From Causal Consistency to Sequential Consistency in Shared Memory Systems

Michel Raynal[1] and André Schiper[2]

[1] IRISA, Campus de Beaulieu, 35042 Rennes Cédex (France) raynal@irisa.fr
[2] Dpt d'informatique, EPFL, 1015 Lausanne (Switzerland) schiper@di.epfl.ch

Abstract. Sequential consistency and causal consistency constitute two of the main consistency criteria used to define the semantics of accesses in the shared memory model. An execution is sequentially consistent if all processes can agree on a same legal sequential history of all the accesses; if processes perceive distinct legal sequential histories of all the accesses, the execution is only causally consistent (legality means that a read does not get an overwritten value).
This paper studies synchronization constraints that, when obeyed by operations of a given causally consistent execution, make it sequentially consistent. More precisely, the paper introduces the MSC synchronization (mixed synchronization constraint) which generalizes (1) the known DRF (data race free) and CWF (concurrent write free) synchronizations and (2) a new one called CRF (concurrent read free). The MSC synchronization allows for concurrent conflicting operations on a same object, while ensuring sequential consistency; this is particularly interesting in the context of distributed systems (where objects are possibly replicated) to cope with partition failures: conflicting operations in two distinct partitions do not necessarily block processes that issue them (as it is the case of quorum based protocols). Technically, a tag (control type) is associated with each operation, and all operations endowed with the same tag obey the same synchronization constraint.

1 Introduction

For several years the shared memory model has become a pervasive concept in parallel and distributed systems. This is due to the universality of the model: processes distributed over a network and interacting through shared objects (objects distributed over the network, and possibly replicated), fit for example perfectly into this model. Moreover, the shared memory model[3] is the adequate framework for defining consistency criteria: a consistency criterion defines the value returned by every read operation invoked by a process on some object (or some variable). It is important to stress that the definition of a consistency criterion must be independent of the possible existence of multiples copies of

[3] A (logically) shared memory includes simultaneously all the objects with all their implicit mutual and causal dependencies.

objects, and must not rely on a particular protocol implementing the criterion; it must be based on a formal model and be as general as possible to make designers capable to study properties of consistency criteria, and to produce results not bound to particular implementations. As for abstract data types, such a classical approach distinguishes clearly the semantics offered to users from its particular implementations. Several authors have correctly claimed that a memory consistency criterion is a contract between the memory system and application programs [11].

Three main consistency criteria have been proposed in the literature: atomic consistency [9] (also called linearizability [7]), sequential consistency [8] and causal consistency [3]. In all three cases a read operation returns the *last* value assigned to the variable (or written into the object). The three consistency criteria differ however in the definition of the *last* write operation. Atomic consistency is the more restrictive of the three consistency criteria: it requires that all the processes agree on a total order including all the read/write operations that they have issued, and this total order has to respect real time (i.e. if op_1 precedes op_2 in real time, then all the processes have to agree that op_1 has occurred before op_2). With sequential consistency, the processes have also to agree on a total order of their read/write operations, but this total order does not have to respect their real time order. With causal consistency, processes can disagree on the way they totally order concurrent write operations.

Causal consistency includes sequential consistency (i.e. an execution that satisfies sequential consistency also satisfies causal consistency) and sequential consistency includes atomic consistency (i.e. an execution that satisfies atomic consistency also satisfies sequential consistency). If some relationships between atomic and sequential consistencies are well understood (e.g. [5] compares their respective powers), a unifying framework, that would allow for a better understanding of the links between sequential consistency and causal consistency criteria, is still missing. This is precisely the purpose of this paper, which shows that sequential consistency can be obtained from causal consistency by adding some appropriate synchronization constraints (a synchronization constraint orders some pair of operations). This is particularly interesting from methodological and implementation points of view as it means that a family of protocols implementing sequential consistency can be seen as consisting of two independent layers: a basic layer implementing causal consistency and, on top of it, another layer enforcing the chosen synchronization constraints (an interesting consequence of the approach is that only this second layer has to be changed when we want to replace a set of synchronization constraints by another one).

Our work can be seen as a continuation of the work started by Ahamad *et al.* [4]. These authors consider however only two types of synchronization constraints: *Data Race Free* (DRF) synchronization, and *Concurrent Write Free* (CWF) synchronization. We generalize their work in two directions. First we distinguish between two classes of synchronization constraints: (1) the *per object* synchronization constraints, which synchronizes operations on each object independently, and (2) the *inter-object* synchronization constraints, which synchronizes operations on distinct objects. The DRF synchronization fits into the per object synchronization class, and the CWF synchronization fits into the inter-object synchronization class. The per object synchronization class is particularly interesting, as it provides the *locality* property introduced by Herlihy and Wing [7]. The second, and most important contribution of the paper, is the introduction of two new synchronization constraints: the *Concurrent Read Free* (CRF) synchronization, and the *Mixed Synchronization Constraint* (MSC) which combines the DRF, CWF and CRF synchronizations. The MSC synchronization has the nice property to allow conflicting operations on the same object to proceed concurrently. This is particularly interesting in the context of distributed systems where objects are possibly replicated, to cope with partition failures. If sequential consistency is obtained by implementing the MSC synchronization on top of a causally consistent distributed shared memory, then conflicting operations issued from two distinct partitions do not necessarily block processes that issued them (when sequential consistency is ensured by quorum based protocols such blocking always occurs).

The paper is structured as follows. Section 2 formally defines causal and sequential consistencies. Section 3 introduces the two classes of synchronization constraints (per object synchronization and inter-object synchronization), and defines the DRF, CWF and CRF synchronization constraints. The mixed synchronization constraint MSC is introduced in Section 4. Due to page limitation, proof of a lemma, practical impact of the MSC synchronization constraint in the context of distributed systems, and a discussion about other consistency criteria from which sequential consistency can also be obtained, are not included in this paper; the interested reader can consult [12] where these points are addressed.

2 Shared Memory Model

2.1 Notations

We consider a finite set of sequential processes $P_1, \ldots, P_n$ that interact via a finite set X of shared objects. Each object $x \in X$ can be accessed by read and write operations. A write into an object defines a new value for the object; a read allows to obtain a value of the object. A write of value v into object x by process P_i is denoted $w_i(x)v$; similarly a read of x by process P_j is denoted $r_j(x)v$ where v is the value returned by the read operation; op will denote either r (read) or w (write). For simplicity, we assume all values written into an object x are distinct. Moreover, the parameters of an operation are omitted when they are not important. Each object has an initial value; it is assumed that this value has been assigned by an initial fictitious write operation.

2.2 Histories

The *local history* $\widehat{h}_i$ of P_i is the sequence of operations issued by P_i. If $op1$ and $op2$ are issued by P_i and $op1$ is issued first, then we say $op1$ precedes $op2$ in P_i's process-order, which is noted $op1 \rightarrow_i op2$. Let h_i denote the set of operations executed by P_i; the local history $\widehat{h_i}$ is the total order $(h_i, \rightarrow_i)$.

An *execution history* (or simply a history) $\widehat{H}$ of a shared memory system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that :

- $H = \bigcup_i h_i$
- $op1 \rightarrow_H op2$ if :
  i) $\exists\ P_i\ : op1 \rightarrow_i op2$ (in that case, $\rightarrow_H$ is called *process-order* relation),
  or *ii)* $op1 = w_i(x)v$ and $op2 = r_j(x)v$ (in that case $\rightarrow_H$ is called *read-from* relation),
  or *iii)* $\exists op3\ : op1 \rightarrow_H op3$ and $op3 \rightarrow_H op2$.

A read operation $r(x)v$ is *legal* if: $\exists\ w(x)v\ : w(x)v \rightarrow_H r(x)v$ and $\nexists\ op(x)u\ : (u \neq v) \wedge (w(x)v \rightarrow_H op(x)u \rightarrow_H r(x)v)$. A history $\widehat{H}$ is legal if all its read operations are legal[4].

Two operations $op1$ and $op2$ are *concurrent* in $\widehat{H}$ if we have neither $op1 \rightarrow_H op2$ nor $op2 \rightarrow_H op1$.

2.3 Sequential Consistency

Sequential consistency has been proposed by Lamport in 1979 to define a correctness criterion for multiprocessor shared memory systems [8]. A system is sequentially consistent with respect to a multiprocess program, 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*".

This informal definition states that the execution of a program is sequentially consistent if it is equivalent to a sequential execution[5]. More formally, we define sequential consistency in the following way.

Definition. Sequential consistency. A history $\widehat{H} = (H, \rightarrow_H)$ is *sequentially consistent* if it admits a linear extension[6] $\widehat{S}$ in which all reads are legal.

[4] Other definitions of legality [3, 4] eliminate only the possibility of an intervening write $(w(x)u)$ between the writing of some value $(w(x)v)$ and a reading of the same value $(r(x)v)$.

[5] In his definition, Lamport assumes that the *process-order* relation defined by the program (see point *(2)* of the definition) is maintained in the equivalent sequential execution, but not necessarily in the execution itself. As we do not consider programs but only executions, we implicitly assume that the *process-order* relation displayed by the execution histories are the ones specified by the programs which gave rise to these execution histories.

[6] A linear extension of a partial order is a topological sort of this partial order, so it maintains the order of all ordered pairs of the partial order.

As an example let us consider the history $\widehat{H_1}$ (Figure 1)[7]. Each process P_i, (i=1,2), has issued three operations on the shared objects x and y. The write operations $w_1(x)0$ and $w_2(x)1$ are concurrent. It is easy to see that $\widehat{H_1}$ is sequentially consistent by building a legal linear extension including first the operations issued by P_2 and then the ones issued by P_1. It is also easy to see that the history $\widehat{H_2}$ (Figure 2) is not sequentially consistent, as no equivalent legal sequential history can be built.

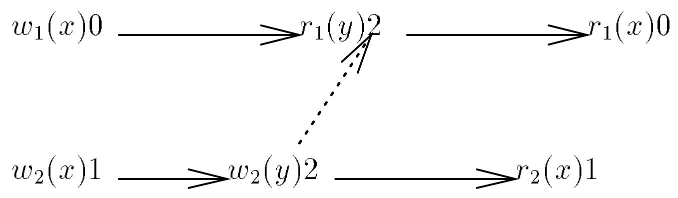

Fig. 1. A sequentially consistent history $\widehat{H_1}$

Various cache-based protocols implementing sequential consistency have been proposed in the context of parallel machines [1, 5, 10]. The protocols presented in [1, 10] allow several read operations and one write operation to concurrently access a same variable (reading of cached values and writing into the main memory) but do not allow concurrent write operations on a same variable. One of the protocols (called *fast write*) presented in [5] allows write operations on a same variable to proceed concurrently. However, these protocols do not assume an underlying causally consistent memory, and thus could not identify the two layers approach and the mixed synchronization constraint given in the paper.

In the context of distributed systems, where each object is supported by several permanent copies, non cache-based protocols implementing sequential consistency have been proposed. Usually these protocols use votes [14] or quorums [6] mechanisms and, consequently, implement actually atomic consistency which is stronger than sequential consistency.

2.4 Causal Consistency

Causal consistency has first been introduced by Ahamad *et al.* in 1991 [3], and then it has been studied by several authors [2, 4]; it defines a consistency criterion weaker than sequential consistency. Causal consistency allows for a wait-free

[7] In all figures, only the edges that are not due to transitivity are indicated (transitivity edges come from *process-order* and *read-from* relations). Moreover, (intra-process) *process-order* edges are denoted by continuous arrows and (inter-process) *read-from* edges by dotted arrows.

implementation of read and write operations in a distributed environment, i.e. causal consistency allows for cheap read/write operations (see [3, 4] for protocols implementing causal consistency).

With sequential consistency, all processes agree on a same legal linear extension ("legal sequential history") $\widehat{S}$. The agreement defined by causal consistency is weaker. Given a history $\widehat{H}$, it is not required that two processes P_i and P_j agree on the same ordering for the write operations which are not ordered in $\widehat{H}$. The reads are however required to be legal.

Definition. Causal consistency. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. $\widehat{H}$ is *causally consistent* if all its read operations are legal.

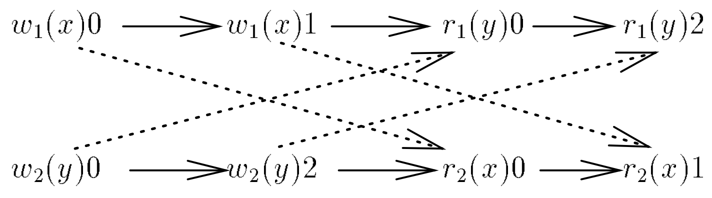

Fig. 2. A causally consistent history $\widehat{H_2}$

In a causally consistent history, all processes see the same partial order on operations but, as processes are sequential, each of them might see a different linear extension of this partial order.

So, in a causally consistent history, no read operation of a process P_i can get a value that, from P_i's point of view, has been overwritten by a more "recent" write. As an example consider history $\widehat{H_2}$ (Figure 2). This history is causally consistent as all its read operations are legal. The history $\widehat{H_3}$ (Figure 3) is not causally consistent as the read operation $r_3(x)1$ issued by P_3 is not legal: $w_1(x)1 \rightarrow_H r_3(x)2 \rightarrow_H r_3(x)1$. Said another way, when P_3 has issued its first read operation on x (namely $r(x)2$), it has got the value 2, and consequently for this process, the value 1 of x has logically been overwritten.

Remark. Our definition of causal consistency is akin to *causal reads* introduced in [2]. Another definition has been proposed in [4]. Let $\widehat{H_i}$ be the sub-history of $\widehat{H}$ from which all read operations not in the local history h_i of P_i have been removed (more formally, $\widehat{H_i}$ is the sub-relation of $\widehat{H}$ induced by the set of all the writes of H and all the reads issued by P_i). History $\widehat{H}$ is causally consistent if, for each process P_i, the associated history $\widehat{H_i}$ admits a legal linear extension $\widehat{S_i}$. From a formal point of view, this definition and ours are not strictly equivalent. Ours is slightly stronger but allows a simple statement of

$$
\begin{tikzcd}
w_1(x)1 \arrow[r] & w_1(y)2 \arrow[d, dotted] & & & \\
w_2(x)2 \arrow[r] & r_2(y)2 \arrow[r] & w_2(z)3 \arrow[d, dotted] & & \\
& & r_3(z)3 \arrow[r] & r_3(x)2 \arrow[r] & r_3(x)1
\end{tikzcd}
$$

Fig. 3. A non causally consistent history $\widehat{H_3}$

causal consistency and, as shown in [12], it corresponds with the definition of causal order [13] in message passing systems. But, it is important to note that the main theorem stated in Section 4 is valid whatever definition of legality is chosen (ours or the one of [4]).

3 Basic Synchronization Constraints

3.1 Adding Synchronization Constraints

As mentioned in the introduction, this paper shows that a causally consistent history whose operations respect some synchronization constraints is a sequentially consistent history $\widehat{H}$: these synchronization constraints ascertain that $\widehat{H}$ admits a legal linear extension $\widehat{S}$. From an implementation point of view this means that a protocol implementing sequential consistency can be seen as consisting of two independent layers: (1) a first layer implementing causal consistency (i.e. basically ensuring the legality of reads), and (2) a second one enforcing the synchronization constraints.

A synchronization constraint orders some pairs of operations. Let $op(x)$ and $op'(y)$ be such a pair. Two classes of constraints can be defined, depending whether or not x and y are the same object.

1. *Per object synchronization.* In this case the synchronization constraint applies to each object x independently of the others. Two operations $op(x)$ and $op'(y)$, such that x and y are distinct, are never synchronized. This type of synchronization is particularly interesting as it provides the *locality* property introduced in [7].
2. *Inter-object synchronization.* In this case the synchronization constraint orders some pairs of operations $op(x)$ and $op'(y)$ on distinct objects x and y. This type of synchronization is more general (as it includes the *per object*

synchronization as a special case). Moreover, as op and op' can each be a read or a write operation, several subclasses of constraints can be envisaged.

We consider below the *data race free* synchronization of the *per object* class, and two types of the *inter-object* class: the *concurrent write free* synchronization, and the *concurrent read free* synchronization. These three basic synchronization constraints can be combined to define what we call the *mixed synchronization constraint*. We show that this mixed synchronization, when added to a protocol implementing causal consistency, leads to sequentially consistent histories.

The synchronization constraints will be defined using the ORD predicate. Two operations satisfy this predicate if they are not concurrent. More formally, let $\widehat{H} = (H, \rightarrow_H)$ be a history, and op, op' be two distinct operations of H:

$$ORD\ (op, op') \stackrel{def}{=} \ (op \rightarrow_H op')\ or\ (op' \rightarrow_H op)$$

3.2 Synchronization Tags

Each read or write operation of a history $\widehat{H}$ is associated with one and only one synchronization tag. The possible tags are *DRF (data race free), CWF (concurrent write free), CRF (concurrent read free)*. A read operation can be tagged *DRF* or *CRF*; a write operation can be tagged *DRF* or *CWF*. Operation op tagged T, and accessing object x, will be noted $op_T(x)$.

Informally, all operations endowed with the same tag T obey the same synchronization constraint (which is called T). Operations with distinct tags are not synchronized. The next two subsections define synchronization constraints based on tagged operations.

3.3 Per Object Synchronization

As indicated previously, the per object synchronization constraint is defined for each object independently of the others: it consists in ordering conflicting operations on each object x (two operations on an object x are conflicting iff one of them is a write). In other words, operations on each object obey the readers-writers synchronization. This constraint is usually called DRF *(data race free)*.

DRF synchronization. The DRF synchronization orders conflicting operations tagged *DRF*. Let $\widehat{H} = (H, \rightarrow_H)$ be a history and x an object. For any two distinct operations $op_{DRF}(x)$ and $op'_{DRF}(x)$, the DRF synchronization ensures that, if at least one of these operations is a write, then $ORD\ (op_{DRF}(x), op'_{DRF}(x))$ holds.

It has been proved in [4] that a causally consistent history $\widehat{H}$ in which (using our terminology) all operations are tagged *DRF* is sequentially consistent. This result is obtained here as a special case of a more general result (see Section 4.3).

3.4 Inter-Object Synchronization

The DRF synchronization does not order operations on distinct objects x and y. We introduce two synchronization constraints that order operations on distinct objects: one called CWF *(concurrent write free)* applies only to write operations, the other, called CRF *(concurrent read free)*, applies only to read operations.

In both definitions hereafter we consider a history $\widehat{H} = (H, \rightarrow_H)$ and a pair of (not necessarily distinct) objects x and y.

CWF synchronization. This synchronization orders write operations tagged *CWF*. For any two distinct write operations, $w_{CWF}(x)$ and $w_{CWF}(y)$, the CWF synchronization ensures $ORD\ (w_{CWF}(x), w_{CWF}(y))$.

CRF synchronization. This synchronization orders read operations tagged *CRF*. For any two distinct read operations, $r_{CRF}(x)$ and $r_{CRF}(y)$, the CRF synchronization ensures $ORD\ (r_{CRF}(x), r_{CRF}(y))$.

It has been shown in [4] that a causally consistent history $\widehat{H}$, in which all write operations are tagged *CWF*, is sequentially consistent. This result will also be proved in Section 4.3 as a special case of our more general result. Notice that the CRF synchronization (with all read operations tagged *CRF*) is not sufficient to ensure sequential consistency out of causal consistency. The CRF synchronization, together with the DRF synchronization for write operations, leads however to this result. This is included in our mixed synchronization constraint.

4 A Mixed Synchronization Constraint

4.1 The Mixed Constraint

The synchronization constraints DRF, CWF and CRF introduced so far can be combined to define a mixed synchronization constraint, noted MSC *(mixed synchronization constraint)*. MSC is a generalization of the previous synchronizations in the sense that it allows for combinations of synchronization constraints (either CWF with DRF, or CRF with DRF). In other words, MSC does not require that the same synchronization tag be attached to every operation of a history. We distinguish between two MSC constraints, MSC_W and MSC_R:

- A history $\widehat{H}$ satisfies the MSC_W constraint if and only if (1) all its operations are tagged either *CWF* or *DRF*, and (2) the *CWF* tagged operations obey the CWF synchronization, and the *DRF* tagged operations obey the DRF synchronization.
  Note that in this case all read operations are necessarily tagged *DRF*, while write operations are tagged *DRF* or *CWF*.
- A history $\widehat{H}$ satisfies the MSC_R constraint if and only if (1) all its operations are tagged either *CRF* or *DRF*, and (2) the *CRF* tagged operations obey the CRF synchronization, and the *DRF* tagged operations obey the DRF synchronization.
  Note that in this case all write operations are necessarily tagged *DRF*, while read operations are tagged *DRF* or *CRF*.

We say that the MSC synchronization is satisfied by a history $\widehat{H}$ if and only if $\widehat{H}$ satisfies either MSC_W or MSC_R. Notice that, in a history $\widehat{H}$ that satisfies the MSC constraint, the tags *CWF* and *CRF* are incompatible. If $w(x)$ is tagged *CWF* in $\widehat{H}$, then all read operations of $\widehat{H}$ are tagged *DRF* (constraint MSC_W). Similarly if a read is tagged *CRF* then all write operations are tagged *DRF* (constraint MSC_R).

In order to understand the MSC_W and MSC_R synchronizations, it is important to understand that they do not require conflicting operations on a given object x be ordered. Consider a history $\widehat{H}$ and the two cases of conflicting operations (namely, read/write conflict and write/write conflict). Consider first read/write conflicting operations. If $\widehat{H}$ satisfies the MSC_W constraint, and if a write operation w on some object x is tagged *CWF*, whereas a read operation r on x is tagged *DRF*, then these two operations are not ordered by the MSC_W synchronization constraint: hence $w_{CWF}(x)$ and $r_{DRF}(x)$ are not synchronized. Consider now the same conflict with $\widehat{H}$ satisfying the MSC_R constraint: if a write operation w on x is tagged *DRF*, whereas a read operation r on x is tagged *CRF*, then these two operations are not ordered by MSC_R: hence $w_{DRF}(x)$ and $r_{CRF}(x)$ are not synchronized.

The same result holds for write/write conflicts. Suppose the MSC_W constraint is satisfied by $\widehat{H}$; if a write operation w on some object x is tagged *DRF* whereas another write operation w' on x is tagged *CWF*, then these operations are not ordered by MSC_W: hence $w_{DRF}(x)$ and $w'_{CWF}(x)$ are not synchronized, i.e. concurrent conflicting writes are allowed!

Remark. To understand that the DRF and the CWF constraints mentioned in [4] are special cases of the MSC constraint, consider the following explanation. First, the DRF synchronization constraint in [4] is obviously a special case of either the MSC_W or of the MSC_R synchronization constraint, in which all the operations are implicitly tagged *DRF*. Second, the CWF synchronization constraint in [4] is a special case of the MSC_W synchronization constraint in which all the write operations are implicitly tagged *CWF* and obey CWF synchronization, and all the read operations are implicitly tagged *DRF* and obey DRF synchronization (in that case the DRF synchronization is actually a *nil* synchronization as it applies only to read operations of $\widehat{H}$ which are never conflicting!).

4.2 Deterministic Read Operations

Section 4.3 proves our main result, namely: a causally consistent history $\widehat{H}$, that satisfies either MSC_W or MSC_R, is sequentially consistent. Because MSC_W allows for concurrent writes on the same object, MSC_W requires an additional deterministic *read rule*, in order for our main result to hold. This rule defines which value has to be returned by a $r(x)$ operation in case the read operation is aware of two concurrent write operations.

Deterministic read rule. Consider $\widehat{H}$ a causally consistent history obeying the MSC_W synchronization, an object x, and two concurrent writes $w_{DRF}(x)u$ and $w_{CWF}(x)v$. Let $r(x)$ be a read operation such that both $r(x)u$ and $r(x)v$ are legal[8]. Then the read operation returns the value written by $w_{CWF}(x)$, i.e. $r(x)$ returns v.

4.3 The MSC Theorem

We prove in this Section our main result relating causal consistency to sequential consistency. We prove the results for MSC_W and for MSC_R together.

Theorem 4.1 *Let $\widehat{H} = (H, \rightarrow_H)$ be a causally consistent history such that (1) either MSC_W or MSC_R is satisfied, and (2) in case of MSC_W each read operation $r(x)$ follows the read rule (Sect. 4.2). Then $\widehat{H}$ is sequentially consistent.*

Preliminary Definitions In order to prove the Theorem 4.1, we introduce the Lemma 4.2. This Lemma is based on two additional relations on the operations of a history $\widehat{H}$: a *logical write-write precedence* relation (denoted $\rightarrow_w$) and a *logical read-write precedence* relation (denoted $\rightarrow_r$).

Logical write-write precedence. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. The write-write precedence relation $\rightarrow_w$ is defined on pairs of write operations, on a same object x, that are concurrent in $\widehat{H}$. By definition, this can happen only if one of them is tagged *DRF* while the other is tagged *CWF* (two writes on a same object, both tagged either *DRF* or *CWF*, are ordered in $\widehat{H}$). The logical write-write precedence relation states that the write tagged *DRF* is logically *before* the write tagged *CWF*.

Definition: Let $w_{CWF}(x)$ and $w_{DRF}(x)$ be two concurrent write operations in $\widehat{H}$. Then we have: $w_{DRF}(x) \rightarrow_w w_{CWF}(x)$. (Moreover $\rightarrow_w$ holds only in these cases.)[9]

Logical read-write precedence. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. The logical read-write precedence relation $\rightarrow_r$ is defined on pairs of read and write operations for each object x.

Definition: Let $w(x)u$, $r(x)u$ and $w(x)v$ be three operations in $\widehat{H}$ such that $w(x)u \rightarrow_H w(x)v$ or $w(x)u \rightarrow_w w(x)v$. Then we have: $r(x)u \rightarrow_r w(x)v$. (Moreover $\rightarrow_r$ holds only in these cases.)[10]

---

[8] This means the read is aware of both writes and there is no intervening operation $op(x)a$, with $a \neq u$ and $a \neq v$, in between $w_{CWF}(x)v$ and $r(x)$ and in between $w_{DRF}(x)u$ and $r(x)$.

[9] So, given $w(x)u$ and $w(x)v$, we necessarily have one of these four relations: $w(x)u \rightarrow_H w(x)v$, or $w(x)u \rightarrow_w w(x)v$, or $w(x)v \rightarrow_H w(x)u$, or $w(x)v \rightarrow_w w(x)u$.

[10] Note that in this case u and v are necessarily distinct.

Lemma 4.2 (Acyclicity) *Let $\widehat{H} = (H, \rightarrow_H)$ be a causally consistent history that satisfies the MSC synchronization constraint. Let $\rightarrow_w$ and $\rightarrow_r$ be the two relations on H defined above. Then the relation $\rightarrow_H \cup \rightarrow_w \cup \rightarrow_r$ defines a partial order on H.*

PROOF. Let $\rightarrow$ be either $\rightarrow_H$ or $\rightarrow_w$ or $\rightarrow_r$, and let $\rightarrow_{w/r}$ be either $\rightarrow_w$ or $\rightarrow_r$. Consider the directed graph whose vertices are the operations of H, and whose edges are the $\rightarrow$ relation. We prove that, for any $n > 0$, there are no (directed) cycle of length n in this graph. The proof is by induction on m, where m is the number of $\rightarrow_{w/r}$ edges in the cycle.

i) **Base step** (m=1). Let $op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_n \rightarrow op_1$ be a simple cycle of length $n > 1$, and assume that one single edge of this cycle is of type $\rightarrow_{w/r}$. We show that this leads to a contradiction. Without loss of generality, let $op_1 \rightarrow op_2$ be the only $\rightarrow_{w/r}$ edge in the above cycle; so, as $\widehat{H}$ is transitive, we have $op_2 \rightarrow_H op_1$. There are two cases to consider, numbered i.1) and i.2).

i.1) $op_1 \rightarrow_r op_2$.
From the definition of $\rightarrow_r$ it follows $op_1 \equiv r(x)u$ and $op_2 \equiv w(x)v$, and either (a) $w(x)u \rightarrow_H w(x)v$ or (b) $w(x)u \rightarrow_w w(x)v$.

(a) $w(x)u \rightarrow_H w(x)v$. As $op_2 \rightarrow_H op_1$, i.e. $w(x)v \rightarrow_H r(x)u$, we have $w(x)u \rightarrow_H w(x)v \rightarrow_H r(x)u$, which means that $r(x)u$ is not legal, in contradiction with the assumption that, because $\widehat{H}$ is a causally consistent history, its read operations are legal.

(b) $w(x)u \rightarrow_w (x)v$. From the definition of $\rightarrow_w$ it follows that $w(x)u$ and $w(x)v$ are concurrent and respectively tagged *DRF* and *CWF*. So we get (1) $w_{CWF}(x)v \rightarrow_H r(x)u$ (because $op_2 \rightarrow_H op_1$), (2) $w_{DRF}(x)u \rightarrow_H r(x)u$ (*read-from* relation), and (3) $w_{CWF}(x)v$ and $w_{DRF}(x)u$ are concurrent. This is in contradiction with the read rule of Section 4.2 (namely $r(x)$ cannot read value u; it reads v or, if it exists, a more recent value v' such that $w(x)v \rightarrow_H w(x)v'$).

i.2) $op_1 \rightarrow_w op_2$.
By definition of $\rightarrow_w$, op_1 and op_2 are two concurrent write operations, but $op_2 \rightarrow_H op_1$ (because $op_1 \rightarrow op_2$ is the only $\rightarrow_{w/r}$ edge in the cycle): a contradiction.

ii) **Induction step** ($m > 1$). Let $op_1 \rightarrow op_2 \rightarrow \ldots \rightarrow op_n \rightarrow op_1$ be a simple cycle of length n, and assume that there is no cycle with m or less than m edges $\rightarrow_{w/r}$ ($m < n$). We prove that there can be no cycle with $m + 1$ edges $\rightarrow_{w/r}$. The proof is again by contradiction.

Assume a cycle with $m + 1$ edges $\rightarrow_{w/r}$, and pick arbitrarily two of these $m + 1$ edges. Without loss of generality let op_1, op_2, op_t ($t \geq 2$), op_{t+1} be the four operations that are the endpoints of the two $\rightarrow_{w/r}$ edges: $op_1 \rightarrow_{w/r} op_2$ and $op_t \rightarrow_{w/r} op_{t+1}$. There are four cases to consider, numbered ii.1) to ii.4).

ii.1) $op_1 \rightarrow_r op_2$ and $op_t \rightarrow_r op_{t+1}$.
By definition of $\rightarrow_r$: op_1, op_t are read operations and op_2, op_{t+1} are write operations.
ii.11). If op_1 and op_2 (or op_t and op_{t+1}) are both tagged *DRF* we have $op_1 \rightarrow_H op_2$ or $op_2 \rightarrow_H op_1$ and consequently there exists a cycle of $\rightarrow_{w/r}$ with less than $m+1$ edges.
ii.12). If both op_1 and op_2 are not tagged *DRF*, and the same holds for op_t and op_{t+1}, then due to the incompatibility of *CRF* and *CWF* tags, either both reads (op_1 and op_t) are tagged *CRF*, or both writes (op_2 and op_{t+1}) are tagged *CWF*. If the reads are tagged *CRF*, then as $\widehat{H}$ satisfies MSC, either $op_1 \rightarrow_H op_t$ or $op_t \rightarrow_H op_1$. If the writes are tagged *CWF*, then as $\widehat{H}$ satisfies MSC, either $op_2 \rightarrow_H op_{t+1}$ or $op_{t+1} \rightarrow_H op_2$. In all of these four cases, we are able to exhibit a cycle with no more than m edges $\rightarrow_{w/r}$, which is in contradiction with the induction hypothesis: if $op_1 \rightarrow_H op_t$ or $op_{t+1} \rightarrow_H op_2$ (respt. $op_t \rightarrow_H op_1$ or $op_2 \rightarrow_H op_{t+1}$) then there is a cycle not including the edge $op_1 \rightarrow_r op_2$ (respt. $op_t \rightarrow_r op_{t+1}$) and so including less than $m+1$ edges $\rightarrow_{w/r}$.

ii.2) $op_1 \rightarrow_r op_2$ and $op_t \rightarrow_w op_{t+1}$.
By the definition of $\rightarrow_r$ and $\rightarrow_w$, op_1 is a read operation, op_2, op_t, op_{t+1} are write operations and op_{t+1} is tagged *CWF*. Because op_{t+1} is tagged *CWF*, there can be no read operations tagged *CRF*, i.e. op_1 is tagged *DRF* and op_2 is tagged *DRF* or *CWF*.
ii.21). If op_2 is tagged *DRF* then we have $op_1 \rightarrow_H op_2$ or $op_2 \rightarrow_H op_1$ and there is a cycle of $\rightarrow_{w/r}$ of less than $m+1$ edges.
ii.22). If op_2 is tagged *CWF* then, as op_{t+1} is also tagged *CWF*, we have either $op_2 \rightarrow_H op_{t+1}$ or $op_{t+1} \rightarrow_H op_2$. In both cases, we can exhibit as previously a cycle with no more than $m \rightarrow_{w/r}$ edges, in contradiction with the induction hypothesis: if $op_2 \rightarrow_H op_{t+1}$ (respt. $op_{t+1} \rightarrow_H op_2$), then there is a cycle not including the edge $op_t \rightarrow_w op_{t+1}$ (respt. $op_1 \rightarrow_r op_2$).

ii.3) $op_1 \rightarrow_w op_2$ and $op_t \rightarrow_r op_{t+1}$.
By renaming op_1 to op_t, op_2 to op_{t+1}, op_t to op_1 and op_{t+1} to op_2, case *ii.3)* becomes identical to *ii.2)*.

ii.4) $op_1 \rightarrow_w op_2$ and $op_t \rightarrow_w op_{t+1}$.
By the definition of $\rightarrow_w$, op_2 and op_{t+1} are write operations tagged *CWF*, i.e. either $op_2 \rightarrow_H op_{t+1}$ or $op_{t+1} \rightarrow_H op_2$. In both cases, we can exhibit as previously a cycle with no more than $m \rightarrow_{w/r}$ edges, in contradiction with the induction hypothesis: if $op_2 \rightarrow_H op_{t+1}$ (respt. $op_{t+1} \rightarrow_H op_2$) there is a cycle not including the edge $op_t \rightarrow_w op_{t+1}$ (respt. the edge $op_1 \rightarrow_w op_2$).
(*end of the proof of Lemma 4.2*)

Proof of the MSC Theorem Lemma 4.2 has showed that $\rightarrow_H \cup \rightarrow_w \cup \rightarrow_r$ defines a (partial) order on H. The following Lemma 4.3 completes the proof of the MSC theorem by showing a legal linear extension $\widehat{S}$ can be constructed by a topological sort of $(H, \rightarrow_H \cup \rightarrow_w \cup \rightarrow_r)$.

Lemma 4.3 (Legality) *A topological enumeration of* $(H, \rightarrow_H \cup \rightarrow_w \cup \rightarrow_r)$ *produces a legal linear extension* $\widehat{S}$ *of* $\widehat{H}$.

Proof. See [12].

5 Conclusion

This paper has studied synchronization constraints that, when obeyed by operations of a given causally consistent execution, make it sequentially consistent. Such an approach is particularly interesting as, from methodological and implementation points of view, it means that a protocol implementing sequential consistency can consist of two independent layers: a basic one implementing causal consistency and, on top of it, another one implementing some synchronization constraints for the operations issued by processes.

The paper introduced the MSC synchronization (mixed synchronization constraint) which generalizes (1) the known DRF (data race free) and CWF (concurrent write free) synchronizations, and (2) a new one called CRF (concurrent read free). A main interest of this constraint lies in the fact it allows concurrent conflicting operations on a same object while ensuring sequential consistency; this is particularly interesting in the the context of distributed systems (where objects are possibly replicated) to cope with partition failures: conflicting operations in two distinct partitions do not necessarily block processes that issue them (as it is the case with quorum based protocols). Technically, a tag (control type) is associated with each operation, and all operations endowed with the same tag obey the same constraint.

The paper has also classified the synchronization constraints in two classes: the *per object* synchronization class, and the *inter-object* synchronization class (which includes the per object synchronization class). This classification allows to better understand linearizability (which has the nice locality property [7]) with respect to sequential consistency: linearizability is obtained by the *per object* synchronization. Finally, while the paper has identified MSC as a sufficient condition to get sequential consistency out of causal consistency, it would be interesting to identify a necessary condition to get sequential consistency out of causal consistency.

Acknowledgements

The presentation has benefited from valuable comments from the anonymous referees. We thank Ambuj Singh for pointing out the difference between the definition of causal consistency used in this paper and the one used in [4].

This work has been supported in part by the Commission of European Communities under ESPRIT Programme BRA 6360 (BROADCAST), by France Telecom under a CNET grant "Cohérence d'objets répartis", and by the "Fonds national suisse" and OFES under contract number 21-32210.91.

References

1. Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Trans. on Prog. Lang. and Systems*, 15(1):182–205, 1993.
2. D. Agrawal, M. Choy, H.V. Leong and A. Singh. Mixed consistency: a model for parallel programming. *In Proc. 13th ACM Symposium on Principles of Dist. Computing*, Los Angeles, pages 101–110, 1994.
3. M. Ahamad, J.E. Burns, P.W. Hutto, and G. Neiger. Causal Memory. In *Proc. 5th Intl. Workshop on Distributed Algorithms (WDAG-5)*, pages 9–30. Springer Verlag, LNCS 579, 1991.
4. M. Ahamad, P.W. Hutto, G. Neiger, J.E. Burns, and P. Kohli. Causal Memory: Definitions, Implementations and Programming. TR GIT-CC-93/55, Georgia Institute of Technology, July 94, 25p.
5. H. Attiya and J.L. Welch. Sequential Consistency versus Linearizability. *ACM Trans. on Comp. Systems*, 12(2):91–122, 1994.
6. H. Garcia-Molina and D. Barbara. How to assign votes in a distributed systems. *Journal of the ACM*, 32(4):841–850, 1985.
7. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Systems*, 12(3):463–492, 1990.
8. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C28(9):690–691, 1979.
9. J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. on Prog. Lang. and Systems*, 8(1):142–153, 1986.
10. M. Mizuno, M. Raynal, and J.Z. Zhou. Sequential Consistency in Distributed Systems. *Proc. Int. Workshop "Theory and Practice in Dist. Systems"*, Dagstuhl, Germany, Springer-verlag LNCS 938 (K.Birman, F. Mattern and A. Schiper Eds), 1994, pp.224-241.
11. D. Mosberger. Memory consitency models. *ACM Operating Systems Review*, 27(1):18–26, 1993.
12. M. Raynal and A. Schiper. *From causal consistency to sequential consistency in shared memory systems.* INRIA Research Report 2557, May 1995, 27 pages.
13. M. Raynal, A. Schiper and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Proc. Letters*, 39:343–350, 1991.
14. R.H. Thomas. A majority consensus approach to concurrency control for multiple copies databases. *ACM Trans. on Database Systems*, 4(2):180–209, 1979.

This article was processed using the LaTeX macro package with LLNCS style