PUBLICATION INTERNE
N° 871

SEQUENTIAL CONSISTENCY IN DISTRIBUTED SYSTEMS : THEORY AND IMPLEMENTATION

M. MIZUNO, M. RAYNAL, J.Z. ZHOU

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

Sequential Consistency in Distributed Systems : Theory and Implementation

M. Mizuno*, M. Raynal**, J.Z. Zhou***

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Publication interne n° 871 — Octobre 1994 — 41 pages

Abstract: Recently, distributed shared memory (DSM) systems have received much attention because such an abstraction simplifies programming. It has been shown that many practical applications using DSMs require competing operations. We have aimed at unifying theory and implementations of protocols for sequential consistency, which provides competing operations. The results are useful not only to clarify previously proposed implementations but also to develop new efficient implementations for consistency criteria which provide competing operations, such as sequential consistency, weak ordering (with sequential consistency for competing accesses), and release consistency (with sequential consistency for competing accesses). By adopting concepts from concurrency control, we have developed theory for sequential consistency, called a *sequentializability theory.* This paper first presents the sequentializability theory, and then demonstrates the correctness of existing protocols using the theory. Finally, the paper presents new protocols which require significantly less communication than previously proposed protocols in systems which do not provide hardware atomic broadcasting facilities.

Key-words: distributed virtual memory, sequential consistency.

*Dept. of Computing and Info. Sciences, Kansas State University, Manhattan, KS 66506, USA `masaaki@cis.ksu.edu`. This work was supported in part by the National Science Foundation under Grant CCR-9201645 and INT-9406785.

**IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE `raynal@irisa.fr`. This work was supported in part by the ESPRIT project BROADCAST (Number 6360) of the Commission of European Communities.

***Dept. of Computing and Info. Sciences, Kansas State University, Manhattan, KS 66506.

INRIA

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

(Résumé : tsvp)

La cohérence séquentielle : théorie et implémentation

Résumé : Le concept de mémoire virtuelle fournit une abstraction qui facilite la programmation des applications ; il est en conséquence, de plus en plus utilisé dans le contexte des systèmes répartis. Ce rapport étudie le critère de cohérence, appelé cohérence séquentielle, pour les mémoires partagées. Il en propose une théorie et étudie son impact pour mettre en oeuvre des mémoires virtuelles distribuées. La théorie fournie permet d'expliquer des protocoles existants et d'en concevoir de nouveaux. Les deux nouveaux protocoles présentés sont particulièrement performants en ce qui concerne les communications.

Mots-clé : cohérence séquentielle, mémoire distribuée partagée.

1 Introduction

In distributed and parallel computing environments, distributed shared memory (DSM) systems have received much attention. In general, operations to a DSM are classified into two types: completing and non-competing (or synchronization and non-synchronization) [1, 9, 10, 13]. It has been shown that many practical applications require competing operations [4]. One of the widely used correctness criteria is sequential consistency [15] that provides only competing operations.

In the last decade, concurrency control theory has been extensively studied in the field of transaction based database systems[6, 14, 16]. Even though there is strong similarity between concurrency control theory and correctness criteria of DSM systems, much of the research in DSMs has been done independently. We have developed a theory for sequential consistency (called *sequentializability theory*) that has deep practical implications in implementations of competing operations in DSMs in very much the same way that concurrency control theory does in implementations of many concurrency control protocols. We have aimed at unifying theory and implementations of consistent DSMs: the results are useful not only to clarify previously proposed implementations but also to develop new efficient implementations for consistency criteria which provide competing operations, such as sequential consistency, weak ordering (with sequential consistency for competing accesses) [1, 9], and release consistency (with sequential consistency for competing accesses) [13].

An execution on a DSM is formalized by an irreflexive partially ordered set of read and write operations. Orders on operations are established by a program order at each processor (called "processor-order relation") and a relation between a read and a write operations such that the read operation reads the value written by the write operation (called "reads-from relation"). Following a "serializable execution" in concurrency control theory, we say that an execution on a DSM is *sequentializable* if its result is the same as if the operations of all the processors had been executed in some sequential order (we call this an "illusory sequential execution") which satisfies processor-order relation and is legal [Note: An execution is said to be *legal* if each read operation reads the value written by the immediately preceding write operation on the same object]. A DSM system is sequentially consistent if all the executions on the DSM system are sequentializable.

It has been shown that determining whether a given execution is sequentializable is an NP-complete problem [12, 20]. Thus, as do many concurrency control protocols [6, 14, 16], actual implementations enforce various constraints. We have identified several such constraints, including the following two:

1. *Write-write (WW)-constraint*: the system totally orders all write operations on all objects (we call this totally ordered set of write operations the "ww-list"). This total order constitutes the order of write operations in an equivalent illusory sequential execution.

2. *Object-ordered (OO)-constraint*: the system orders any pair of (write, write), (write, read), and (read, write) operations on each object. These orders are compatible with those in an equivalent illusory sequential execution.

Other constraints are also defined similarly to concurrency control theory, including write-read-write (WRW)-constraint [14].

We show the following:

1. A DSM system is sequentially consistent if each of its executions is under the WW-constraint and legal.

2. A DSM system is sequentially consistent if each of its executions is under the OO-constraint and legal.

If the first approach (the WW-constraint) is used, all the write operations must be globally synchronized to establish the ww-list. Under this constraint, the system establishes reads-from relation by associating each read operation to a write operation such that legality and the processor-order relation are satisfied. If each processor recognizes (some prefix of) the ww-list, read operations may be processed locally at each processor. One way to achieve this is to provide each processor with local processor memory which stores copies of the DSM objects. The protocols presented in [2, 5, 7] are classified in this category. In these protocols, the WW-constraint is achieved by atomic broadcasting of write operations. Legality is guaranteed by updating (or invalidating) processor memory at certain times based on broadcast write messages.

If the second approach (the OO-constraint) is used, operations need to be synchronized only at each object level. However, not only each write operation but also each read operation must be synchronized with other write operations on the same object. A protocol presented in [19] is classified in this category. In this protocol, each write and read operation is sent to the associated object memory module. Thus, the OO-constraint and legality are easily enforced.

Based on the sequentializability theory, we have developed two efficient protocols; one based on the WW-constraint and another on the OO-constraint. The protocols are aimed at systems which do not provide hardware atomic broadcasting facilities. Under this assumption, we intended to minimize the number of messages. In order to increase performance, both protocols utilize local processor memory. We separate issues according to synchronization (to enforce the WW- or OO-constrain) and legality. Synchronization may be enforced by centralized arbitrators, distributed mutual exclusion, or other methods. To enforce legality, the system maintains a special data structure. Based on the information in the data structure, the system invalidates (or updates) certain objects in processor memory. This data structure is very general and may be applicable to many other protocols to implement competing operations.

The paper is organized as follows: In section 2, we introduce some fundamental definitions of distributed shared memory systems, including sequentializability and sequential consistency. Section 3 defines sequentializability under the WW- and OO-constraints and proves some theorems. In Section 4, we apply the theory presented in Section 3 to real implementations. The section also demonstrates the correctness of previous protocols using the sequentializability theory. Section 5 presents two new protocols based on the WW- and OO-constraints.

2 Sequentializability Theory

This section introduces sequentializability theory that will help to deepen our understanding of sequential consistency in DSM systems.

2.1 Definitions

- A *distributed shared memory system* is a pair (P, M), where P is a set of N processors $\{P_1, \cdots, P_N\}$, and M is a shared memory.

  Each processor P_i sequentially executes read and write operations on data items (also called "objects") in M in the order defined by the program running on it.

  A write operation by processor P_i on a data item x is denoted by $w_i(x)v$, where v is the value written on x by this operation. A read operation on x by P_i is denoted by $r_i(x)u$, where u is the value of x returned by this operation. For simplicity, we assume that all values written by write operations are distinct. The parameters of an operation may be omitted when they are not important.

  If $r_j(x)$ reads a value in object x which was written by $w_i(x)$ (that is, $r_j(x)v$ and $w_i(x)v$ for some value v are operations on the shared memory system), then we say "$r_j(x)$ reads x from $w_i(x)$" or "P_j reads x from P_i."

We assume that for each object, an imaginary write operation is performed to initialize the object before the first read operation by any processor is executed on the object.

- An *execution of processor* P_i is a pair $(\Sigma_i, <_{P_i})$, where $\Sigma_i \subseteq \{r_i(x)v, w_i(x)v | x$ is a data item $\in M$ and v is a value which can be stored in $x\}$ and $<_{P_i}$ is a total order on Σ_i, which is specified by the program running on P_i.

- A *complete execution history* (or simply *complete history*) on a distributed shared memory system (P, M) is a pair $\overrightarrow{H} = (H, <_H)$, where $H = \cup_{i=1}^{N} \Sigma_i$ and $<_H$ is a well-formed[1] irreflexive transitive relation on H that includes the relation defined by the following:

  1. If $op_1 <_{P_i} op_2$, then $op_1 <_H op_2$, for $1 \leq i \leq N$. We call this relation *processor-order relation* (of P_i) and sometimes explicitly denote the relation by $op_i <_{pr} op_j$.
  2. If $r_j(x)v$ reads from $w_i(x)v$ in H, then $w_i(x)v <_H r_j(x)v$. We call it *reads-from relation* and sometimes explicitly denote this relation by $w_i(x) <_{rf} r_j(x)$.

  Note that irreflexive and transitive relation implies antisymmetric relation. Thus, this relation is equivalent to an irreflexive directed acyclic graph (or an irreflexive partially ordered set) [3].

  An *execution history* (or *history*) is a prefix of a complete history.

  Throughout the paper, for a relation with transitivity $<$, we use $op_i <^* op_j$ to denote relation either $op_i = op_j$ or $op_i < op_j$.

- A history $\overrightarrow{H} = (H, <_H)$ is a *sequential history* if $<_H$ is a total order.

- A history $\overrightarrow{H} = (H, <_H)$ is *legal* if for every read operation $r_j(x)v$ in H, there exists a write operation $w_i(x)v$ such that $r_j(x)v$ reads from $w_i(x)v$ and there does not exist another write operation $w_k(x)u$ such that $w_i(x)v <_H w_k(x)u <_H r_j(x)v$.

  We also apply term "legal" to read operations; that is, we may say read operation $r_j(x)v$ is legal if $r_j(x)v$ reads from $w_i(x)v$ and there does not exist another write operation $w_k(x)u$ such that $w_i(x)v <_H w_k(x)u <_H r_j(x)v$.

[1] The well-formed property is that any element in the set has only finitely many predecessors.

2.2 Sequential consistency

Sequential consistency was proposed by Lamport to formulate a correctness criterion for a multiprocessor shared-memory system [15]. A multiprocessor system is sequentially consistent if *the result of any execution is the same as if (1) the operations of all the processors were executed in some sequential order, and (2) the operations of each individual processor appear in this sequence in the order specified by its program.*

The system guarantees that operations issued by all processors that access shared memory interleave in a certain manner so that the resulting execution is as if the operations are executed in some legal sequential order. As introduced in Section 1, we call this legal sequential order of operations an *illusory sequential history.* Lamport assumes that the processor-order relation is maintained in an illusory sequential history, but not necessarily in an execution itself. In contrast, we view a distributed execution to be collection of sequential execution histories of individual processes running on distributed processors that interleave through memory access operations. Thus, in our definition, we assume that the processor-order relation is maintained in an execution history.

In order to formally define sequential consistency, we first define the concept of *sequentializable* histories.

Definition : Two histories $\vec{H} = (H, <_H)$ and $\vec{H'} = (H', <_{H'})$ are **equivalent** (or **view-equivalent**), denoted $\vec{H} \equiv \vec{H'}$, if

1. they are over the same set of processors P and the same operations ($H = H'$),
2. they have the same processor-order relation, and
3. they have the same reads-from relation.

Definition : A history $\vec{H} = (H, <_H)$ is **sequentializable** if $\vec{H}$ is equivalent to some legal sequential history.

We now formally define sequential consistency.
Definition SC : A distributed shared memory system (P, M) is **sequentially consistent** if for each of its execution histories $\vec{H}$, $\vec{H}$ is sequentializable.

Theorem 1 : A history $\vec{H}$ is sequentializable if and only if $\vec{H}$ has a topological sort $\vec{S}$ such that $\vec{S}$ is legal.

Proof :
if part : Suppose that $\vec{H}$ has a topological sort $\vec{S}$ which represents a legal history. Then, $\vec{S}$ and $\vec{H}$ (1) are over the same set of processors and operations, (2) have the same processor-order relation, and (3) have the same reads-from relation. Thus, $\vec{H}$ is equivalent to legal sequential history $\vec{S}$, and, therefore, is sequentializable.

only if part : Assume that $\vec{H}$ is sequentializable. Then, we have a legal sequential history, say $\vec{S}$, such that (1) $H = S$, (2) $\vec{H}$ and $\vec{S}$ have the same processor-order relation and the same reads-from relation. Thus, $op_i <_H op_j$ implies $op_i <_S op_j$. Therefore, $\vec{S}$ is a topological sort of $\vec{H}$. $\square$

3 Sequentializability with Constraints

It has been shown that determining whether or not a given execution is sequentializable is an NP-complete problem [12, 20]. Therefore, we impose constraints on each history to ensure efficient implementations of sequentially consistent DSMs. This approach is similar to imposing constraints on view serializability for concurrency control protocols[6, 14]. Intuitively, the constraints enforced by a system include additional ordering in a history $(H, <_H)$ such that it is always possible to sequentialize $\vec{H}$. Identifying such constraints is important to obtain efficient implementations. After presenting three such constraints, we study two of them in detail.

3.1 Definitions of Constraints

Definition : Let $\vec{H} = (H, <_H)$ be a history.

- *Total-order (TO)-constraint :* the system totally orders H; that is, for any two operations op_1, op_2 in $\vec{H}$, either $op_1 <_H op_2$ or $op_2 <_H op_1$.

- *Write-write (WW)-constraint*: the system totally orders all write operations on all objects.

  Let $\vec{H}$ be a history under the WW-constraint. Let $w_i(x)$ and $w_j(y)$ be any two write operations in $\vec{H}$. Then, either $w_i(x) <_H w_j(y)$ or $w_j(y) <_H w_i(x)$. Sometimes, we use $<_{ww}$ to explicitly denote this relation and call $<_{ww}$ a *ww-relation.* We call the total order of the write operations the "ww-list."

If $w_i(x) <_{ww} w_j(y)$, then $w_i(x) <_S w_j(y)$ in an equivalent illusory sequential execution $\vec{S}$.

- *Object-ordered (OO)-constraint*: the system orders any pair of (write, write), (write, read), and (read, write) operations on each object.

  Let $\vec{H}$ be a history under the OO-constraint. Let $op_i(x)$ and $op_j(x)$ be two operations on the same object in $\vec{H}$ where at least one of them is a write operation. Then, $op_i(x) <_H op_j(x)$ or $op_j(x) <_H op_i(x)$. Sometimes, we use $<_{oo}$ to explicitly denote this relation and call $<_{oo}$ an *oo-relation.*

  If $op_i(x) <_{oo} op_j(x)$, then $op_i(x) <_S op_j(x)$ in an equivalent illusory sequential execution $\vec{S}$. □

The TO-constraint is a strong constraint because it forces each execution to be sequential, and may be appropriate for systems which have the centralized shared memory and do not have local processor memory. In systems with processor memory, however, efficient implementations may be difficult to develop based on the TO-constraint and less restrictive constraints such as the WW- and OO-constraints are desirable.

Other constraints may also be defined similarly to concurrency control theory, including write-read-write (WRW)-constraint [14]. In this paper, we discuss the WW- and OO-constraints in detail. We first introduce a sequentialization graph of a history $\vec{H}$.

3.2 Sequentialization graph

This subsection introduces a sequentialization graph for a history under the WW- or OO-constraint. Then, we prove that a history under the WW- or OO-constraint is sequentializable iff its sequentialization graph is acyclic. Testing acyclicity of a sequentialization graph is done in polynomial time.

Definition : Let $\vec{H} = (H, <_H)$ be a history under the WW- or OO-constraint. The Sequentialization Graph of $\vec{H}$, denoted SG($\vec{H}$) $= (N, E)$, is a directed graph where $N = H$. Edges in E are created only by the following two rules:

1. If $op_i <_H op_j$ in $\vec{H}$, then $op_i \rightarrow op_j$ is in E.

   This type of edges captures processor-order, reads-from, and ww- or oo-relation.

2. $r_i(x)v$ is a read operation which reads x from a write operation $w_k(x)v$ in $\vec{H}$. $w_j(x)u$ is another write operation on x, and $w_k(x)v <_H w_j(x)u$ in $\vec{H}$. Then, $r_i(x)v \rightarrow w_j(x)u$ is in E.

   We call this type of edges **exclusive edges**. We also say an edge is created by the **exclusive rule**. Sometimes, we use $\rightarrow_{ex}$ to explicitly denote an exclusive edge. □

Graph SG($\vec{H}$) is used to determine whether $\vec{H}$ is sequentializable; if SG($\vec{H}$) is acyclic, a topological sort of the graph represents a legal sequential history equivalent to $\vec{H}$. Exclusive edges are added to guarantee legality when applying a topological sort. In the above situation, by the exclusive edge, $w_j(x)u$ is forced to be placed either before $w_k(x)v$ or after $r_i(x)v$, but not in between $w_k(x)v$ and $r_i(x)v$, in an equivalent sequential history.

Theorem 2 : Let $\vec{H} = (H, <_H)$ be a history under the WW- or OO-constraint. Then, $\vec{H}$ is sequentializable if and only if SG($\vec{H}$) is acyclic.
Proof :
if part : Suppose $\vec{H}$ be a history such that SG($\vec{H}$) is acyclic. Let $\vec{S}$ be a topological sort of SG($\vec{H}$). Then,

1. $\vec{H}$ and $\vec{S}$ are over the same set of processors and the same operations;

2. $\vec{H}$ and $\vec{S}$ have the same processor-order relation and the same reads-from relation.

Now, we show that $\vec{S}$ is legal. Assume, on the contrary, $\vec{S}$ is not legal. Then, there must exist a read operation $r_i(x)v$ such that $w_j(x)v <_S w_k(x)u <_S r_i(x)v$. Since $\vec{H}$ is under the WW or OO-constraint, either $w_j(x)v <_H w_k(x)u$ or $w_k(x)u <_H w_j(x)v$ holds. If $w_k(x)u <_H w_j(x)v$ holds, we have $w_k(x)u \rightarrow w_j(x)v$ in SG($\vec{H}$). This contradicts the assumption that $w_j(x)v <_S w_k(x)u$ since $\vec{S}$ is a topological sort of SG($\vec{H}$). Therefore, $w_j(x)v <_H w_k(x)u$ must hold. Then, since $r_i(x)v$ reads x from $w_j(x)v$, we have $r_i(x)v \rightarrow_{ex} w_k(x)u$ in SG($\vec{H}$), which contradicts the assumption that $w_k(x)u <_S r_i(x)v$. Therefore, $\vec{S}$ is legal.

only if part : Assume that $\vec{H}$ is sequentializable. Then, there exists a legal sequential history $\vec{S}$ which is equivalent to $\vec{H}$. To show that SG($\vec{H}$) is acyclic, we use the following claim:

Claim : For any two operations op_1 and op_2, if $op_1 \rightarrow op_2$ is in SG($\overrightarrow{H}$), then $op_1 <_S op_2$ holds .

Proof of Claim :

There are two cases to consider :

- *case 1 :* $op_1 \rightarrow op_2$ in SG($\overrightarrow{H}$) by $op_1 <_H op_2$. In this case, $op_1 <_S op_2$ holds since $\overrightarrow{S}$ is equivalent to $\overrightarrow{H}$.

- *case 2 :* $op_1 \rightarrow op_2$ is in SG($\overrightarrow{H}$) by the exclusive rule. In this case, (1) op_1 is a read operation and op_2 is a write operation; (2) op_1 reads from another write operation op_w such that $op_w \rightarrow_H op_2$ in $\overrightarrow{H}$; and (3) op_1, op_2, and op_w are on the same object.

  Since $\overrightarrow{S}$ is equivalent to $\overrightarrow{H}$, $op_w <_S op_1$ holds (they have the same reads-from relation). Since $\overrightarrow{H}$ is under the WW or OO-constraint, $op_w <_S op_2$ holds (they order the write operations (on the same objects) in the same way). Recall that op_1 reads from op_w and op_2 is a write operation. Since $\overrightarrow{S}$ is legal, $op_1 <_S op_2$ must hold.

The claim is proved.

Now, we assume that SG($\overrightarrow{H}$) has a cycle: $op_1 \rightarrow op_2 \rightarrow \cdots \rightarrow op_n \rightarrow op_1$. By the above claim, we have $op_1 <_S op_1$ in $\overrightarrow{S}$. This is a contradiction. Hence, SG($\overrightarrow{H}$) must be acyclic. □

3.3 Sequentializability under the WW-constraint

A protocol based on Theorem 2 would be required to construct sequentialization graph SG($\overrightarrow{H}$) for each history $\overrightarrow{H}$ at run time. The following Lemma 1 states direct relation between a given history $\overrightarrow{H}$ under the WW-constraint and its sequentializability. Thus, a protocol based on Lemma 1 would not have to construct sequentialization graphs and, as a result, would be more efficient.

Lemma 1 : Let $\overrightarrow{H} = (H, <_H)$ be a history under the WW-constraint. SG($\overrightarrow{H}$) is acyclic if and only if $\overrightarrow{H}$ is legal.

Proof :

if part : Suppose that $\overrightarrow{H}$ is legal. On the contrary, assume that SG($\overrightarrow{H}$) contains a cycle. Since $<_H$ is irreflexive antisymmetric, there must exist at least one exclusive edge in the cycle. There are two cases to consider:

1: There is exactly one exclusive edge $r_i(x) \rightarrow_{ex} w_k(x)$ in the cycle. In this case, $r_i(x)$ reads from some write operation $w_j(x)(j \neq k)$ and $w_j(x) <_H w_k(x)$. Since $r_i(x) \rightarrow_{ex} w_k(x)$ is the only exclusive edge in the cycle and $<_H$ is transitive, we have $w_k(x) <_H r_i(x)$. It follows that $w_j(x) <_H w_k(x) <_H r_i(x)$, and consequently $\vec{H}$ is not legal. This is a contradiction.

2: There are N exclusive edges involved in the cycle, where $N > 1$. Suppose $r_{i_1}(x_1) \rightarrow_{ex} w_{k_1}(x_1) \rightarrow r_{i_2}(x_2) \rightarrow_{ex} w_{k_2}(x_2) \rightarrow \cdots \rightarrow r_{i_N}(x_N) \rightarrow_{ex} w_{k_N}(x_N) \rightarrow r_{i_1}(x_1)$ is the cycle in SG($\vec{H}$).

   Consider $w_{k_1}(x_1)$ and $w_{k_2}(x_2)$. Since $\vec{H}$ is under the WW-constraint, either $w_{k_1}(x_1) <_H w_{k_2}(x_2)$ or $w_{k_2}(x_2) <_H w_{k_1}(x_1)$ holds.

   1. If $w_{k_2}(x_2) <_H w_{k_1}(x_1)$ holds, $w_{k_1}(x_1) \rightarrow r_{i_2}(x_2) \rightarrow_{ex} w_{k_2}(x_2) \rightarrow w_{k_1}(x_1)$ forms a cycle with exactly one exclusive edge. As we have shown in Case 1, this leads to a contradiction.
   2. If $w_{k_1}(x_1) <_H w_{k_2}(x_2)$ holds, there exists a cycle $r_{i_1}(x_1) \rightarrow_{ex} w_{k_1}(x_1) \rightarrow w_{k_2}(x_2) \rightarrow r_{i_N}(x_N) \rightarrow_{ex} w_{k_N}(x_N) \rightarrow r_{i_1}(x_1)$ in SG($\vec{H}$). This cycle contains $N - 1$ exclusive edges. This, combining with the above case, implies that if we have a cycle with N exclusive edges, then we also have a cycle with $N - 1$ exclusive edges. Applying this argument recursively, we conclude that if we have a cycle with N exclusive edges, we also have a cycle with exactly one exclusive edge, which leads to a contradiction.

This proves that SG($\vec{H}$) is acyclic.
only if part : Let $\vec{H}$ be a history under the WW-constraint and SG($\vec{H}$) is acyclic. Assume, on the contrary, that $\vec{H}$ is not legal. Then, there must exist a read operation $r_i(x)v$ such that $w_j(x)v <_H w_k(x)u <_H r_i(x)v$. Thus, we have $w_j(x)v \rightarrow w_k(x)u \rightarrow r_i(x)v$ in SG($\vec{H}$). By the exclusive rule, we have $r_i(x)v \rightarrow_{ex} w_k(x)u$ in SG($\vec{H}$). Thus, SG($\vec{H}$) has a cycle, and this contradicts the assumption. Therefore, $\vec{H}$ is legal. □

¿From Theorem 2 and Lemma 1, we have the following:

Corollary 1 : Let $\vec{H} = (H, <_H)$ be a history under the WW-constraint. $\vec{H}$ is sequentializable if and only if $\vec{H}$ is legal. □

Corollary 1 states that in implementations of sequentially consistent DSM systems, if a protocol enforces the WW-constraint and the processor-order relation, it only

needs to guarantee legality when the protocol establishes reads-from relation. ¿From Definition SC and Corollary 1, we have the following theorem:
Theorem 3 : A distributed shared memory system (P, M) is **sequentially consistent** if for each of its execution histories $\vec{H}$, $\vec{H}$ is under the WW-constraint and legal. $\square$

3.4 Sequentializability under the OO-constraint

Similarly, for a history under the OO-constraint, we have the following Lemma:
Lemma 2 : Let $\vec{H} = (H, <_H)$ be a history under the OO-constraint. SG($\vec{H}$) is acyclic if and only if $\vec{H}$ is legal.

Proof :
if part : Suppose that $\vec{H}$ is legal. On the contrary, assume that SG($\vec{H}$) has a cycle. There are two cases to consider:

1: There is exactly one exclusive edge $r_i(x) \rightarrow_{ex} w_k(x)$ in the cycle. This case is the same as Case 1 in the proof of Lemma 1 and leads to contradiction.

2: There are N exclusive edges involved in the cycle, where $N > 1$. Suppose $r_{i_1}(x_1) \rightarrow_{ex} w_{k_1}(x_1) \rightarrow r_{i_2}(x_2) \rightarrow_{ex} w_{k_2}(x_2) \rightarrow \cdots \rightarrow r_{i_N}(x_N) \rightarrow_{ex} w_{k_N}(x_N)$ $\rightarrow r_{i_1}(x_1)$ is the cycle in SG($\vec{H}$).

   Since $\vec{H}$ is under the OO-constraint, either $r_{i_1}(x_1) <_H w_{k_1}(x_1)$ or $w_{k_1}(x_1) <_H$ $r_{i_1}(x_1)$ holds.

   1. Assume that $w_{k_1}(x_1) <_H r_{i_1}(x_1)$ holds. Then, $w_{k_1}(x_1) \rightarrow r_{i_1}(x_1) \rightarrow_{ex}$ $w_{k_1}(x)$ in SG($\vec{H}$) forms a cycle with exactly one exclusive edge. As we have shown in Case 1, this leads to a contradiction.
   2. Assume that $r_{i_1}(x_1) <_H w_{k_1}(x_1)$ holds. Then, there exists a cycle $r_{i_1}(x_1)$ $\rightarrow w_{k_1}(x_1) \rightarrow r_{i_2}(x_2) \rightarrow_{ex} w_{k_2}(x_2) \rightarrow \cdots \rightarrow r_{i_N}(x_N) \rightarrow_{ex} w_{k_N}(x_N) \rightarrow$ $r_{i_1}(x_1)$ in SG($\vec{H}$). This cycle contains $N-1$ exclusive edges. Applying this argument recursively, we conclude that there also exists a cycle with exactly one exclusive edge, which leads to a contradiction.

This proves that SG($\vec{H}$) is acyclic.
only if part : The proof is the same as the only if part of the proof of Lemma 1. $\square$

Note that in the proof of Lemma 1, we used the fact that $w_{k_1}(x_1)$ and $w_{k_2}(x_2)$ are ordered (x_1 may be different from x_2); whereas in the proof of Lemma 2, we used the fact that $r_{i_1}(x)$ and $w_{k_1}(x)$ are ordered.

¿From Theorem 2 and Lemma 2, we have the following:

Corollary 2 : Let $\overrightarrow{H} = (H, <_H)$ be a history under the OO-constraints. $\overrightarrow{H}$ is sequentializable if and only if $\overrightarrow{H}$ is legal. □

¿From Definition SC and Corollary 2, we have the following theorem:
Theorem 4 : A distributed shared memory system (P, M) is **sequentially consistent** if and only if for each of its execution histories $\overrightarrow{H}$, $\overrightarrow{H}$ is under the OO-constraint and legal. □

3.5 Alternative condition for sequential consistency

Scheurich and Dubois gave a sufficient condition of sequential consistency [18]. First, they defined the notion of *performing with respect to a processor*. Let $\{w_i(x)\}+$ denote the sequence of write operations on x following the write operation $w_i(x)$, including $w_i(x)$.

1. A write operation $w_i(x)$ is considered *performed with respect to processor* P_k when a subsequently issued read operation to the same object by P_k returns the value written by a write operation in the sequence $\{w_i(x)\}+$;

2. A write operation $w_i(x)$ is *globally performed* when it is performed with respect to all processors;

3. A read operation $r_i(x)$ is considered *performed with respect to processor* P_k if a subsequently issued write operation to the same address by processor P_k cannot affect the value returned by $r_i(x)$; and

4. A read operation $r_i(x)$ is said to be *globally performed*, if it is performed with respect to all processors and if the write operation which wrote the value read by $r_i(x)$ has been globally performed.

Then, they defined a sufficient condition of sequential consistency as: *Sequential consistency is satisfied in any system if an access may not be performed with respect to any processor until the previous access by the same processor has been globally performed and if accesses of each individual processor are globally performed in program* *order.* Based on this condition, Scheurich and Dubois demonstrated the correctness of the protocols proposed in [8, 17] and also gave a protocol which does not count on atomic update.

A proof that the above condition is a sufficient condition of sequential consistency was not given in [18]. It appears that the condition is similar to the OO-constraint. However, the expression "subsequently issued (operations)" in the definition of *performing with respect to a processor* seems to indicate that this condition is based on the concept of real time. This real time constraint seems to enforce the OO-constraint and guarantee the relation formed by processor-order and reads-from to be acyclic. Thus, this condition is more restrictive than the OO-constraint, and some protocols, such as our protocol based on the OO-constraint given in Section 5.2, would not be developed based on this condition[2].

4 Applying the theory to implementations

In this section, we discuss how to interpret the theorems presented in Section 3 and apply them to implementations. We consider a system that enforces minimal constraints; that is, the order relation in a history produced by the system contains only the processor-order relation, reads-from relation, and either the ww- or oo-relation.

4.1 Implementations based on the WW-constraint

4.1.1 A method to enforce legality

Theorem 3 states that if the DSM system enforces the WW-constraint, the system only needs to maintain the processor-order relation and establish legal reads-from relation by associating each read operation to a write operation. Consider an example history $\vec{H}_1$ under the WW-constraint given in Figure 1. The figure shows a part of the ww-list and some of P_i's operations. The ww-relation, processor-order relation, and reads-from relation are indicated by thick arrows, thin arrows, and dashed arrows, respectively. An exclusive edge of SG($\vec{H}_1$) is also given by a dotted arrow. The execution of processor P_i is $r_i(x_1)\ w_i(x_4)\ r_i(x_3)\ r_i(x_2)\ r_i(x_5)\ r_i(x_4)\ w_i(x_7)\ \cdots$, and the order specified by the ww-relation is $\cdots w_{j_1}(x_1)\ w_{j_2}(x_2)\ w_{j_3}(x_3)\ w_i(x_4)\ w_{j_4}(x_4)\ w_{j_5}(x_5)\ w_{j_6}(x_6)\ w_i(x_7)\ \cdots$. In $\vec{H}_1$, read operations $r_i(x_1)$, $r_i(x_3)$, $r_i(x_2)$, and $r_i(x_5)$

[2] In our OO-constraint based protocol, some local read operations $r_i(x)$ may be executed after a write operation $w_i(x)$ in real time even though $r_i(x) <_{oo} w_i(x)$.

are legal. However, $r_i(x_4)$ is not legal because $r_i(x_4)$ reads x_4 from $w_i(x_4)$, and $w_i(x_4) <_{ww} w_{j_4}(x_4) <_{H_1} r_i(x_4)$. Note that as Lemma 1 states, SG($\overrightarrow{H_1}$) contains an exclusive edge $r_i(x_4) \rightarrow_{ex} w_{j_4}(x_4)$ and a cycle, $w_{j_4}(x_4) \rightarrow w_{j_5}(x_5) \rightarrow r_i(x_5) \rightarrow r_i(x_4)$ $\rightarrow_{ex} w_{j_4}(x_4)$, exists.

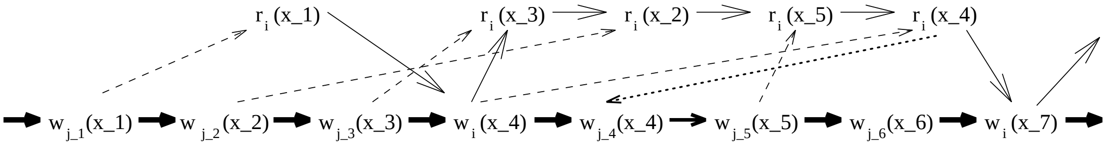

Figure 1: History under the WW-constraint

Since the processor-order relation is easily enforced, we discuss a method to enforce legality. We assume that the DSM is initialized when the execution begins; that is, an imaginary write operation, denoted $w_{Init}(x)$, is assumed to be performed on each object x before the execution begins.

First, we will define some terminology.

Definition (prefix): For an operation op_i in a history $\overrightarrow{H} = (H, <_H)$ which is under the WW-constraint, the prefix of $\overrightarrow{H}$ with respect to op_i, denoted PRE($\overrightarrow{H}, op_i$), is a history $(H^{op_i}, <_H^{op_i})$, where

- $H^{op_i} = \{op_i\} \cup \{op_j | op_j <_H op_i\}$
- $<_H^{op_i} =$ irreflexive transitive closure of Σ, where $\Sigma = \{(a,b) | a, b \in H^{op_i}, a <_{ww} b$ or $a <_{pr} b$ or $a <_{rf} b\} - \{(w, op_i) | w \in H^{op_i}, w <_{rf} op_i\}$

When op_i is a read operation, PRE($\overrightarrow{H}, op_i$) represents a part of the history right at the time when the system tries to establish reads-from relation for op_i; therefore, PRE($\overrightarrow{H}, op_i$) does not include the reads-from relation to op_i.

Definition (last-write): For an operation op_i in a history $\overrightarrow{H}$ which is under the WW-constraint, the **last write** on object x with respect to op_i, denoted LW$_x(op_i)$, is $w(x)$ such that

1. if op_i is the first operation of processor P_i and is a read operation, then $w(x) = w_{Init}(x)$;
2. otherwise, if op_i is a write operation on x, then $w(x) = op_i$;

3. otherwise, in PRE($\vec{H}, op_i$), $w(x) <_H^{op_i} op_i$ and there does not exist another write operation on x, say $w'(x)$, such that $w(x) <_H^{op_i} w'(x)$ and $w'(x) <_H^{op_i} op_i$.

We call the value written by the last-write the "last-write value."

Consider history $\vec{H}_1$ given in Figure 1. Suppose $r_i(x_1)$ is the first operation of P_i. Then, $\mathrm{LW}_{x_k}(r_i(x_1)) = w_{Init}(x_k)$ for all objects x_k. $\mathrm{LW}_{x_k}(w_i(x_4)) = \mathrm{LW}_{x_k}(r_i(x_3)) = \mathrm{LW}_{x_k}(r_i(x_2)) = \mathrm{LW}_{x_k}(r_i(x_5)) = \mathrm{LW}_{x_k}(r_i(x_4)) = w_{j_k}(x_k)$ for $k = 1, 2$, and 3. $\mathrm{LW}_{x_4}(w_i(x_4)) = \mathrm{LW}_{x_4}(r_i(x_3)) = \mathrm{LW}_{x_4}(r_i(x_2)) = \mathrm{LW}_{x_4}(r_i(x_5)) = w_i(x_4)$, but $\mathrm{LW}_{x_4}(r_i(x_4)) = w_{j_4}(x_4)$.

Consider the following condition when the system defines reads-from relation for $r_i(x)$:

Condition 1: $r_i(x)$ reads a value written by $w_j(x)$ such that $\mathrm{LW}_x(r_i(x)) <_{ww}^* w_j(x)$.

Theorem 5 : A history under the WW-constraint is legal iff for each read operation $r(x)$, Condition 1 is satisfied.

Proof:

if part : Suppose that the protocol establishes reads-from relation such that $r_i(x)$ reads x from $w_j(x)$ by following Condition 1.

1. At the moment when the reads-from relation is established, there is no $w_k(x)$ such that $w_j(x) <_{ww} w_k(x)$ and $w_k(x) <_H r_i(x)$.

2. At any time in future, no additional relation (due to processor-order relation, reads-from relation, or ww-relation) will be created to any of the operations of P_i which precede $r_i(x)$ in the processor-order of P_i.

Thus, $r_i(x)$ is legal.

only if part : Suppose that there exists a read operation $r(x)$ such that Condition 1 is not satisfied; that is, $r_i(x)$ reads from $w_j(x)$ such that $w_j(x) <_{ww} \mathrm{LW}_x(r_i(x))$. Since, $\mathrm{LW}_x(r_i(x)) <_H r_i(x)$, $r_i(x)$ is not legal. □

In H_1, the reads-from relation of each of $r_i(x_1)$, $r_i(x_3)$, $r_i(x_2)$, and $r_i(x_5)$ satisfies Condition 1. For $r_i(x_4)$, however, Condition 1 is not satisfied.

4.1.2 Protocols using local processor memory

In the previous subsection, we established a general condition that DSM systems under the WW-constraint should follow. In this section, we focus our attention on implementations of a special type of DSM systems under the WW-constraint —

DSM systems that use processor memory. We will establish some conditions for efficient implementations of such systems.

Under the WW-constraint, all write operations are totally ordered. If all processors recognize the total order of the write operations, it is possible that each processor locally establishes legal reads-from relation for each read operation. This implies that read operations may be processed locally at each processor. This approach requires that each processor holds (a part of) the ww-list. One way to achieve this is to provide each processor with local processor memory which stores copies of DSM objects. The protocols presented in [2, 5, 7] fall in this category.

In order for every processor to recognize the ww-list, each write operation needs to be propagated to other processors. In implementations of sequential consistency, however, processors do not have to recognize the current ww-list in real time. Instead, at any time, processors may recognize only some prefix of the ww-list. The processor memory at each processor stores (a part of) such a prefix.

A read operation reads a value from processor memory if the memory holds an appropriate value. There are two approaches to managing processor memory: update and invalidation approaches. In the update approach, when a processor is notified of a write operation, it updates the processor memory. In the invalidation approach, when a processor receives a write operation, it invalidates the current value in the associated object in its processor memory. If a processor tries to read an invalidated object, it must find a valid value from other places.

In the rest of this subsection, we develop principles that systems with processor memory may follow.

At any time, let LAST_{ww} denote the last write operation in the current ww-list. In protocols which use processor memory, we say that a processor **recognizes** a write operation $w_j(x)$ in the ww-list when it applies update or invalidation of $w_j(x)$ in its processor memory. We use $\mathrm{LAST}_i(op_i)$ to denote the last write operation in the ww-list that P_i has recognized when operation op_i is executed. Clearly, $\mathrm{LAST}_i(op_i)$ $<^*_{ww}$ LAST_{ww} for any operation op_i of any processor P_i.

We now have the following theorem:

Theorem 6: In a history under the WW-constraint, if a read operation $r_i(x)$ reads x from $w_j(x)$ such that $\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x))) <^*_{ww} w_j(x)$, $r_i(x)$ is legal. □

A proof of Theorem 6 is in Appendix A. Theorem 6 does not state iff relation. However, as shown in the next subsection, it provides efficient implementations for systems with processor memory. Recall that $\mathrm{LAST}_i(r_i(x))$ is the last write operation in the prefix of the ww-list that P_i has recognized when it executes $r_i(x)$. The condition "$\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x))) <^*_{ww} w_j(x)$" states that a processor does not have to hold more than one value for each object in its processor memory. If the processor memory holds a value, then it should hold the value written by only the most recent write operation in the prefix of the ww-list that the processor has recognized.

In the next subsection, we briefly review three protocols presented in [2, 5, 7] and show that each protocol satisfies Theorem 6.

4.1.3 Review of previous protocols based on the WW-constraint

In the protocols in [2, 5, 7], all processors are connected by a network which supports (by means of either software or hardware) atomic broadcasting. Each processor has a local processor memory. In addition, the protocols in [2, 7] assume that a centralized shared memory module is connected to the network, which holds $\mathrm{LW}_x(\mathrm{LAST}_{ww})$ for all objects x. The WW-constraint is enforced by atomic broadcasting of write operations — the ww-list is formed in the order in which the atomic broadcast processes write operations. In the protocol in [2], the centralized shared memory also plays a role in enforcing the WW-constraint. A processor updates (or invalidates) its processor memory in the same order as it receives broadcast write operations. This guarantees that every processor recognizes some prefix of the ww-list. The protocols in [2, 5] use an update approach, and the protocol in [7] uses an invalidation approach.

In the rest of the paper, in protocols that assume the existence of shared memory modules (Brown protocol [7], the protocol by Afek *et al.* [2], and the two protocols presented in Section 5), we use the following terminology:

- A read operation that reads a value in the processor memory is called *internal read*;
- A read operation that requires an access to the shared memory module is called *external read*;
- Write operations and external read operations are referred to as *external operations*, since in these protocols, a write operation always accesses the shared memory.

Now, we review the three protocols and show each protocol satisfies Theorem 6.

Attiya and Welch's protocol :
In this protocol [5], a processor memory is as large as the entire DSM, and read operations always find values in the processor memory. A write operation initiates

atomic broadcasting of an update request of the object. The time when an atomic broadcasting executed is considered to be the time when the ww-list (and LAST_{ww}) is updated. When a processor receives a broadcast update request, it immediately updates the processor memory. This is the time when LAST_i is updated. Since the updates at processor P_i are done in the order in which P_i receives the corresponding update requests, for any read operation $r_i(x)$, its processor memory always contains only values written by $\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$. Therefore, $r_i(x)$ reads x only from $\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$, and from Theorem 6, $r_i(x)$ is legal.

Brown's protocol :
This protocol [7] assumes that, in addition to local processor memory, a centralized shared memory module is connected to the network. The protocol is based on the invalidation approach.

When a processor issues a write operation, it updates the value of the object both in its processor memory and in the shared memory and also initiates atomically broadcasting of an invalidation request for the object to each of the other processors. They are all done as a single atomic operation, and the time when this atomic operation is executed is considered to be the time when the ww-list (and LAST_{ww}) is updated. Because of the update of the shared memory module, the shared memory always holds only values written by $\mathrm{LW}_x(\mathrm{LAST}_{ww})$ for all objects x.

Invalidation requests are queued at each processor. When a processor issues a read operation, if a valid value is found in its processor memory, it reads the value (internal read); otherwise, it reads a value from the shared memory and updates its processor memory (external read). When a processor accesses the shared memory (an external operation: either to read or to write), all the invalidation requests in the queue are processed on the processor memory and the queue is emptied. This timing is considered to be the time when LAST_i is updated.

Let op_i^1 and op_i^2 be two consecutive external operations of processor P_i. When P_i completes the execution of op_i^1, all the queued invalidation requests up to LAST_{ww} are performed on P_i's processor memory. Thus, $\mathrm{LAST}_i(op_i^1)$ becomes LAST_{ww}.

First, consider legality of an external read operation; that is, $r_i(x) = op_i^1$. Then, $r_i(x)$ reads a value in the shared memory module, which is $\mathrm{LW}_x(\mathrm{LAST}_i(op_i^1))$ ($=$ $\mathrm{LW}_x(\mathrm{LAST}_{ww})$). Thus, from Theorem 6, $r_i(x)$ is legal.

Next, consider legality of an internal read operation; that is, the case in which $r_i(x)$ is an operation in between op_i^1 and op_i^2. Execution of the invalidation requests at op_i^1 invalidates values which were written by any write operations $w_k(x)$ such that $w_k(x) <_{ww} \mathrm{LW}_x(\mathrm{LAST}_i(op_i^1))$ ($= \mathrm{LW}_x(\mathrm{LAST}_{ww})$, since $\mathrm{LAST}_i(op_i^1) = \mathrm{LAST}_{ww}$).

Therefore, after op_i^1 is executed, the processor memory holds only values written by $\mathrm{LW}_y(\mathrm{LAST}_i(op_i^1))$ for some objects y. After op_i^1 is executed, no invalidation or update (through accessing the shared memory module) will be performed on the processor memory until op_i^2 is executed. Thus, for any internal read operations $r_i(x)$ in between op_i^1 and op_i^2, $\mathrm{LAST}_i(r_i(x))$ is equal to $\mathrm{LAST}_i(op_i^1)$. Therefore, $r_i(x)$ reads x from $\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$, and Theorem 6 holds.

Afek, Brown, and Merritt's protocol :
This protocol [2] assumes a similar architecture as Brown's protocol, except that at each processor P_i, there are two queues, IN_i and OUT_i, associated with the processor memory. The protocol is based on the update approach.

When P_i writes a value to an object, it places a write request in OUT_i. As a separate operation, the shared memory module picks up a request at the head of OUT_j of any processor P_j. It updates the shared memory based on the request and places the request in IN_k of all processors P_k (using atomic broadcasting). These steps are done atomically. The time when this atomic operation is executed is the time when the ww-list (and LAST_{ww}) is updated. The order of write operations chosen by the shared memory module defines the ww-relation.

As another separate operation, each processor P_i dequeues the request at the head of IN_i queue, say the request for $w_j(x)$, and updates its processor memory based on the request. The time when this operation is done is the time when LAST_i is updated at P_i. Thus, at this moment, P_i is considered to have recognized $w_j(x)$. A read operation returns a value found in its processor memory.

Note that P_i may place requests for consecutive write operations in OUT_i without waiting for a completion of the previous write operation. A read operation, however, must be delayed if any write request issued previously by the processor is still in its IN or OUT queue. This condition is necessary to preserve processor-order relation.

Assume that when a read operation $r_i(x)$ is executed, $w_j(y)$ is the last write operation which has caused the update of processor memory at P_i; that is, $\mathrm{LAST}_i(r_i(x)) = w_j(y)$. Then, P_i's processor memory contains only the values written by $\mathrm{LW}_x(w_j(y))(= \mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$ for all objects x. Therefore, $r_i(x)$ reads a value from $\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$, and from Theorem 6, $r_i(x)$ is legal.

4.1.4 Note on the well-formed property

In the protocols in [2, 7], it would be possible that a processor performs local read operations for infinitely long time, without recognizing write operations performed by other processors. For example, consider the following scenario [7]:

- Data item x is initialized to 0.
- Processor P_i executes statement "**while** ($x = 0$) **do** skip **od** ."

  In the first execution of condition "$x = 0$", it reads x from $SMem$.
- Then, processor P_j executes statement "$x := 1$."

Processor P_i will never know the execution by P_j and keep looping in the while statement. This situation is considered to be that all of the (infinitely many number of) read operations performed by P_i are scheduled before the write operation by P_j in an illusory sequential execution history $\overrightarrow{S}$ and violates the well-formed property. In order to enforce the well-formed property on an execution history, each processor must recognize updates performed by other processors within a finite length of its execution steps. Thus, if a processor P_i does not update its local memory for a certain length of time (e.g., the number of consecutive local read operations exceeds a certain predefined limit), it must be artificially forced to perform updates even though it is not necessary in terms of guaranteeing legality.

4.2 Implementations based on the OO-constraint

Theorem 4 states that if the DSM system enforces the OO-constraint, the system only needs to establish reads-from relation by associating each read operation to a write operation such that legality and the processor-order relation are satisfied. The OO-constraint may be enforced by object level mutual exclusion. For example, broad band atomic broadcasting may be used to transmit operations, where each band is assigned to different object. In another approach, object based shared memory modules may be used to execute operations, where each memory module is assigned to a different object, and all operations on the same object are sent to the associated memory module for synchronization.

Note that in order to enforce the OO-constraint, not only each write operation but also each read operation must be globally synchronized with other write operations on the same object. Synchronizing only write operations on the same objects is not enough to guarantee a given history to be sequentializable. Thus, the OO-constraint may be more difficult to enforce on systems with local processor memory, since it is difficult to synchronize internal read operations of different processors with other write operations. For example, consider a simple DSM system with two objects, x and y and two processors P_i and P_j. Assume that each processor has a processor memory. Executions of P_i and P_j are

$$\overrightarrow{P_i} = w_i(x) <_{P_i} w_i(y) <_{P_i} r_i(x) \quad \text{and} \quad \overrightarrow{P_j} = w_j(y) <_{P_j} w_j(x) <_{P_j} r_j(y).$$

Assume that $r_i(x)$ and $r_j(y)$ are performed locally; as a result, P_i does not notice $r_j(y)$ when it executes $w_i(y)$, and P_j does not notice $r_i(x)$ when it executes $w_j(x)$. Thus, P_i and P_j agree only on the order of write operations as $w_i(x) <_{oo} w_j(x)$ and $w_j(y) <_{oo} w_i(y)$. Figure 2(a) and 2(b) show possible executions in P_i's and P_j's view, respectively. Thick arrows represent object-order relation, and thin arrows represent processor-order relation. Each execution seems to be legal. However, the graph which combines the executions in P_i's and P_j's views has a cycle, and does not represent a history (Figure 2(c)).

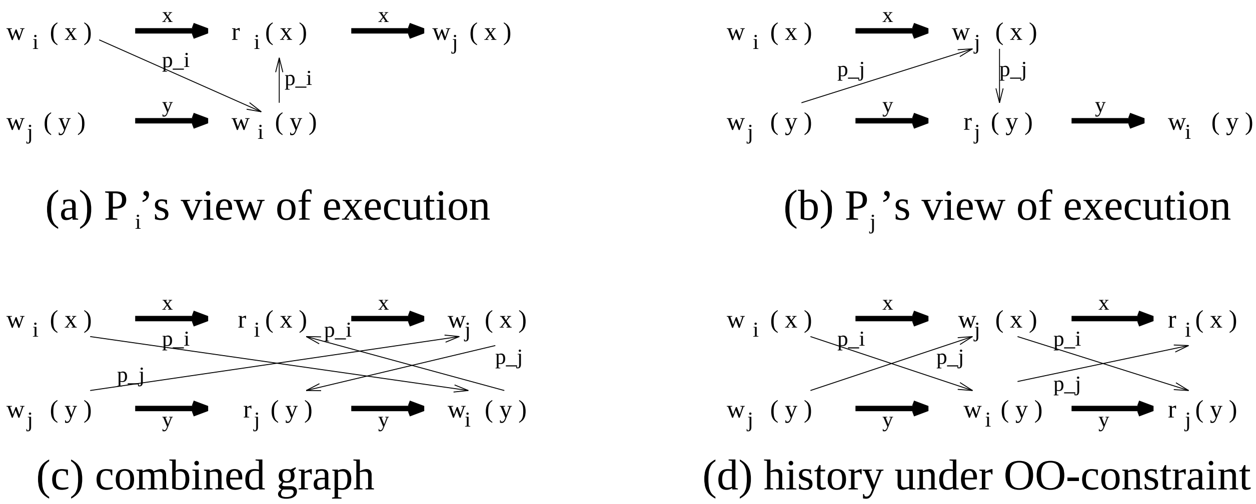

Figure 2: History under the OO-constraint

Now assume that the read and write operations are synchronized as shown in Figure 2(d). The order is compatible with the processor-order relation. Then, the system only needs to guarantee legality. Thus, if $r_i(x)$ reads from $w_j(x)$ and $r_i(y)$ reads from $w_i(y)$, the resulting history is sequentializable.

Note that the above problem would occur because no component in the system has all the information necessary to enforce the OO-constraint. Each processor obtains only partial information: information about the write operations by all the processors and the local read operation by itself. Based on the incomplete information, each processor tries to enforce the OO-constraint, and as a result, the processors establish incompatible cyclic relation.

A protocol presented in [19] is based on the OO-constraint. It assumes a multiprocessor architecture that interconnects N processors with N object memory modules by a buffered multistage network. Each object memory module stores a disjoint set of objects. Thus, one extreme is that each memory module stores exactly one object. Each write and read operation is sent to the associated object memory module and processed by the module sequentially. In this approach, each object memory module obtains the complete information necessary to enforce the OO-constraint; that is, the information about all the (read and write) operation on the object. Therefore, the OO-constraint and legality are easily enforced, and the problem described above does not occur.

In this protocol, however, enforcing processor-order relation is more difficult. Since each request has to pass $O(\log N)$ multistage switches, the protocol tries to hide network latency by pipelining requests. Thus, if two consecutive requests by one processor are routed to different memory modules, they may not arrive at memory modules in the same order as the processor-order. In order to guarantee the processor-order relation, the protocol attaches a sequence number to each request. Each memory module maintain a table, one entry for each processor, which stores the next sequence number of the request to be executed. Since all memory modules must see a consistent view of the table, they communicate with each other to update all the table entries whenever an operation is performed at one module. To achieve this at high speed, the protocol assumes hardware support.

5 New implementations based on the WW- and OO-constraints

We have developed two protocols for sequentially consistent DSM systems; one based on the WW-constraint and another based on the OO-constraint. In this section, we briefly introduce these two protocols. Note that as in the protocols in [2, 7], both protocols may violate the well-formed property without an extra mechanism.

5.1 Implementation based on the WW-constraint

5.1.1 Protocol

All three protocols for sequentially consistent DSM systems based on the WW-constraint reviewed in the previous section rely on atomic broadcasting for synchronization of write operations. If the underlying hardware provides an atomic broadcast facility, these protocols can be implemented efficiently. However, without such support, they can be expensive.

Based on the sequentializability theory, we have developed an efficient protocol[3] for sequentially consistent DSM systems, aiming at an environment where no special support for atomic broadcast exists and the cost of communication is high, such as distributed memory parallel machines. The protocol eliminates the need of atomic broadcast and significantly reduces the amount of communication when compared with the previous three protocols. Although the protocol adopts the invalidation approach, a modification to incorporate the update approach is simple.

The protocol uses a shared memory module abstraction (SMem) which works as a centralized arbitrator. SMem maintains memory for all the objects of the DSM system. Each processor has local processor memory and can communicate with SMem. All write operations are sent to SMem, and SMem executes the operations by updating its memory sequentially to enforce the WW-constraint. The time when SMem updates its memory is the time when the ww-list (and LAST_{ww}) is updated. If a read operation tries to access an invalidated value in its processor memory, the processor accesses a valid value from SMem. Note that the shared memory module is an abstraction. It may be implemented by a special processor in the network. However, other methods can be used. For example, the memory objects may be replicated, and quorum protocols may be used to improve fault tolerance and accessibility [11].

Under the above assumptions, the WW-constraint and the processor-order relation are naturally enforced. In order to guarantee legality, we have developed a mechanism by which SMem can efficiently identify obsolete values held in the processor memory of each processor. This mechanism minimizes the amount of information flow among processors so that only information necessary to maintain sequential consistency is propagated to each processor. This further reduces the amount of communication.

5.1.2 A mechanism to guarantee legality

Let $r_i(x)$ be an internal read operation of P_i. Let op_i be the last external operation which precedes $r_i(x)$ in $\overrightarrow{P_i}$. As Theorem 6 states, one approach to guarantee legality is to prevent any $r_i(x)$ from reading a value written by a write operation which precedes $\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$. Our mechanism is to have SMem efficiently identify, whenever it executes an external operation op_i, which objects in P_i's processor memory do and do not hold the last-write values with respect to LAST_{ww}. Based on this information, the system invalidates the objects x in P_i's processor memory which do not hold

[3] A brief description of this protocol is also found in [21].

the value written by $\mathrm{LW}_x(\mathrm{LAST}_{ww})$. This is the time when LAST_i is updated — $\mathrm{LAST}_i(op_i)$ becomes LAST_{ww}.

SMem maintains a two dimensional binary array, Hold_Last_Write [Processor_Range, Object_Range]. At any time, Hold_Last_Write$[i, x]$ is 1 if object x in the local memory at processor P_i holds the value written by $\mathrm{LW}_x(\mathrm{LAST}_{ww})$; it is 0 otherwise. First, each element of Hold_Last_Write is initialized to 0.

A detailed description of the protocol and its correctness proof are given in Appendix B. SMem updates Hold_Last_Write in the following way. When SMem performs $w_i(x)$, $w_i(x)$ becomes LAST_{ww}; therefore, only the processor memory for x at P_i holds $\mathrm{LW}_x(\mathrm{LAST}_{ww})$, and processor memory for x at all other processors will no longer hold $\mathrm{LW}_x(\mathrm{LAST}_{ww})$. Thus, Hold_Last_Write$[i, x]$ is set to 1 and Hold_Last_Write$[i, y]$, $y \neq x$, is set to 0 (refer to steps (c) and (d) in the protocol in Appendix B).

SMem always holds the value written by the last-write operation with respect to LAST_{ww}. Thus, when an external read operation $r_i(x)$ is executed at SMem, the value in the processor memory for object x at P_i becomes the last-write value with respect to LAST_{ww}; therefore, Hold_Last_Write$[i, x]$ is set to 1 (refer to step (f) in the protocol in Appendix B).

When SMem executes an external operation by P_i, it returns the i^{th} row of Hold_Last_Write to P_i. Upon a receipt of the information, P_i invalidates object values, in its processor memory, corresponding to x such that Hold_Last_Write$[i, x] = 0$ (refer to steps (a), (b), (e), and (g) in the protocol in Appendix B).

5.2 A protocol based on the OO-constraint

As discussed in Section 4, the OO-constraint is easier to enforce on an implementation which does not use processor memory. In this section, we sketch one method to apply the OO-constraint to systems with processor memory. In our WW-constraint based protocol, SMem may become a bottleneck. One way to relieve the load of SMem is to divide SMem into several primary memory modules and distribute the job of SMem to these memory modules. Each memory module handles a subset of the objects in the DSM.

A detailed description of the protocol and its correctness proof are given in Appendix C. A system consists of multiple processors and memory modules connected via a network. Each processor is equipped with processor memory. Let the memory module which handles object (or a set of objects) x be denoted PMem_x. When a processor P_i issues a write operation $w_i(x)$, it sends a message to PMem_x. When P_i issues a read operation $r_i(x)$, it tries to read from its processor memory. However, if the value is invalidated, P_i sends a message to PMem$_x$. Thus, synchronization of a pair of external operations (either a pair of write operations or a pair of an external read and a write operations) is easily achieved by PMem$_x$. However, synchronization of a write operation and an internal read operation is difficult. Our method to achieve such synchronization is to have memory modules communicate with one another when an external operation is performed.

Let LAST$_x$ denote the write operation which is performed most recently at PMem$_x$. Each PMem$_x$ maintains memory M$_x$ to store objects and one dimensional binary array Hold_Last_Write$_x$ [Processor_Range]. Hold_Last_Write$_x[i] = 1$ iff P_i's processor memory holds the value written by LAST$_x$.

When P_i issues an external operation op_i on object x, the message to PMem$_x$ contains not only the operation itself, but also the information about which values in P_i's processor memory are currently valid. Assume that a value of y is in P_i's processor memory. PMem$_x$, then, communicates with PMem$_y$ to see if P_i's memory contains the value written by LAST$_y$.

1. If so, a future internal read operation on y by P_i which would read from P_i's processor memory, say $r_i(y)$, is ordered (in the oo-relation) in between LAST$_y$ and the next write operation on y. Thus, the system does not invalidate the value of y in P_i's processor memory. (refer to step (c) in the protocol in Appendix C)

2. Otherwise, $r_i(y)$ is ordered after the current LAST$_y$ in the oo-relation and should not read a value in P_i's processor memory. Thus, the system invalidates the value of y in P_i's processor memory. (refer to step (b) in the protocol in Appendix C)

If the system has hardware support, similar to the one described in [19], communication among memory modules can be done efficiently. In such a system, each memory module manages the two dimensional binary array as our WW-constraint based protocol. When PMem$_x$ processes external operation op_i, it sets and resets, by using the hardware support, entries of Hold_Last_Write managed by all other memory modules in the same way as explained above.

This protocol may be combined with the first protocol to form a hierarchical protocol. Assume that system consists of a wide area network that connects several local area networks. Assume that the access pattern of the processors in each local area network shows locality; that is, each network tends to access a certain portion of the shared memory objects more often than other parts. Then, within each local area network, the WW-constraint based protocol is used, and within the wide area network, this OO-constraint based protocol is used as distributing primary object modules based on the pattern of locality.

6 Conclusion

Even though there is strong similarity between concurrency control theory and correctness criteria of DSM systems, much of the research in DSMs has been done independently. We have developed a theory for sequential consistency (called *sequentializability theory*) that has deep practical implications in implementations of competing operations in DSMs in very much the same way that concurrency control theory does in implementations of many concurrency control protocols. The results are useful not only to clarify previously proposed implementations but also to develop new efficient implementations for consistency criteria which provide competing operations, such as sequential consistency, weak ordering (with sequential consistency for competing accesses), and release consistency (with sequential consistency for competing accesses). Using the sequentializability theory, we demonstrated the correctness of existing protocols for sequential consistency and also developed new protocols.

7 Acknowledgement

Discussions with Dr. Gurdip Singh were helpful to develop the protocols presented in Section 5.

References

[1] Adve, S.V. and Hill, M.D. Weak ordering - a new definition. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–14, 1990.

[2] Afek, Y., Brown, G., and Merritt, M. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, 1993.

[3] Ahamad, M., Burns, J., Hutto, P., and Neiger, G. Causal memory. In *Proceedings of 5th IWDAG, LNCS 579*, pages 9–30, 1991.

[4] Attiya, H. and Friedman, R. A correctness condition for high-performance multiprocessors. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 679–690, 1992.

[5] Attiya, H. and Welch, J. Sequential consistency versus linearizability. *ACM TOCS*, 12(2):91–122, 1994.

[6] Bernstein, P.A, Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Co., 1987.

[7] Brown, G. Asynchronous multicaches. *Distributed Computing*, 4:31–36, 1990.

[8] Dubois M. and Briggs F.A. Effects of cache coherency in multiprocessors. *IEEE Transactions on Computers*, C-31(11), 1982.

[9] Dubois, M., Scheurich, C., and Briggs, F. Memory access buffering in multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 434–442, 1986.

[10] Dwarkadas, S., Keleher, P., Cox, A.L., and Zwaenepoel, W. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the International Symposium on Computer Architecture*, pages 144–155, 1993.

[11] Garcia Molina H. and Barbara D. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.

[12] Gharachorloo, K. and Gibbons, P. Detecting violations of sequential consistency. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 1991.

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

[14] Ibaraki T., Kameda T., and Minoura T. Serializability with constraints. *ACM Transactions on Database Systems*, 12(3):429–452, 1987.

[15] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[16] Papadimitriou C.H. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[17] Rudolph L. and Segall Z. Dynamic decentralized cache schemes for mimd parallel processors. In *11th International Symposium on Computer Architecture*, pages 340–347, 1984.

[18] Scheurich, C. and Dubois, M. Correct memory operation of cache-based multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 234–243, 1987.

[19] Stenstrom, P. A latency-hiding scheme for multiprocessors with buffered multistage networks. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 39–42, 1992.

[20] Taylor, R.N. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1083.

[21] Zhou, Z., Mizuno, M., and Singh, G. A sequentially consistent distributed shared memory. In *Proceedings of the International Conference on Computer and Information*, pages 165–169, 1993.

Appendix A: Proof of Theorem 6

We first define terminology and show some properties.

The following proposition holds:

Proposition 1: Let $w_i(x)$ and $w_j(y)$ be write operations such that $w_i(x) <_{ww} w_j(y)$. Then, $\mathrm{LW}_z(w_i(x)) <^*_{ww} \mathrm{LW}_z(w_j(y))$ for any object z.

Proof: Note that for any write operation $w_i(x)$, $\mathrm{LW}_z(w_i(x)) <^*_{ww} w_i(x)$.

Assume, on the contrary, that $\mathrm{LW}_z(w_j(y)) <_{ww} \mathrm{LW}_z(w_i(x))$. There are two cases to consider: $y = z$ and $y \neq z$.

1. If $y = z$, then $\mathrm{LW}_z(w_j(y)) = w_j(y)$. Since $w_i(x) <_{ww} w_j(y)$ and $\mathrm{LW}_z(w_i(x)) <^*_{ww} w_i(x)$, $w_i(x) <_{ww} \mathrm{LW}_z(w_j(y)) <_{ww} \mathrm{LW}_z(w_i(x)) <^*_{ww} w_i(x)$. This is a contradiction.

2. If $y \neq z$, then $\mathrm{LW}_z(w_j(y)) <_{ww} \mathrm{LW}_z(w_i(x)) <_{ww} w_j(y)$. This contradicts the definition of "last write."

Therefore, $\mathrm{LW}_z(w_i(x)) <^*_{ww} \mathrm{LW}_z(w_j(y))$ □

We now define the following terminology:

Definition (most-recent-write): For a read operation $r_i(x)$ in a history $\vec{H}$ that is under the WW-constraint, the most-recent-write with respect to $r_i(x)$, denoted $\mathrm{MRW}(r_i(x))$, is a write operation $w_j(y)$ such that

1. $w_j(y) = \mathrm{LW}_y(r_i(x))$, and
2. $\mathrm{LW}_z(r_i(x)) <_H w_j(y)$ for any object $z \neq y$. □

$\mathrm{MWR}(r_i)$ is the latest operation among the last writes of all objects with respect to $r_i(x)$. For example, consider history $\vec{H_1}$ given in Figure 1. $\mathrm{MWR}(r_i(x_3))$ = $\mathrm{MWR}(r_i(x_2))$ = $\mathrm{MWR}(r_i(x_5))$ = $w_i(x_4)$. $\mathrm{MWR}(r_i(x_4))$ = $w_{j_5}(x_5)$.

Recall that we consider DSM systems in which each processor locally establishes reads-from relation for each read operation based on the values in its local processor memory. Since $\mathrm{LAST}_i(r_i(x))$ is the latest write operation in the ww-list that processor P_i has recognized, there must be no reads-from relation from any write operation that follows $\mathrm{LAST}_i(r_i(x))$ to any read operation that precedes $r_i(x)$ in the processor-order of P_i. Thus, it is easy to see that the following property holds:

Proposition 2: In a history under the WW-constraint, for any read operation $r_i(x)$, $\mathrm{MRW}(r_i(x)) <^*_{ww} \mathrm{LAST}_i(r_i(x))$. □

Also, the following proposition holds:

Proposition 3: In a history under the WW-constraint, for any read operation $r_i(x)$, $\mathrm{LW}_x(\mathrm{MRW}(r_i(x))) = \mathrm{LW}_x(r_i(x))$. □

Proof : Let $\mathrm{MRW}(r_i(x))$ be $w^1(y)$. There are two cases to consider:

1. Case 1: $x = y$ ($\mathrm{MRW}(r_i(x)) = w^1(x)$).

   Then, clearly, $\mathrm{LW}_x(\mathrm{MRW}(r_i(x))) = \mathrm{LW}_x(r_i(x)) = w^1(x)$.

2. Case 2: $x \neq y$

   Let $\mathrm{LW}_x(\mathrm{MRW}(r_i(x)))$ be $w^2(x)$. Then, there does not exist $w^3(x)$ such that $w^2(x) <_{ww} w^3(x) <_{ww} \mathrm{MRW}(r_i(x))(= w^1(y))$. Assume, on the contrary, that $\mathrm{LW}_x(\mathrm{MRW}(r_i(x))) \neq \mathrm{LW}_x(r_i(x))$. Then, there must exist $w^4(x)$ such that $w^1(y) <_{ww} w^4(x) <_{ww} r_i(x)$. This contradicts the assumption that $w^1(y)$ is the most recent write with respect to $r_i(x)$ ($w^1(y) = \mathrm{MRW}(r_i(x))$). Thus, $\mathrm{LW}_x(\mathrm{MRW}(r_i(x))) = \mathrm{LW}_x(r_i(x))$.

□

Using Theorem 5 and Properties 1, 2, and 3, we prove Theorem 6.

Proof of Theorem 6 : ¿From Proposition 2, $\mathrm{MRW}(r_i(x)) <^*_{ww} \mathrm{LAST}_i(r_i(x))$. By applying Proposition 1, $\mathrm{LW}_x(\mathrm{MRW}(r_i(x)) <^*_{ww} \mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$.

Theorem 5 states that if $r_i(x)$ reads x from $w_j(x)$ such that $\mathrm{LW}_x(r_i(x)) <^*_{ww} w_j(x)$, $r_i(x)$ is legal. $\mathrm{LW}_x(r_i(x)) = \mathrm{LW}_x(\mathrm{MRW}(r_i(x)))$ from Proposition 3, and $\mathrm{LW}_x(\mathrm{MRW}(r_i(x)) <^*_{ww} \mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$ from the above. Thus, if $r_i(x)$ reads x from $w_j(x)$ such that $\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x))) <^*_{ww} w_j(x)$, $r_i(x)$ is legal. □

Appendix B: Protocol based on the WW-constraint

Protocol

SMem manages the following data structures:

- Object memory : M[Object_Range]
- Two-dimensional binary array : Hold_Last_Write [Processor_Range, Object_-Range].
  - Hold_Last_Write[i, x]=1 if object x in the processor memory at processor P_i holds a value written by LW$_x$(LAST$_{ww}$),
  - Hold_Last_Write[i, x]=0 if object x in the processor memory at processor P_i does not hold a value written by LW$_x$(LAST$_{ww}$).

  Each element of Hold_Last_Write is initialized to 0.

Each processor P_i maintains the following data structures:

- One-dimensional binary array : Valid$_i$ [Object_Range] :
  - Valid$_i[x]$ = 1 if object x in the processor memory is valid
  - Valid$_i[x]$ = 0 if object x in the processor memory is not valid

  Each element of Valid$_i$ is initialized to 0.
- Cache memory: $C_i[x]$
  for each object x such that Valid$_i[x] = 1$, $C_i[x]$ holds a value of object x

In the following protocol, we denote all the elements in a one-dimensional array R by $R[*]$. Similarly, we denote all the elements in the i^{th} row and the x^{th} column of a two-dimensional array R by $R[i,*]$ and $R[*,x]$, respectively.

Operations at processor i

Write(x, v)::
send [write,x, v] message to *SMem*;
receive [New_Valid[$*$]] message from *SMem*;
(a) Valid$_i[*]$:= New_Valid[$*$];
/$*$ element-wise assignments $*$/

$C_i[x] := v;$

Read(x)::
if valid$_i[x] = 0$ **then**
send [read,x] message to *SMem*;
receive [v, New_Valid[$*$]] from *SMem*;
(b) Valid$_i[*]$:= New_Valid[$*$];
$C_i[x] := v;$
endif
return $C_i[x]$;

Operations at *SMem*:

Process [write,x, v**] message from processor** i::
$M[x] := v;$
(c) Hold_Last_Write[$*, x$] := 0;
(d) Hold_Last_Write[i, x] := 1;
(e) send [Hold_Last_Write[$i, *$]] to processor i;
/$*$ send the i's row of Hold_Last_Write to processor P_i.
Processor P_i receives the row in New_Valid $*$/

Process [read,x**] message from processor** i::
(f) Hold_Last_Write[i, x] := 1;
(g) send [$M[x]$, Hold_Last_Write[$i, *$]] to processor P_i;

Note that each procedure is executed atomically.

Correctness:

Theorem 7: The above protocol implements a sequentially consistent distributed shared memory system.

Proof of Theorem 7: Since SMem processes all the write operations sequentially and this order relation constitutes the ww-relation, the WW-constraint is enforced.

We show that Theorem 6 is satisfied. If $r_i(x)$ is an external operation, it reads from LW$_x$(LAST$_{ww}$), and Theorem 6 holds. Now, suppose that $r_i(x)$ is an internal read operation. Let op_i^1 and op_i^2 be two consecutive external operations of P_i such that $op_i^1 <_{pr} r_i(x) <_{pr} op_i^2$. After op_i^1 is performed, P_i's processor memory holds only values of objects x written by $\mathrm{LW}_x(\mathrm{LAST}_{ww})$ $(= \mathrm{LW}_x(\mathrm{LAST}_i(op_i^1))))$. Since internal read operations do not change LAST_i and there is no external operation between op_i^1 and $r_i(x)$, $\mathrm{LAST}_i(r_i(x)) = \mathrm{LAST}_i(op_i)$, Therefore, $r_i(x)$ reads from $\mathrm{LW}_x(\mathrm{LAST}_i(r_i(x)))$, and Theorem 6 is satisfied. $\Box$

Appendix C: Protocol based on the OO-constraint

Protocol

The primary copy site of object x, $PMem_x$ manages the following data structures:

- Object memory : M_x
- One-dimensional binary array : Hold_Last_Write$_x$[Processor_Range]
  - Hold_Last_Write$_x[i] = 1$: object x in the processor memory at processor i holds a value written by LAST$_x$,
  - Hold_Last_Write$_x[i] = 0$: object x in the processor memory at processor i does not hold a value written by LAST$_x$.

  Each element of Hold_Last_Write is initialized to 0.

Each processor P_i maintains the following data structures:

- One-dimensional binary array : Valid$_i$[Object_Range] :
  - Valid$_i[x] = 1$: object x in the processor memory is valid
  - Valid$_i[x] = 0$: object x in the processor memory is not valid

  Each element of Valid$_i$ is initialized to 0.
- For each object x such that Valid$_i[x] = 1$, $C_i[x]$ ($C_i[x]$ is processor memory to hold value of object x)

Operations at processor P_i

Write(x, v)::
send[write,v,Valid$_i[*]$] to $PMem_x$;
receive[New_Valid[$*$]] message from $PMem_x$;
Valid$_i[*]$:= New_Valid[$*$]; /$*$ element-wise assignments $*$/
$C_i[x] := v$;

Read(x)::
if valid$_i[x] = 0$ **then**

```
      send[read,Valid_i[*]] message to PMem_x;
      receive[v, New_Valid[*]] message from PMem_x;
      Valid_i[*] := New_Valid[*];
      C_i[x] := v;
    endif
    return C_i[x];
```

Operations at $PMem_x$:

```
  Process [write,v,Valid_i[*]] message from processor i::
    M_x := v;
    Hold_Last_Write_x[*] := 0;
    Hold_Last_Write_x[i] := 1;
    call Check_Validity(Valid_i[*], x);
    Valid_i[x] := 1
    send [Valid_i[*]] to processor i;
      /* send to processor P_i the new value of Valid_i.
         Processor P_i receives the row in New_Valid */

  Process [read,Valid_i[*]] message from processor i::
    Hold_Last_Write_x[i] := 1;
    call Check_Validity(Valid_i[*], x);
    Valid_i[x] := 1
    send [M_x, Valid_i[*]] to processor P_i;

  Local Procedure Check_Validity(VAR Valid_i[*], x)
    for y such that y ≠ x and Valid_i[y] = 1 do
      send [validate, i] to PMem_y
    for y such that y ≠ x and Valid_i[y] = 1 do
      receive [message] from PMem_y
      if message = nack then Valid_i[y] := 0
  end

  Process [validate,j] message from PMem_y::
(a)   if Pmem_x is processing a write request or Hold_Last_Write_x[j] = 0
(b)     then send [nack] to PMem_y
(c)     else send [ack] to PMem_y;
```

Correctness

Theorem 8: The above protocol implements a sequentially consistent distributed shared memory system.

Proof of Theorem 8; First, we show that the protocol enforces the OO-constraint and legality. On the same object, the order between any two write operations and the order between a write operation and an external read operation are determined by the order in which the operations are processed by the associated PMem module. Since an external read operation reads from the most recently executed write operation on the same object, the legality is guaranteed.

Consider the order between a write operation and an internal read operation. Let $r_i(x)$ be an internal read operation. Then, there must be a write operation on x by some processor, say $w_j(x)$, and an external operation by P_i, say g_i, (g_i may be $w_j(x)$) such that

1. g_i is the last external operation by P_i before $r_i(x)$ is executed;
2. when g_i is executed, $w_j(x) = \mathrm{LAST}_x$ and the local memory of P_i contains the value written by $w_j(x)$; and
3. there is no write operation on x that PMem_x starts to execute before PMem_x receives the validate message for g_i.

Let $w_k(x)$ be the next write operation that PMem_x executes following $w_j(x)$ (PMem_x may execute several external read operations in between $w_j(x)$ and $w_k(x)$). Then, $w_j(x) <_{oo} r_i(x)$ and $r_i(x) <_{oo} w_k(x)$. Since $r_i(x)$ reads the value written by $w_j(x)$, the legality is guaranteed.

In the manner described above, any pair of (write, write), (read, write), and (write, read) operations is ordered. Thus, the OO-constraint is enforced. furthermore, the legality is guaranteed.

In this protocol, no component in the system maintains the complete information on all the read and write operation on any one object. Thus, as discussed in Section 4.2, we need to show that the protocol forms acyclic relation of the processor-order and the oo-relation. For operation *op* on object x by processor P_i, let $T_s(op)$ and $T_f(op)$ denote,

- if *op* is a global operation, the times at which PMem_x starts and completes the execution of *op*, respectively;
- if *op* is a local operation, the times at which P_i starts and completes the execution of *op*, respectively;

Then, the following properties hold:

1. If op_1 and op_2 are global operations on the same object x and $op_1 <_{oo} op_2$ (we use $<_1$ to denote this relation), then $T_f(op_1) \leq T_s(op_2)$.

2. If op_1 and op_2 are local operations by the same processor P_i and $op_1 <_{pr} op_2$ (we use $<_2$ to denote this relation), then $T_f(o_1) \leq T_s(o_2)$.

3. If $r_i(x)$ is a local read operation and $w_j(x)$ is a write operation $(i \neq j)$ and $r_i(x) <_{oo} w_j(x)$ (we use $<_3$ to denote this relation), then we cannot determine the real time relation between $r_i(x)$ and $w_j(x)$.

We denote a relation that is either $<_1$ or $<_2$ by $<_{1,2}$.

Suppose that the execution creates cyclic relation. Then, there must exist at least one relation $<_3$ in the cycle. By renaming processors if necessary, the cycle may be represented by $r_1(x_1) <_3 w_2(x_1) <_{1,2} r_3(x_3) <_3 w_4(x_3) <_{1,2} r_5(x_5) <_3 w_6(x_5) <_{1,2} \cdots <_{1,2} r_{n-1}(x_{x-1}) <_3 w_n(x_{n-1}) <_{1,2} r_{n+1}(x_{n+1}) = r_1(x_1)$.

Consider a sequence of operations related only by $<_{1,2}$ in the cycle: that is, a sequence of all the operations in between $w_{i-1}(x_{i-2})$ and $r_i(x_i)$ such that $w_{i-1}(x_{i-2}) <_{1,2} \cdots <_{1,2} r_i(x_i)$ for $3 \leq i \leq n+1$. There must be at least one global operation $g_i(y_i)$ on some object y_i by P_i such that $w_{i-1}(x_{i-2}) <^*_{1,2} g_i(y_i) <_{1,2} r_i(x_i)$; since otherwise, $w_{i-1}(x_{i-2})$ and $r_i(x_i)$ would not be related by $<_{1,2}$. If there are more than one global operation by P_i in between $w_{i-1}(x_{i-2})$ and $r_i(x_i)$, then let $g_i(y_i)$ be the last such operation.

When PMem_{y_i} processes the request for $g_i(y_i)$, it sends a validate message to PMem_{x_i} since processor P_i holds a value for x_i in its local memory; otherwise, local read $r_i(x_i)$ would not occur. Let $T_v(g_i(y_i))$ be the time at which

1. if $y_i \neq x_i$, PMem_{x_i} receives the validate message from PMem_{y_i} on behalf of $g_i(y_i)$; and

2. if $y_i = x_i$, PMem_{y_i} completes the execution of $g_i(y_i)$.

There are two cases to consider:

1. Case 1 $(y_i \neq x_i)$: since PMem_{y_1} did not invalidate the x_i value in P_i's local memory, $T_v(g_i(y_i)) < T_s(w_{i+1}(x_i))$ must hold according to the protocol (refer to step (a)).

2. Case 2 $(y_i = x_i)$: since PMem_{x_1} precesses operations sequentially, and $g_i(x_i) < w_{i+1}(x_i)$, $T_v(g_i(y_i)) < T_s(w_{i+1}(x_i))$ holds.

Furthermore, $T_s(w_{i-1}(x_{i-2})) \leq T_v(g_i(x_i))$ since $w_{i-1}(x_{i-2}) <^*_{1,2} g_i(y_i)$. Thus, we must have $T_v(g_1(y_1)) < T_s(w_2(x_1)) \leq T_v(g_{i_3}(y_3)) < T_s(r_{i_3}(x_3)) \leq \cdots \leq T_v(g_{n-1}(y_{n-1})) < T_s(w_n(x_{n-1})) \leq T_v(g_1(y_1)$ Thus, $T_v(g_1(y_1)) < T_v(g_{n+1}(y_{n+1})) = T_v(g_1(y_1))$. This is a contradiction; therefore, the processor-order and object-order relation created by the execution is acyclic, and the execution represents a history. $\square$