memory $M_i$ of $P_i$. A write operation $W_{P_i}(x)v$ is performed when it is performed with respect to all processors that compose the system. An order relation that is used to define all memory consistency models proposed is program order ($\rightarrow_{po}$). An operation $O_1$ is related to an operation $O_2$ by program-order ($O_1 \rightarrow_{po} O_2$) if:

1. Both operations are issued by the same processor $P_i$ and $O_1$ immediately precedes $O_2$ in the code of $P_i$ or
2. $\exists O_3$ such that ($O_1 \rightarrow_{po} O_3$) and ($O_3 \rightarrow_{po} O_2$)
3.2.1 Atomic Consistency (AC)

This is the strictest of all consistency models. With atomic consistency, operations take effect at some point in an operation interval. It is easiest to think of operation intervals as dividing time into non-overlapping and consecutive slots.

Tanenbaum [17] describes atomic consistency as, “any read to a memory location x returns the value stored by the most recent write operation to x”. For example, the clock cycle of a memory bus could serve as an operation interval. Multiple accesses during the same operation interval are allowed, which causes a problem if reads and writes to the same location occur in the same operation interval. One solution is to define read operations to take effect at read-begin time and write operations to take effect at write-end time. This is called static atomic consistency. With dynamic AC, operations can take effect at any point in the operation interval; as long as the resulting history is equivalent to some serial execution [1, 5, 6]. Atomic consistency is often used as a base model when evaluating the performance of an MCM.

**Definition:** On unified framework a history $H$ is *atomically consistent* if there is a legal linear sequence of $H$ that respects the order $\overset{\text{AT}}{\rightarrow}$ which is defined as follows:

1. $\forall O_1, O_2 : \text{if } O_1 \overset{\text{PO}}{\rightarrow} O_2 \text{ then } O_1 \overset{\text{AT}}{\rightarrow} O_2 \text{ and}$
2. $\forall O_1, O_2 : \text{if } gt(\text{performed}(O_1)) < gt(\text{issue}(O_2)) \text{ then } O_1 \overset{\text{AT}}{\rightarrow} O_2$
In above definition relation $\rightarrow^{AT}$ shows the order relation where all processors must perceive the same execution order of all shared memory accesses.

### 3.2.2 Sequential Consistency (SC)

Sequential consistency was first defined by Lamport [10]. He defined a memory system to be sequentially consistent “if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”. In a sequentially consistent system, all processors must agree on the order of observed effects [3, 4, 12, 14]. Figure 3.3 shows a legal execution history for SC:

<table>
<thead>
<tr>
<th>P₁</th>
<th>W(x)₁</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₂</td>
<td></td>
<td>W(y)₂</td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td>R(y)₂</td>
<td>R(x)₀</td>
<td>R(x)₁</td>
</tr>
</tbody>
</table>

**Figure 3.3** Execution history of $P_1$, $P_2$, and $P_3$ for SC

Note that R(y)₂ by processor P₃ reads a value that has not been written yet, of course this is not possible in any real physical system. However, it shows a surprising flexibility of the SC model. This is not a legal history for atomic consistency as the write operations W(x)₁ and W(y)₂ appear commuted at processor P₃. Sequential consistency has been the canonical memory consistency model for a long time.
Definition: On unified framework a history $\mathbf{H}$ is \textit{sequentially consistent} if there is a legal linear sequence of $\mathbf{H}$ that respects the order $\xrightarrow{\text{sc}}$ which is defined as follows:

$$\forall O_1, O_2 : \text{if } O_1 \xrightarrow{\text{po}} O_2 \text{ then } O_1 \xrightarrow{\text{sc}} O_2$$

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

Sufficient Conditions for SC

For maintaining any system sequentially consistent, following condition must be satisfied; these are only the sufficient condition but may not be necessary condition:

- Assumes general cache coherence
  - Writes to the same location are observed in same order by all Processor.
- For each processor, delay issue of access until previous completes
  - A read completes when the return value is back
  - A write completes when new value is visible to all processors
  - For simplicity, we assume writes are atomic
- assume write-back caches
  - general cache coherence maintained by serialization at bus
- writes to same location serialized and observed in the same order by all
• writes are atomic because all processors observe the write at the same time

• accesses from a single process complete in program order:
  
  o cache is busy while servicing a miss, effectively delaying later access

In short we can say that for sequential consistency in a system before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be globally performed and all previous accesses must be performed, and

Before a STORE is allowed to perform with respect to any other processor, all previous LOAD accesses must be globally performed and all previous accesses must be performed.

3.2.3 Causal Consistency (CC)

Causal Consistency means that all processor see all causally related shared access in the same order. Mosberger [6] describes Causal consistency as “A memory is causally consistent if all machines agree on the order of causally related events. Causally unrelated events (concurrent events) can be observed in different orders”. For proper defining we should know about read by order (rb). A write operation \( w(x)v \) is read by a read operation \( r(x)v \) if the operation \( r(x)v \), issued by processor \( p_i \), reads the value written by the operation \( w(x)v \) issued by processor \( p_i \) and \( i \neq j \).
**Definition:** On unified framework a History $H_{pi+w}$ is **causal consistent** if $H$ be the history of writes $w$ with respect to processor $p_i$ and it is a subhistory of $H$ that contains all operations issued by processor $p_i$ and all write operations issued by the other processors.

A memory is causally consistent if all processors agree on the order of causally related events. Causally unrelated events (concurrent events) can be observed in different [6, 11]. Figure 3.4 shows a legal execution history under CC, Note that W(x)1 and W(x)2 are causally related as P2 observed the first write by P1.

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>W(x)1</th>
<th>W(x)3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td></td>
<td>R(x)1</td>
<td>W(x)2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td>R(x)1</td>
<td>R(x)3</td>
<td>R(x)2</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td>R(x)1</td>
<td>R(x)2</td>
<td>R(x)3</td>
</tr>
</tbody>
</table>

**Figure 3.4** Execution history of $P_1$, $P_2$, and $P_3$ for CC

Furthermore, $P_3$ and $P_4$ observe the accesses W(x)2 and W(x)3 in different orders, which would not be legal under SC.

**Sufficient Conditions for CC**

A history $H$ is causally consistent if there is a legal linear sequence of $H_{pi+w}$ that respects the order $\rightarrow_{cc}$ which is defined for each processor $p_i$ follows the following condition:

1. Satisfy the program order (po)

$$\forall O_1, O_2 : if \quad O_1 \xrightarrow{po} O_2 \quad then \quad O_1 \xrightarrow{cc} O_2$$
2. Satisfy the read by order (rb)

\[ \forall O_1, O_2 : \text{if } O_1 \xrightarrow{rb} O_2 \text{ then } O_1 \xrightarrow{cc} O_2 \]

3. Satisfy the transitivity relation

\[ \forall O_1, O_2 : \text{if } O_1 \xrightarrow{cc} O_2 \text{ and } O_2 \xrightarrow{cc} O_3 \text{ then } O_1 \xrightarrow{cc} O_3 \]

3.2.4 Processor Consistency (PC)

Generally all relaxed memory consistency model are concentrating on single copy of memory to the programmer as it is easy implement and there is no requirement of coherency maintenance. Processor consistency is the model where multiple copies of memory with coherence are exposed to the programmer. Processor consistency (PC) model is an intermediate level of consistency which is strong than weak but weak than strong. It guarantees correct behavior and permits substantially higher performance than strong ordering. A DSM system is processor consistent if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program [6, 9, 13].

Conditions for Processor Consistency (PC)

Memory sub-operations must execute in a sequential order that satisfies the following conditions:

a) Sub-operations appear in this sequence in the order specified by the program order requirement as shown in the figure 3.5 and

\[ \forall O_1, O_2 : \text{if } O_1 \xrightarrow{po} O_2 \text{ then } O_1 \xrightarrow{pc} O_2 \]
b) The order among sub-operations satisfies the write-update coherence (WUC) requirement, and

\[ \forall O_1, O_2 : \text{if } O_1 \xrightarrow{PO} O_2 \text{ and } O_1 \xrightarrow{WUC} O_2 \text{ then } O_1 \xrightarrow{PC} O_2 \]

c) A read sub-operation issued by R(i) returns the value of either the last write sub-operation W(i) to the same location that appears before the read in this sequence or the last write sub-operation to the location that is before the read in program order, whichever occurs later in the execution sequence.

\[
\begin{array}{cccc}
\text{Program Order} \\
\hline
R & R & W & W \\
\hline
R & W & R & W \\
\hline
\end{array}
\]

\[+\]

\begin{center}
\textbf{Coherence}
\end{center}

\textbf{Figure 3.5} Program Order Requirement of PC

A processor consistent DSM system with write-update coherence protocol and data replication consists of several processors each with their own copy of the entire memory. By modeling memory as being replicated at every processing node, we can capture the non-atomic effects that arise due to presence of multiple copies of a single memory location. Since the memory no longer behaves as a single logical copy, we need to extend the notion of read and write memory operations to deal with the presence of multiple copies. Write operations no longer appear atomic, however. Each write operation conceptually results in all memory copies corresponding to the location
to be updated to the all processor’s memory. We have taken write-update coherence protocol for updating to all processor’s memory.

### 3.2.5 Pipelined RAM (PRAM)

The acronym PRAM is often used as a shorthand for *Parallel Random Access Machine* which has nothing in common with the Pipelined RAM consistency model. The reasoning that led to this model was as follows: consider a multi-processor, where each processor has a local copy of the shared memory. Mosberger [6] describes “PRAM consistency is consistency in which all processors observe the writes from a single processor in the same order while they may disagree on the writes by different processors.

**Definition:** A history \( H \) is PRAM consistent if there is a legal linear sequence of \( H_{p_i+w} \) that respects the order \( \rightarrow_{\text{PRAM}} \) which is defined for each processor \( p_i \) follows:

\[
\forall O_1, O_2 : \text{if } O_1 \rightarrow_{\text{PO}} O_2 \text{ then } O_1 \rightarrow_{\text{PRAM}} O_2
\]

For memory to be scalable, an access should be independent of time it takes to access the other processors’ memories. On a read, a PRAM would simply return the value stored in the local copy of memory. On a write, it would update the local copy first and broadcast the new value to the other processors. Assuming a constant time for initiating a broadcast operation, the goal of making the cost for a read or write constant is thus achieved. In terms of ordering constraints, this is equivalent to requiring that all processors observe the writes from a single processor in the same order while they may disagree on the
order of writes by different processors [8, 11]. Figure 3.6 shows the execution history for PRAM consistency:

<table>
<thead>
<tr>
<th></th>
<th>W(x)1</th>
<th></th>
<th>W(x)3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>R(x)1</td>
<td></td>
<td>W(x)2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td>R(x)1</td>
<td></td>
<td>R(x)2</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td>R(x)2</td>
<td></td>
<td>R(x)1</td>
</tr>
</tbody>
</table>

Figure 3.6 Execution history of P₁, P₂, P₃, and P₄ for PRAM

P₃ and P₄ observe the writes by P₁ and P₂ in different orders, although W(x)₁ and W(x)₂ are potentially causally related.

### 3.2.6 Cache Consistency (CcC)

Cache consistency and coherence are synonymous. Coherence is a location-relative weakening of SC. Recall that under SC, all processors have to agree on some sequential order of execution for all accesses. Coherence only requires that accesses are sequentially consistent on a per-location basis. Clearly, SC implies coherence but not vice versa. Thus, coherence is strictly weaker than SC [2, 9, 11].

**Definition:** Goodman [7] states that Cache Consistency “only guarantees that accesses to a given memory location are strongly ordered”. Mosberger [6] indicates that “Cache Consistency only requires that accesses are SC on a per-location basis”.

On unified framework let O be all the operations of a computation C of a DSM system. Then C is Coherent if for satisfying
each object $x$ there is some linearization $(O \mid x, \xrightarrow{ls})$ satisfying $(O \mid x, \xrightarrow{po}) \subseteq (O \mid x, \xrightarrow{ls})$. The example below is a history that is coherent but not sequentially consistent:

<table>
<thead>
<tr>
<th></th>
<th>$W(x)1$</th>
<th>$R(y)0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P2$</td>
<td>$W(y)1$</td>
<td>$R(x)0$</td>
</tr>
</tbody>
</table>

**Figure 3.7** Execution history of $P_1$, $P_2$, and $P_3$ for Coherence

Clearly, any serial execution that respects program order starts with writing 1 into either $x$ or $y$. It is therefore impossible that both read accesses return 0. However, the accesses to $x$ can be linearized into $R(x)0$, $W(x)1$ and so can the accesses to $y$: $R(y)0$, $W(y)1$. The history is therefore coherent, but not SC. In essence, coherence removes the ordering constraints that program order imposes on accesses to different memory locations.

### 3.2.7 Slow Memory Consistency

Slow memory is a location relative weakening of PRAM. It requires that all processors agree on the order of observed writes to each location by a single processor. Furthermore, local writes must be visible immediately (as in the PRAM model). The name for this model was chosen because writes propagate slowly through the system. Slow memory is probably one of the weakest uniform memory consistency models that can still be used for intercrosses communication [6, 11]. However, this algorithm guarantees physical exclusion only. There is no guarantee of logical exclusion.
**Definition:** On unified framework a history \( H \) is Slow consistent if there is a legal linear sequence of \( H_{pi+w} \) that respects the order \( \overrightarrow{SL} \) which is defined for each processor \( p_i \) follows:

1. All processors must agree about the processor write order on the same memory location
   \[
   \forall O_1, O_2 : \text{if } \text{processor}(O_1) = \text{processor}(O_2) = p_i \text{ and } O_1 \xrightarrow{PO} O_2 \text{ then } O_1 \xrightarrow{SL} O_2
   \]

2. All processors must eventually see all write operations issued by all processors since the order is defined on
   \[
   \forall O_1, O_2 : \text{if } \text{processor}(O_1) = \text{processor}(O_2) = p_i \text{ and address}(O_1) = \text{address}(O_2) \\
   \text{and } O_1 \xrightarrow{PO} O_2 \text{ then } O_1 \xrightarrow{SL} O_2
   \]

For example, after two processes P1 and P2 were subsequently granted access to a critical section and both wrote two variables a and b, then a third process P3 may enter the critical region and read the value of a as written by P1 and the value of b as written by P2. Thus, for P3 it looks like P1 and P2 had had simultaneous access to the critical section. This problem is inherent to slow memory because the knowledge that an access to one location has performed cannot be used to infer that accesses to other locations have also performed. Slow memory does not appear to be of any practical significance.

### 3.3 Relating the Uniform Memory Models

Defining memory consistency models formally makes it easier to compare and relate them. The following figure shows the relationship among the different memory models of structural uniform memory model.
Each rectangle shows the possible results that can be produced under the unified framework. As the figure shows that the strongest model is Atomic consistency model because it imposes a total order on execution and requires real time order to be preserved. Sequential consistency imposes only a total order on execution [14, 16]. So atomic consistency is strictly stronger than the sequential consistency. Causal consistency is strictly stronger than the PRAM consistency. Both memory models require that program order on running process. Causal model requires also that all processors must respect the read by order. PRAM consistency is strictly stronger than Slow memory since slow memory only requires programmers order.

3.3.1 SC versus CcC

Sequential consistency requires strong ordering; all processor to see all writes to all location in same order and all operation in a program must be in same order. Whereas Coherence requires serialization of writes in
same order and writes should be visible to all processors. Let \( O \) be the 
set of all the operations of a computation \( C \) of a system \((P, D)\), then \( C \) 
satisfies SC if there is a valid total order \((O, \prec)\) such that 
\( (O, \prec_{po}) \subseteq (O, \prec) \). e.g.

i. **Computation** \( C_1 \):

\[
\begin{align*}
  p : w(x) & 1 \quad r(x) & 2 \\
  q : r(x) & 1 \quad w(x) & 2
\end{align*}
\]

satisfies sequential consistency (SC) as it satisfies total ordering 
\((O, \prec) = \prec w_p(x) l, r_q(x) l, w_q(x) 2, r_p(x) 2 \rangle\) as well as the program 
ordering \((O, \prec_{po}) = \prec w_p(x) l, r_p(x) 2, r_q(x) l, w_q(x) 2 \rangle\).

ii. **Computation** \( C_2 \):

\[
\begin{align*}
  p : w(x) & 1 \quad r(x) & 2 \\
  q : w(x) & 2 \quad r(x) & 1
\end{align*}
\]

does not satisfy (SC) as it violates 
the program order (PO) and the total ordering \((O, \prec)\) is not valid.

Coherence is a location-relative weakening of SC. The 
computation \( C \) for the set of operations \( O \) and system \((P, D)\) satisfies 
coherence if

1. \( C \) satisfies SC and

2. For each \( x \in D \) there is valid total order \((O \mid x, \prec_x)\) such 
   that \((O \mid x, \prec_{po} x) \subseteq (O \mid x, \prec_x)\).

i. **Computation** \( C_1 \):

\[
\begin{align*}
  p : w(x) & 1 \quad r(x) & 2 \\
  q : r(x) & 1 \quad w(x) & 2
\end{align*}
\]

satisfies coherency as for all \( x \) in \( D \) it satisfies 
\((O \mid x, \prec_x) = \prec w_p(x) l, r_q(x) l, w_q(x) 2, r_p(x) 2 \rangle\) and it also satisfy 
SC also.

ii. **Computation** \( C_2 \):

\[
\begin{align*}
  p : w(x) & 1 \quad r(x) & 2 \\
  q : w(x) & 2 \quad r(x) & 1
\end{align*}
\]

does not satisfy coherence.
iii. Computation \( C_3 \) satisfies coherence but not SC.

So by the computation \( C_1, C_2 \) and \( C_3 \) we can say that

a. If a computation \( C \) satisfies SC then it satisfies coherence and

b. If a computation satisfies coherence then it does not necessarily satisfy sequential consistency.

All computation is satisfying consistency model CM = C(CM). Figure 3.9 shows the relationship between sequential consistency and cache consistency where \( C_1 \) satisfies SC as well as coherence and \( C_3 \) satisfies only SC but not coherence.

Figure 3.9 SC versus Coherence consistency model

3.3.2 SC versus PRAM

Let \( O \) be the set of all the operations of a computation \( C \) of a system \( (P,D) \), then \( C \) satisfies PRAM if for each \( p \in P \) there is a valid total order \( \{O \cup O \mid w, <_p\} \) such that \( \{O \cup O \mid w, <_{p_0}\} \leq \{O \cup O \mid w, <_p\} \), e.g.
i. Computation $C_1$ satisfies PRAM consistency as

\[
\begin{align*}
\{(O \cup O \mid w, <_p) = &< w_p(x), w_q(x), r_p(x) > \text{ and} \\
(O \cup O \mid w, < q) = &< w_p(x), r_q(x), w_q(x) > \}
\end{align*}
\]

it also satisfies SC as well as Coherence consistency.

ii. Computation $C_3$ does not satisfy PRAM as well as SC.

iii. Computation $C_4$ satisfies PRAM but not SC.

So by the computation $C_1$, $C_3$ and $C_4$ we can say that

a. If a computation $C$ satisfies SC then it satisfies PRAM and

b. If a computation satisfies PRAM then it does not necessarily satisfy sequential consistency.

Figure 3.10 shows the relationship between sequential consistency and PRAM consistency where $C_1$ satisfies SC as well as PRAM consistency and $C_4$ satisfies only PRAM but not SC.

![Figure 3.10 SC versus PRAM consistency model](image-url)
3.3.3 SC versus CC

Let define \textit{write-before-read} \((O, <_{wbr})\) by \(O_1 <_{wbr} O_2\) if \(O_1\) is \(w(x)v\) and \(O_2\) \(r(x)v\) for some \(x\) and \(v\). So if \(O\) be the set of all the operations of a computation \(C\) of a system \((P, D)\), then \(C\) satisfies \textbf{Causal Consistency} if for each \(p \in P\) there is valid total order \((O | p \cup O | w, <_p)\) such that \((O | p \cup O | w, <_{co}) \subseteq (O | p \cup O | w, <_p)\) e.g.

i. Computation \(C_1\) \[
\begin{array}{ll}
p : w(x) & 1 \\
q : r(x) & 2
\end{array}
\]
satisfies SC, coherence, PRAM as well as CC.

ii. Computation \(C_2\) \[
\begin{array}{ll}
p : w(x) & 1 \\
q : w(x) & 2 \\
r : r(x) & 2
\end{array}
\]
satisfies CC.

iii. Computation \(C_5\) \[
\begin{array}{ll}
p : w(x) & 3 \\
w : w(x) & 1 \\
w : w(y) & 2 \\
r : r(y) & 2 \\
r : r(y) & 3
\end{array}
\]
satisfies CC but not SC.

So by the computation \(C_1, C_2\) and \(C_5\) we can say that

a. If a computation \(C\) satisfies \textbf{SC} then it satisfies \textbf{CC} and

b. If a computation satisfies \textbf{CC} then it does not necessarily satisfy \textbf{sequential consistency}.

Figure 3.11 shows the relationship between sequential consistency and Causal consistency where \(C_1\) satisfies SC as well as CC consistency \(C_2\) satisfies causal consistency CC and \(C_5\) satisfies only CC but not SC.
3.4 Verification of Uniform Memory Models

In DSM systems with store atomicity, the lifetime of an operation is limited. Two operations with non-overlapping lifetimes should be ordered. To carry out verification, we first introduce the terminology and notations. A system is a multiprocessor system of \( p \) processors that access a shared memory. Each processor can issue operations to access the memory through a series of read and write operations on the memory or to synchronize. \( O \) is the set of operations issued by all processors. The temporary internal structure of a system is the internal structure excluding cache and memory. Every operation enters the temporary internal structure of a processor in program order and is finally globally viewed by all processors. We use \( O \) with subscript to represent a memory operation, \( r \) (with subscript) to represent a read operation, \( w \) (with subscript) to represent a write operation. WE denote the problem of memory consistency verification under time order restriction as \( VMC\text{-time} \). WE refer to VMC-time with additional read mapping information as VMC-time-read.
Operation orders are the basis of memory consistency verification. First let us come to the definitions of program order, processor order and execution order, which are three well known types of partial orders in uniform memory model of DSM systems.

**Definition: 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 program order iff \( O_1 \) is before \( O_2 \) in the program. i.e.

\[
O_1 \xleftarrow{\text{po}} O_2
\]

**Definition: Processor Order:**
Given two different operations \( O_1 \) and \( O_2 \) in the same processor, we say that \( O_1 \) is before \( O_2 \) in processor order iff there is global agreement that \( O_1 \) is before \( O_2 \) for all processors.

\[
O_1 \xleftarrow{\text{PrO}} O_2
\]

Many DSM systems have cache subsystems, thus they maintain cache consistency, which can be defined as coherence order: all write operations to the same location are performed in some sequential order. When a DSM system is required to be coherent, it makes sense to consider both write-before order and read-before order, which are called execution orders.

**Definition: Execution Order:** We say that a write operation \( w \) is before operation \( O_1 \) in execution order iff \( w \) is the latest write operation before \( O_1 \) that accesses the same location as \( O_1 \). i.e.

\[
w \xleftarrow{E} O_1
\]
We say that a write operation \( w \) is after operation \( O_1 \) in execution order iff \( w \) is the first write operation after \( O_1 \) that accesses the same location as \( O_1 \).

\[ O_1 \rightarrow w \]

In addition, global order is another well-known partial order based on processor order and execution order. It is the transitive closure of processor order and execution order.

**Definition: Global Order:**
We say that operation \( O_1 \) is before operation \( O_2 \) in global order iff \( O_1 \) is before \( O_2 \) in processor order, or \( O_1 \) is before \( O_2 \) in execution order, or \( O_1 \) is before some operation \( O \) in global order and \( O \) is before \( O_2 \) in global order.

\[
(O_1 \xrightarrow{GO} O_2) \rightarrow [(O_1 \xrightarrow{PO} O_2) \lor (O_1 \xrightarrow{E} O_2) \lor \exists O_i \in O : O_i \xrightarrow{GO} O \xrightarrow{GO} O_2]
\]

Most memory consistency models consist of different rules for processor order. When an operation \( O_1 \) is before \( O_2 \) in program order, this does not always mean that \( O_1 \) is before \( O_2 \) in processor order.

### 3.4.1 Verifying Sequential Consistency (VSC)

**INSTANCES:** \( O \) be the set of all the operations of a computation \( C \) of a system \((P, D)\), \( C \) satisfies a valid total order \((O, \prec)\) such that \((O, \prec_{PO}) \subseteq (O, \prec)\).

**TO VERIFY:** \( C \) is sequentially 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:-
By V.1 and V.2 it is clear that computation C satisfies program order, processor order, execution order and global order. Hence C satisfies Sequential Consistency.

### Verifying Causal Consistency (VCC)

**Instances:** O be the set of all the operations of a computation C, C satisfies for each \( p \in P \) a valid total order \((O \cup P, \leq)\).

**To verify:** C is causally 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 for CC it satisfies write-before-read \((O, \leq_{wbr})\) by \(O_1 \leq_{wbr} O_2\) if \(O_1\) is \(w(x)\) \(v\) and \(O_2\) \(r(x)\) \(v\) for some \(x\) and \(v\), so it must satisfy the following:-

\[
\forall o_1, o_2 \in O \cup OW \left( \exists_{j=1..n} o_1 \rightarrow_{j} o_2 \Rightarrow o_1 \rightarrow_{i} o_2 \right) \quad (V.3)
\]

So by V.3 all machines agree on the order of causally related events. Hence C satisfies Causal Consistency.
3.4.3 Verifying Processor Consistency (VPC)

**INSTANCES:** \( O \) be the set of all the operations of a computation \( C \), \( C \) satisfies for each \( p \in P \) a valid total order \( (O | p \cup O | w, <_p) \) with write update coherence protocol.

**TO VERIFY:** \( C \) is processor 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 with write update coherence protocol and for PC it satisfies \( 2O \cup W \subseteq O \) and \( O \cap W \subseteq O \), so it must satisfy the following:

\[
\forall w_1, w_2 \in O \cap W \subseteq O \exists i \in \{1..n\} \quad \forall w_1 \mapsto_i w_2 \lor \forall w_2 \mapsto_i w_1 \quad (V.4)
\]

\[
\forall w_1, w_2 \in O \cap W \subseteq O \exists i \in \{1..n\} \quad \forall w_1 \mapsto_i w_2 \lor \forall w_2 \mapsto_i w_1 \quad (V.5)
\]

So V.4 and V.5 satisfies operations of each individual processor appear in the sequential order specified by its program. Hence \( C \) satisfies Processor Consistency.

3.4.4 Verifying Cache Consistency (VCC)

**INSTANCES:** \( O \) be the set of all the operations of a computation \( C \), \( C \) satisfies for \( x \in D \) there is valid total order \( (O | x, <_x) \) such that \((O | x, <_x) \subseteq (O | x, <_x)\).

**TO VERIFY:** \( C \) is cache 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 for cache consistency it satisfies \(O | x, \xrightarrow{PO} \subseteq O | x, \xrightarrow{LxPO}\), so it must satisfy the following:

\[
\forall x \in X \forall w_1, w_2 \in OW \cap O | x \left( \forall i=1..n \right) \left( w_1 \xrightarrow{i} w_2 \lor w_2 \xrightarrow{i} w_1 \right) \quad (V.6)
\]

So by V.6 satisfies that accesses are SC on a per-location basis. Hence \(C\) is Cache Consistent.
REFERENCE


