Communication efficient distributed shared memories

Masaaki Mizuno, Gurdip Singh, Michel Raynal, Mitchell L. Neilsen

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1817

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

COMMUNICATION EFFICIENT DISTRIBUTED SHARED MEMORIES

Masaaki MIZUNO
Gurdip SINGH
Michel RAYNAL
Mitchell L. NEILSEN

Décembre 1992

RR-1817

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Publication Interne n°691-Décembre 1992-24 pages-Programme 1

Programme 1, projet ADP (Algorithmes Distribués et aPplications)
novembre 1992

Communication Efficient Distributed Shared Memories

MASAAKI MIZUNO GURDIP SINGH
MICHEL RAYNAL MITCHELL L. NEILSEN

Abstract

Recently, distributed shared memory (DSM) systems have received much attention because such an abstraction simplifies programming. An important class of DSM implementations is one which uses cache memories to improve efficiency. In this paper, we present a cache-consistency protocol which uses considerably less communication as compared to previously proposed protocols. This is realized by maintaining state information and capturing causal relations among read and write operations. We prove that the protocol satisfies a formulation of sequential consistency. We also present several modifications to the protocol and compare the classes of execution histories captured by these protocols and several previously proposed protocols.

Un protocole efficace pour les mémoires partagées réparties

Cet article propose une définition de la cohérence des exécutions réparties (voisine de la "cohérence séquentielle") et présente un algorithme qui gère des copies d'objets en accord avec cette définition. Le protocole obtenu est efficace. L'idée sur laquelle il repose consiste à tenir compte des relations de causalité qui existent entre les lectures et les écritures sur les divers objets et qui définissent leurs dépendances mutuelles.

Communication Efficient Distributed Shared Memories

Masaaki Mizuno* Gurdip Singh†
Michel Raynal‡ Mitchell L. Neilsen§

Abstract

Recently, distributed shared memory (DSM) systems have received much attention because such an abstraction simplifies programming. An important class of DSM implementations is one which uses cache memories to improve efficiency. In this paper, we present a cache-consistency protocol which uses considerably less communication as compared to previously proposed protocols. This is realized by maintaining state information and capturing causal relations among read and write operations. We prove that the protocol satisfies a formulation of sequential consistency. We also present several modifications to the protocol and compare the classes of execution histories captured by these protocols and several previously proposed protocols.

Index Terms : cache-consistency protocols, causal relations, distributed shared memory, sequential consistency

1 Introduction

A fundamental problem in concurrent computing is to provide programmers with a model of shared data which simplifies programming. Efficiency is a major issue

*Dept. of Computing and Info. Sciences, Kansas State University, Manhattan, KS 66506. This work was supported in part by the National Science Foundation under Grant CCR-9201645.

†Dept. of Computing and Info. Sciences, Kansas State University, Manhattan, KS 66506. This work was supported in part by the National Science Foundation under Grant CCR-9211621.

‡IRISA, Campus de Beaulieu, 35042 Rennes-Cédex, FRANCE. This work was supported in part by the ESPRIT project BROADCAST (Number 6360) of the Commission of European Communities.

§Computer Science Department, Oklahoma State University, Stillwater, OK 74078.

in the implementation of shared objects. To maintain appropriate performance level, often multiple copies of an object are maintained, which might cause data consistency problems. Modifications to the individual copies must be handled properly in order to avoid inconsistent system states. Complications may also arise due to the fact that the operations on object may not be instantaneous. The purpose of a *data consistency protocol* is to provide an illusion of serial execution of operations on single copies of objects. Recently, data consistency protocols for implementing shared memories in message passing systems have received much attention [AHJ91] [HA90] [ABD90].

Most proposed distributed shared memory (DSM) implementations are based on cache-coherency protocols for multiprocessor systems. In a cache-based system, multiple copies of an object may reside in local cache of different processors. Traditional cache-consistency protocols have required that all copies of an object be identical at all times. This, however, may be a very stringent consistency requirement. Lamport proposed the notion of *sequential consistency*, which states that a shared memory system is sequentially consistent if each execution of the memory system is equivalent to a sequential execution of the same set of operations [Lam79]. Several protocols have been proposed which use different formulations of this notion and each one has tried to capture a larger set of sequentially consistent executions. Scheurich and Dubois proposed a sufficient condition for sequential consistency which requires that a processor cannot issue a memory access until its previous write operation has been globally performed [SD87]. Brown proposed a protocol for multiprocessor systems which allows invalidation requests to a local cache to be buffered until the shared memory needs to be accessed [Bro90]. Afek, Brown and Merritt also proposed a protocol for multiprocessor systems which captured a larger set of executions by allowing buffering of write requests [ABM89]. The protocols in both [Bro90] and [ABM89] require that for each write operation, all processors be informed that this operation has taken place. Furthermore, both need an atomic broadcast facility to totally order the write operations. For this purpose, they assume a special hardware capability to do atomic broadcast.

Communication in a distributed system may be orders of magnitude more expensive as compared to that in a multiprocessor system. Hence, DSM imple-

mentations requiring frequent atomic broadcasts can be very expensive. Since communication is expensive while memories and high-speed CPUs are becoming cheaper, a strategy appropriate for a distributed system is to reduce communication at the expense of memory and computation time. We show that by maintaining some additional state information, the amount of communication required to maintain consistency can be considerably reduced. In this paper, we propose efficient DSM implementations based on this idea. These protocols do not require atomic broadcasts. The consistency definition used in this paper is a formulation of sequential consistency. Our formulation reflects the intuition used in many protocols that all processors must agree on the ordering of the write operations and each processor should individually schedule its read operations in an appropriate manner.

We first propose a protocol which maintains a data structure to keep track of the recent versions of objects in each cache. When a processor accesses the shared memory, the set of objects in its local cache, whose values have become out of date, are detected using the information stored in this data structure, and these values are invalidated. This protocol allows the same set of sequentially consistent executions as the protocol in [Bro90] but uses less communication. In this protocol, once object x has been invalidated at processor i, no more invalidation requests for x are sent to i until i reads x. In the protocol in [Bro90], an invalidation request for x is sent to i every time x is updated by any other processor.

We then modify the protocol to obtain a protocol which allows the same set of sequentially consistent executions as the protocol in [ABM89] but again uses less communication. In this protocol, only the necessary writes are propagated. On the other hand, in the protocol in [ABM89], all the writes are propagated to all the processors. In the extreme case in which the cache is large enough to hold all the objects, a read operation can always read a value from the local cache in our protocol. In this case, the protocol requires only one round of message exchange between a processor and the shared memory for a write operation and no messages for a read operation.

Although, we have used sequential consistency as the correctness criteria, our method may be applied to data consistency protocols which satisfy other notions

of consistency. For example, we show that the number of invalidations in the protocol proposed in [HA90] to maintain causal consistency may be reduced by using the same information and computation as used by our protocol.

This paper is organized as follows. In the next section, we give a definition of consistency and show that it is equivalent to the notion of consistency used previously. In Section 3, we give a brief review of the protocols in [Bro90] and [ABM89]. In Section 4, we present our protocol and prove that it is correct. Section 5 discusses several modifications to the protocol and compare the performance of these protocols to some existing implementations. We also discuss a data structure which can be used to capture causality information effectively.

2 Definitions

In this section, we give a formulation of consistency and prove that it is equivalent to the notion of consistency used previously. Some of the definitions and notations introduced in this section follow [ABHN91]. A shared memory system consists of a set of processors P and a memory M. Each processor in P may execute a sequence of read and write operations on objects in M. A write operation by processor i on an object 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 i is denoted by $r_i(x)u$, where u is the value of x returned by this operation. We may omit the parameters of an operation when they are not important. For simplicity, we assume that all values written by write operations are distinct.

An *execution history* of a shared memory system is a poset $\widehat{U} = (U, \longrightarrow_U)$. A *poset* is a pair $(U, \longrightarrow_U)$, where U is a set and $\longrightarrow_U$ is an irreflexive and antisymmetric partial order. In the following, we give some definitions:

- We say that an execution history $\widehat{U} = (U, \longrightarrow_U)$ is *processor-ordered* if the operations of each processor in U are totally ordered by $\longrightarrow_U$.
- An execution history $\widehat{S} = (S, \longrightarrow_S)$ is a *sequential history* iff it is processor-ordered and $\longrightarrow_S$ is a total order.
- A sequential history $\widehat{S} = (S, \longrightarrow_S)$ is *legal* if for every read operation $r(x)v$ in S, there exists a write operation $w(x)v$ such that $w(x)v \longrightarrow_S r(x)v$

and there does not exist a write operation $w(x)u$ such that $w(x)v \longrightarrow_S w(x)u \longrightarrow_S r(x)v$.

- A *restriction* of $\widehat{V} = (V, \longrightarrow_V)$ to the set U, where $U \subseteq V$, is an execution history $\widehat{U} = (U, \longrightarrow_U)$ such that for any operations o and o' in U, $o \longrightarrow_U o'$ iff $o \longrightarrow_V o'$.
- We define $\widehat{U}|i$ to be the restriction of history $\widehat{U}$ to the set of operations performed by i.
- Two execution histories $\widehat{S}$ and $\widehat{U}$ are *equivalent* if for every processor i, $\widehat{S}|i = \widehat{U}|i$.
- Two execution histories $\widehat{S}$ and $\widehat{U}$ are *result-equivalent* if every read operation returns the same value, every write operation writes the same value, and the final write operations on each object are the same in both $\widehat{S}$ and $\widehat{U}$.

  For example, $w_1(x)1$, $w_2(x)2$, $r_2(x)1$ and $w_1(x)1$, $r_2(x)1$, $w_2(x)2$ are result-equivalent but not equivalent.
- $\widehat{U} = (U, \longrightarrow_U)$ *respects* $\widehat{V} = (V, \longrightarrow_V)$ iff $V \subseteq U$ and for any two operations o and o' in V, if $o \longrightarrow_V o'$ then $o \longrightarrow_U o'$.

Sequential consistency was proposed by Lamport to formulate a correctness criterion for multiprocessor shared-memory system [Lam79]. A multiprocessor system is sequentially consistent if *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.* We present a formulation of this definition in the following:

Definition 1: A memory M is *sequentially consistent* iff for each of its execution histories $\widehat{H}$, there exists a legal sequential execution history $\widehat{WR} = (WR, \longrightarrow_{WR})$, where WR is the set of all read and write operations in $\widehat{H}$, such that $\widehat{H}$ and $\widehat{WR}$ are result-equivalent.

Sequential consistency, like the notion of view serializability, only requires that the results of the operations be the same. Thus, an execution history

$\widehat{H}$ in which operations are not invoked in the order specified by the program could be sequentially consistent. For example, consider a program in which processor 1 writes a value 1 on x while processor 2 reads x and then writes 1 on y. Then, $w_1(x)1, w_2(y)1, r_2(x)1$ is sequentially consistent execution since it is result-equivalent to the following sequential execution: $w_1(x)1, r_2(x)1, w_2(y)1$. We want to consider systems in which a processor issues operations in the order specified in the program. For this purpose, we give the following (more restrictive) definition of consistency:

Definition 2: A memory M is *consistent* iff for each of its execution histories $\widehat{H}$, there exists a legal sequential execution history $\widehat{WR} = (WR, \longrightarrow_{WR})$, where WR is the set of all read and write operations in $\widehat{H}$, such that $\widehat{H}$ and $\widehat{WR}$ are equivalent.

Most cache-consistency protocols implementing sequential consistency impose a total ordering on write operations. The order viewed by all processors is consistent with this total ordering. Furthermore, each processor independently tries to schedule its read operations in such a manner that legality and processor order are maintained. We give an alternative definition of sequential consistency which follows this intuition:

Definition 3: A memory M is *sequentially consistent* iff for each of its execution histories $\widehat{H}$, there exists a sequential history $\widehat{W} = (W, \longrightarrow_W)$, where W is the set of all writes in $\widehat{H}$, such that the following holds for each processor i:

(a) Let $WR_i = W \cup R_i$, where R_i is the set of read operations performed by i in $\widehat{H}$. Then, there exists a legal sequential history $\widehat{WR_i} = (WR_i, \longrightarrow_{WR_i})$ such that $\widehat{WR_i}$ respects $\widehat{W}$ and $\widehat{H}|i$ and $\widehat{WR_i}|i$ are result-equivalent.

Definition 4: A memory M is *consistent* iff for each of its execution histories $\widehat{H}$, there exists a sequential history $\widehat{W} = (W, \longrightarrow_W)$, where W is the set of all writes in $\widehat{H}$, such that the following holds for each processor i:

(a) Let $WR_i = W \cup R_i$, where R_i is the set of read operations performed by i in $\widehat{H}$. Then, there exists a legal sequential history $\widehat{WR_i} = (WR_i, \longrightarrow_{WR_i})$ such that $\widehat{WR_i}$ respects $\widehat{W}$ and $\widehat{H}|i = \widehat{WR_i}|i$.

Lemma 1: Definitions 2 and 4 are equivalent.

Proof of Lemma 1:
(2 $\Rightarrow$ 4) Let $\widehat{H}$ be an execution of M such that there exists a legal sequential history $\widehat{WR} = (WR, \longrightarrow_{WR})$, where WR is the set of all operations in $\widehat{H}$, such that $\widehat{H}$ and $\widehat{WR}$ are equivalent. Let $\widehat{W} = (W, \longrightarrow_W)$ be a poset, where W is the set of write operations in $\widehat{WR}$, and $\widehat{W}$ is the restriction of $\widehat{WR}$ to the set W. Since $\widehat{W}$ is a restriction of $\widehat{WR}$, $\widehat{W}$ is also sequential. We will show that $\widehat{W}$ satisfies condition (a) of Definition 4. Let $\widehat{WR_i}$ be the restriction of $\widehat{WR}$ to WR_i. First, we have to show that $\widehat{WR_i}$ is a legal sequential history. Since $\widehat{WR}$ is sequential, $\widehat{WR_i}$ is also sequential. Removing read operations from a legal sequential history will not invalidate the legality of the rest of the history. Since $\widehat{WR_i}$ is obtained from $\widehat{WR}$ by removing read operations of all processors except i, $\widehat{WR_i}$ will also be legal. Hence, $\widehat{WR_i}$ is a legal sequential history and respects $\widehat{W}$. Since $\widehat{WR_i}$ is a restriction of $\widehat{WR}$ and contains all operations performed by i, $\widehat{WR_i}|i = \widehat{WR}|i$. Since $\widehat{H}|i = \widehat{WR}|i$, we have that $\widehat{WR_i}|i = \widehat{H}|i$. Hence, condition (a) of Definition 4 is satisfied.
(4 $\Rightarrow$ 2) Let $\widehat{H}$ be an execution of M such that there exists a sequential history $\widehat{W} = (W, \longrightarrow_W)$ such that for each i, there exists a legal sequential history $\widehat{WR_i} = (WR_i, \longrightarrow_{WR_i})$ such that $\widehat{WR_i}$ respects $\widehat{W}$ and $\widehat{WR_i}|i = \widehat{H}|i$. Assume that, for simplicity, that processors are labeled $1, \ldots, n$. Let $WM_k = W \cup R_1 \cup R_2 \cup \cdots \cup R_k$, where $k \geq 1$. We will prove the following claim:
Claim 1: There exists a legal sequential history $\widehat{WM_k} = (WM_k, \longrightarrow_{WM_k})$, such that for all $j \leq k$, $\widehat{H}|j = \widehat{WM_k}|j$.
Then, from Claim 1, $\widehat{WM_n}$ is a legal sequential history containing all read and write operations in $\widehat{H}$ and for all i, $\widehat{H}|i = \widehat{WM_n}|i$. Hence, it satisfies the conditions of Definition 2.

Proof of Claim 1: We will prove it by induction on k.
Base case: $k = 1$. Let $\widehat{WM_1} = \widehat{WR_1}$. Then, $\widehat{WM_1}$ satisfies the conditions of the claim.
Induction Step: Assume that there exists a legal sequential history $\widehat{WM_k} = (WM_k, \longrightarrow_{WM_k})$, such that $\widehat{WM_k}$ respects $\widehat{W}$ and for each $j \leq k$, $\widehat{H}|j = \widehat{WM_k}|j$. We construct $\widehat{WM_{k+1}}$ using $\widehat{WM_k}$ and $\widehat{WR_{k+1}}$ as follows: Let o and o' be op-

erations in $\widehat{WM_{k+1}}$. If both operations belong to $\widehat{WM_k}$ then $o \longrightarrow_{WM_{k+1}} o'$ if $o \longrightarrow_{WM_k} o'$. This will impose a total order on all operations belonging to set WM_k. We will now add the read operations in R_{k+1} to this total order. We will add the read operations in the order in which they appear in $\widehat{WR_{k+1}}$. If o is a read operation, $r(x)v$, in R_{k+1} and o' is the operation immediately preceding $r(x)v$ in $\widehat{WR_{k+1}}$ and o'' is the operation immediately following o' in $\widehat{WM_{k+1}}$ (according to the ordering introduced so far) then $o' \longrightarrow_{WM_{k+1}} o$ and $o \longrightarrow_{WM_{k+1}} o''$. ¿From the construction, it is clear that $\widehat{WM_{k+1}}$ is sequential. We must now show that $\widehat{WM_{k+1}}$ is legal and that $\widehat{WM_{k+1}}|i = \widehat{H}|i$. Adding a read operation will not violate the legality of other read operations. Furthermore, read operations in R_{k+1} are added in such a manner that the value read corresponds to the immediately preceding write operation on the same object. Hence, since $\widehat{WR_{k+1}}$ and $\widehat{WM_k}$ are legal, $\widehat{WM_{k+1}}$ is legal. It can be shown that $\widehat{WR_{k+1}}$ is a restriction of $\widehat{WM_{k+1}}$. Since $\widehat{WR_{k+1}}|(k+1) = \widehat{H}|(k+1)$, we have that for all $i \leq k+1$, $\widehat{WM_{k+1}}|i = \widehat{H}|i$. □

3 Related Work

In this section, we will review some implementations of consistent memories proposed in the literature.

3.1 Brown's Algorithm

Brown presented an asynchronous multicache algorithm which satisfies a more restricted form of consistency than the one given in the previous section since his definition of consistency applies only to a specific architecture [Bro90]. In this algorithm, the real-time ordering on write operations is preserved; that is, if o and o' are write operations such that o completes before o' starts then o is ordered before o' by the protocol. The implementation assumes a system model which consists of a shared memory and processors. All the processors and the shared memory are connected by a fully connected bus. Each processor has local cache memory and a special queue which stores invalidation requests for cache objects.

When a processor writes a value to an object, then as a single atomic action, it updates the value of the object in both its local cache and the shared memory and places an invalidation request for the object in the queues in all other processors. To read an object, it searches its local cache. If the object is found, it returns the value in the object. Otherwise, it reads the value from the shared memory. The invalidation requests are buffered at a processor. When the processor accesses the shared memory (either to read or to write), all the invalidation requests in the queue are performed and the queue is emptied.

For each write operation, a processor needs to send a message to the shared memory and an invalidation request to each of the processors. Furthermore, this operation must be atomic. The implementation assumes the existence of a special hardware capability for this purpose. A read operation at a processor requires one round of message exchange between the processor and the shared memory if the object is not found in its local cache.

3.2 The Lazy Cache Algorithm

Afek, Brown, and Merritt presented a lazy cache algorithm in [ABM89] which satisfies a consistency definition that is less restrictive than that of [Bro90]. In this protocol, the order in which the write operations are performed may not reflect the real-time order. This algorithm assumes a similar architecture as in [Bro90] except that there are two queues, IN_i and OUT_i, associated with the local cache of processor i. When a processor i writes a value to an object, it places a write request consisting of the value and the object name in OUT_i. As a separate atomic operation, the request at the head of OUT_i is placed in the IN queue of all processors, and the shared memory is updated. As another separate operation, a processor dequeues the request at the head of IN queue and updates it local cache based on the request. Thus, this algorithm queues updates to data objects instead of invalidation requests in the IN queues and allows a processor to issue a write operation before the previous one has been dequeued from the OUT queue. The ordering of write operations depends on the order in which requests are dequeued from the OUT queue of various processors, which may not be the same as the real-time order.

In this protocol, for each write operation at a processor, one message is sent to the shared memory. The processor also broadcasts the request (when the request reaches the head of OUT) to every processor. Furthermore, this broadcast must be atomic. Since a write operation does not have to wait for the previous write operation to be dequeued from the OUT queue, this protocol will incur less latency as compared to [Bro90]. A read operation requires access to the shared memory only if the object is not found in the local cache.

4 A Communication Efficient Implementation

In this section, we will describe a data consistency protocol which maintains additional information in the shared memory in order to reduce the amount of communication. The protocol allows the same set of sequentially consistent execution as [Bro90]. We also show that the protocol satisfies our notion of consistency.

4.1 Overview of the Protocol

We assume a system which consists of a shared memory module, residing at a network processor, and multiple processors. The shared memory module, $SMem$, stores a set of data objects. Each processor can reliably communicate with $SMem$. Each processor has a local cache memory, which stores a subset of the objects stored in the shared memory.

A processor performs a write operation on an object by sending a message to $SMem$ and updating its local cache. It then awaits response indicating completion of the operation. When a processor makes a request to read an object, it looks up its local cache. If the object is found in the local cache, then its value is returned. Otherwise, it reads the object value from $SMem$, updates its local cache, and returns the value.

Since multiple copies on the object are being maintained, some of the values in a local cache may become out of date with respect to the write operations performed on $SMem$. If a processor reads such values, consistency might be violated. Thus, processor must invalidate these out of date cache values. We

require that when a processor i reads an object from the cache, the value returned must be at least as recent as the contents of $SMem$ when the last operation by the same processor was performed on $SMem$. The invalidations are detected as follows:

> $SMem$ keeps track of the most recent write operation on each object. In addition, it keeps track of the values in the local cache of each processor. When $SMem$ receives a request from a processor, this request is causally ordered after all the previous write operations. At this point, using the information present, $SMem$ determines the values in the local cache of the processor which are out of date, and notifies the processor of these out of date values along with the response to the operation. The processor invalidates the corresponding location on receiving the message.

4.2 Description of the Protocol

We assume that a manager process exists on the shared memory and on each of the processors. The manager process on a processor processes requests to read or write objects from the user processes running on the processor. It also communicates with the manager process on the shared memory. The manager process on the the shared memory processes request messages from processors to read or write objects.

In order to capture causal relations among read and write operations, the shared memory manager assigns *version numbers* to all the write operations on each object. The version number assigned to a write operation is one greater than the previous write operation on the same object (the first write on an object is assigned the version number one). Then, each write may be identified by its version number and the object, such as "version 4 of object x." Since we assume that each write operation writes a different value, a value in an object can be identified by the version number of the write operation which wrote the value. Thus, we also identify a value as "value of version 4 of object x."

The data structures maintained in $SMem$ are:

- Memory area $M[Object\_Range]$.

- Two-dimensional array $Cache\_Ver[Processor\_Range, Object\_Range]$, where $Cache\_Ver[p, o]$ stores the version number of object o which processor p holds in its cache memory. The value is 0 if the corresponding cache location has been invalidated. Each element of $Cache\_Ver$ is initialized to zero.
- One-dimensional array $Causal[Object\_Range]$, where $Causal[o]$ keeps the version number of the most recent write on object o in the shared memory. Each element of Causal is initialized to zero, Note that $Causal[o] = Max_{p=1}^{Process\_Range} Cache\_Ver[p, o]$ (therefore, this array need not be explicitly maintained. It is included for ease of exposition).

The data structures maintained in processor i are:

- A set of valid cache objects $Valid_i$. $Valid_i$ is initialized to be an empty set.
- Cache memory area $C_i[x]$ for each object $x \in Valid_i$.

Operations by the manager process at processor i are described below:

Write(x,v)::
send $[write, x, v]$ message to $SMem$;
receive $[Invalid]$ message from $SMem$;
$Valid_i := (Valid_i - Invalid) \cup \{x\}$; $C_i[x] := v$;

Read(x)::
if $x \notin Valid_i$
then
send $[read, x]$ message to $SMem$;
receive $[v, Invalid]$ message from $SMem$;
$Valid_i := (Valid_i - Invalid) \cup \{x\}$; $C_i[x] := v$;
return $C_i[x]$;

Operations by the manager process at $SMem$ are described below:

Local Procedure Invalidate (**var** $Invalid$) ::
Invalid := $\emptyset$;

for each y **in** $Object\_Range, y \neq x$ **do**
if $Cache\_Ver[i,y] \neq 0$ **and** $Cache\_Ver[i,y] < Causal[y]$
/* check if object y in cache at processor i is out of date
relative to the current version of y */
then $Invalid := Invalid \cup \{y\}$; $Cache\_Ver[i,y] := 0$;

Receive $[write, x, v]$ **message from processor** i::
$M[x] := v$; increment($Causal[x]$); $Cache\_Ver[i,x] = Causal[x]$;
Invalidate($Invalid$); return $[Invalid]$ to processor i;

Receive $[read, x]$ **message from processor** i::
$Cache\_Ver[i,x] := Causal[x]$;
Invalidate($Invalid$); return $[M[x], Invalid]$ to processor i;

Each operation must be performed indivisibly. We assume that a write request to an object is processed before the first read request on the object is issued. The proof of the following theorem is given in the Appendix.

Theorem 1: The implementation is consistent; that is, it satisfies Definition 4.

5 Discussions

5.1 Piggy-Backing Updates

In the protocol described in Section 4.2, the shared memory manager informs a processor to invalidate certain values. Consider, for example, the case where processor i is informed to invalidate the value on an object x. If a subsequent read operation at i reads an object x then it will have to communicate with the shared memory to obtain the current value. We can optimize the protocol to avoid this communication as follows:

• When shared memory manager notifies a processor to invalidate x, it also sends the current value of x along with this message. After the processor updates x in its local cache, a subsequent read operation on x can read the value from the local cache if the value of x is not invalidated (due to local memory management). In the extreme case in which the size of the cache is equal to that of the main

memory, cache values will never be invalidated. In this case, a processor needs to communicate with the shared memory only when it writes a value to the shared memory.

Our original protocol and the above modified protocol preserve the real-time ordering on write operations and allow the same set of sequentially consistent executions as allowed by the protocol in [Bro90]. In addition, the following modification is possible to introduce concurrency among write operations within a processor:

• A write operation does not have to wait for a response from $SMem$ indicating completion of the operation. However, when a read operation is performed by a processor then it must wait for responses to all previous write operations performed by i.

By introducing the above modification, the order in which the write operations are performed may no longer reflect the real-time order, and the resulting protocol allows the same set of sequentially consistent executions as allowed by the protocol in [ABM89].

5.2 Performance

In our protocols, each write operation requires one round of message exchange between the processor and the shared memory. A read operation at a processor also requires one round of message exchange between the processor and the shared memory if the object is not found in its local cache. Thus, our protocols require considerably less number of messages as compared to the protocols in both [Bro90] and [ABM89]. In particular, we do not require an atomic broadcast capability. In the protocol in Section 4, if x has been invalidated at processor i then no more invalidation requests for x are sent to i until i reads x (since $Cache\_Ver[i, x]$ will remain 0 until i reads x and hence, x will not be included in *Invalid*). This property is especially useful if the degree of sharing is less. Similarly, in the protocol described in Section 5.1, a write operation does not have to be propagated to all processors. For example, if i performs several write operations on x and j subsequently reads x then only the most recent write operation on x has to be propagated to j.

Our protocols, however, require maintenance of some extra data structures to keep track of version numbers known to processors. Recent technological advances, however, have made large memories and high speed CPUs available at reasonable costs. On the other hand, communication is expensive in distributed systems. In particular, as compared to tightly coupled systems, communication may be orders of magnitude more expensive. Thus, we believe that our protocols are appropriate for distributed systems.

For the protocol presented in Section 4, it is possible to reduce the amount of state information and the amount of computation at $SMem$ at the expense of larger message size. We can distribute the information stored in array $Cache\_Ver$ among the processors; that is, each processor i maintains the row associated with i of $Cache\_Ver$. When the shared memory receives a request from processor i, it sends the entire array $Causal$ back to processor i. Processor i invalidates its cache based on $Causal$ and the row of $Cache\_Ver$. This approach increases the bit complexity of messages but decreases the computational load of the shared memory manager.

5.3 Ordering between Read and Write Operations

When the shared memory manager processes a write or read request, it orders this request after all the previous write operations. For a write operation, this ordering is established in order to form a sequential history of write operation, $(W, \rightarrow_W)$. On the other hand, for a read operation, the ordering is introduced to satisfy *legality*. However, legality only requires an ordering be established with respect to the previous write operation on the same object. For example, let $\ldots, w_a(x), w_b(y), w_c(z)$ be a sequence of write operations which have been processed by the shared memory. Assume that none of the operations are issued by processor i. Suppose that the shared memory now receives a read request on object x from processor i (denoted $r_i(x)$). In order to satisfy legality, the read operation should return the value written by $w_a(x)$ and therefore, $r_i(x)$ should be ordered after $w_a(x)$. In this case, any one of the following ordering will be sufficient :

1. $r_i(x) \Rightarrow w_b(y)$

2. $w_b(y) \Rightarrow r_i(x)$ and $r_i(x) \Rightarrow w_c(z)$

3. $w_c(z) \Rightarrow r_i(x)$

The algorithm described in Section 4.2 orders the operation as indicated in (3). As a result, if the local cache of i contains y and z then these values have to be invalidated. On the other hand, if ordering was done as indicated by (1), then these invalidations can be avoided. For this purpose, we have to maintain another two-dimensional array $Last\_Write[Object\_Range, Object\_Range]$, where $Last\_Write[o, v]$ stores the version number of object v at the time when o was last written. When the memory manager processes a write request on object o, it updates row o of $Last\_Write$ to $Causal$. When the memory manager processes a read request on object o, $Invalid$ is computed using row o of $Last\_Write$ instead of $Causal$.

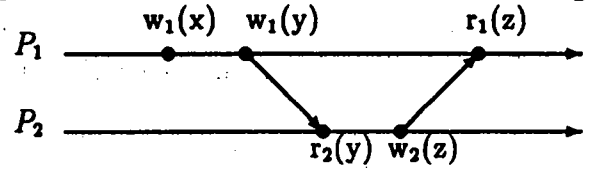
5.4 Capturing Causal Relations

In order to efficiently implement cache-consistency protocols, it is important to capture causal relations between write operations. Vector clocks are often used to identify causal relations among events in message passing systems [Fid91] [Mat89]. A vector clock consists of a set of local clocks, one for each processor. Vector clocks in message passing systems keep causal information between events of the same type. In implementing shared memory, firstly we have to order operations on each object. For this purpose, we can employ a separate vector clock for each object. Secondly, we also have to capture causal relations among operations on different objects. Thus, it is natural to use a vector of vector clocks. Therefore, we find that a two-dimensional array $CR2[Processor\_Range, Object\_Range]$ is appropriate for shared memory systems. When processor i writes on object x, the values in $CR2$ are interpreted as follows: $CR2[j, y] = k$ indicates that all the write operations up to and including the k^{th} write by processor j on object y causally precede this write on x. Now, we will demonstrate that $CR2$ generalizes the data structures used in our protocol and the protocol proposed by Ahamad et al. [HA90] in order to capture causal relations between write and read operations.

We can view array *Causal* used by the protocol described in Section 4 as $CR2$ being collapsed into a one-dimensional array, such that $Causal[x] = \sum_{k \in Processor\_Range} CR2[k, x]$, for $x \in Object\_Range$. This is possible by assigning system wide version numbers to all the writes on an object, instead of processor-level version numbers.

Ahamad et al. proposed a protocol which maintains a one-dimensional array VT [*Processor_Range*] [HA90]. We can view VT as $CR2$ being collapsed into one-dimensional array such that $VT[j] = \sum_{k \in Object\_Range} CR2[j, k]$, for $j \in Processor\_Range$.

In order to compare *Causal* and VT, consider the following example:



When P1 performs $r1(z)$, VT only records that "P2 has updated *some* object" but does not know which object. On the other hand, if *Causal* is used, it captures that "*some* processor has updated z, not x or y." Therefore, *Causal* is more appropriate to capture causal relations between operations and may allow a protocol to minimize unnecessary invalidations as compared to VT.

6 Conclusion

We presented an efficient cache-consistency protocol for distributed shared memory systems. The protocol enforces a formulation of sequential consistency. The amount of communication required by the protocol is considerably less as compared to previously proposed protocols. Furthermore, it does not require atomic broadcast. This is realized by maintaining, within the shared memory module, the state information of local cache memories and causal information among the write operations. Based on the information, the set of out of date cache values is computed. Thus, the protocol requires additional memory space and computation. Since communication is expensive while memories are becoming cheaper and CPUs are becoming more powerful, our strategy to trade communication for memory and computation is appropriate for distributed systems.

We discussed several modifications to the protocol. If the cache is large enough to hold all the objects, the protocol may be modified to require only one round of message exchange between a processor and the shared memory module for a write operation and no messages for a read operation. We also compared the classes of execution histories captured by our protocols and several previously proposed protocols.

References

[ABD90] Attiya, H., Bar-Noy, A., and Dolev, D. Sharing memory robustly in message-passing systems. In *Proceedings of the ACM Sym. on Principles of Distributed Computing*, pages 363–376, 1990.

[ABHN91] Ahamad, M., Burns, J., Hutto, P., and Neiger, G. Causal memory. Technical Report GIT-ICS-91/42, Georgia Institute of Technology, Atlanta, Georgia, 1991.

[ABM89] Afek, Y., Brown, G., and Merritt, M. A lazy cache algorithm. In *Proceedings of the ACM Sym. Parallel Algorithms and Architectures*, pages 209–222, 1989.

[AHJ91] Ahamad, M., Hutto, P., and John, R. Implementing and programming causal distributed shared memory. In *Proceedings of the IEEE Int'l Conference on Distributed Computing Systems*, pages 274–281, 1991.

[Bro90] Brown, G. Asynchronous multicaches. *Distributed Computing*, 4:31–36, 1990.

[Fid91] C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[HA90] Hutto, P. and Ahamad, M. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the IEEE Int'l Conference on Distributed Computing Systems*, pages 302–309, 1990.

[Lam79] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Bonas, France, 1989. North-Holland.

[SD87] 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.

Appendix: Proof of Theorem 1

Let $\widehat{H}$ be an execution history of the protocol described in the previous section. In order to show the above implementation is consistent, we have to show that

(A) We can construct a sequential history $\widehat{W} = (W, \longrightarrow_W)$, where W is the set of all write operations in $\widehat{H}$,

(B) For each processor i, we can construct a legal sequential history $\widehat{WR_i} = (WR_i, \longrightarrow_{WR_i})$, where $WR_i = W \cup R_i$ and R_i is the set of read operations performed by i in $\widehat{H}$, such that $\widehat{WR_i}$ respects $\widehat{W}$ and $\widehat{H}|i = \widehat{WR_i}|i$.

Let $\widehat{W} = (W, \longrightarrow_W)$ be a history such that if o and o' are operations in W then $o \longrightarrow_W o'$ if o is processed before o' by the shared memory. Since the shared memory processes the write operation in a serial manner, $\widehat{W} = (W, \longrightarrow_W)$ is a sequential history. We will now show (B). We will construct a legal sequential history $\widehat{WR_i} = (WR_i, \longrightarrow_{WR_i})$ as follows:

(1) For any two operations o and o' in WR_i which access the shared memory, $o \longrightarrow_{WR_i} o'$ if o is processed before o'.

(2) For any two operations o_i^1 and o_i^2 performed by processor i, where o_i^1 is performed before o_i^2, $o_i^1 \longrightarrow_{WR_i} o_i^2$.

(3) Let o_i be an operation performed by processor i which accesses the shared memory. Let $< r_{i_1}, r_{i_2}, \cdots, r_{i_n} >$ be a sequence of consecutive read operations performed by processor i which access only the local cache and r_{i_1} directly follows o_i in S_i (due to ordering imposed by (2)). Let o_j be an operation by any processor which accesses the shared memory and o_j directly follows o_i in S_i (due the ordering enforced by (1)). Then $r_{i_n} \longrightarrow_{WR_i} o_j$.
Note: If a history is finite, we add an imaginary operation in the history which accesses the shared memory after the execution terminates.

Then, from (1) and the fact that all operations in W access shared memory, we have that $\widehat{WR_i}$ respects $\widehat{W}$, and from (2), $\widehat{H}|i = \widehat{WR_i}|i$. To prove that $\widehat{WR_i}$ is legal, we will use the following claims:

Claim 2: Let o_i be a write operation performed by processor i which accesses *SMem*. If $o_i = w(x)v$ or $o_i = r(x)v$ then $C_i[x] = v$ on the completion of this

operation.

Claim 3: Let o_i be an operation performed by processor i which accesses $SMem$ such that o_i is not an operation on x. Let $w(x)v$ be an operation such that $w(x)v \longrightarrow_{WR_i} o_i$ and there does not exist $w(x)u$ such that $w(x)v \longrightarrow_{WR_i} w(x)u \longrightarrow_{WR_i} o_i$. Then, on the completion of o_i, either $C_i[x] = v$ or $x \notin Valid_i$.

Using Claim 2 and 3, we will now prove that S_i is legal. Assume that, on the contrary, S_i is not legal. Then there must exist a read operation $r(x)v$ such that $w(x)v \longrightarrow_{WR_i} w(x)u \longrightarrow_{WR_i} r(x)v$ and there does not exist $w(x)s$ ordered in between $w(x)u$ and $r(x)v$ according to $\longrightarrow_{WR_i}$. We have the following cases to consider:
Case 1: $r(x)$ access the shared memory. Then, from Claim 2, $C_i[x] \neq v$. Hence, $r(x)$ cannot return value v.
Case 2: $r(x)$ reads from local cache. Let o_i be the last operation by i which accesses the shared memory. We have the following cases to consider:
(a) $w(x)v \longrightarrow_{WR_i} w(x)u \longrightarrow_{WR_i} o_i$. If o_i is an operation on x then $C_i[x] \neq v$ (from Claim 2). If o_i is not an operation on x then $C_i[x] = u$ or $x \notin Valid_i$ (from Claim 3). Since $r(x)$ did not access shared memory, $x \in Valid_i$. Therefore, $C_i[x] = u$. Hence, in either case, $r(x)$ cannot return the value v.
(b) $o_i \longrightarrow_{WR_i} w(x)u$. In this case, rule (3) will order $r(x)v$ in between o_i and $w(x)u$. Hence, $r(x)v$ cannot follow $w(x)u$.

Proof of Claim 2: Obvious from the protocol description.

Proof of Claim 3: We know that $Cache\_Ver[i,x] \leq Causal[x]$. Let $Causal[x] = l$ on the completion of $w(x)v$. Then, $Causal[x] = l$ before the procedure **Invalidate** is invoked while processing o_i (since there are no write operations on x in between $w(x)v$ and o_i and $Causal[x]$ is only incremented when a write operation on x is performed). Let $Cache\_Ver[i,x] = k$. If $k = 0$ then $x \notin Valid_i$. In this case, x is not included in $Invalid$ since it has been already invalidated. If $0 < k < l$ then **Invalidate** will include x in $Invalid$ and therefore, $C_i[x]$ will be invalidated when o_i completes. If $k = l$ then there must exist a read previous operation which read the l^{th} version of x. In this case, $C_i[x] = v$. Hence, when o_i completes, either $C_i[x] = v$ or $x \notin VALID_i$.

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 683 TARGET TRACKING BY VISUAL SERVOING
Aristide S. SANTOS, François CHAUMETTE
Octobre 1992, 50 pages.

PI 684 UNE DESCRIPTION LINEAIRE COMPLETE ET IRREDONDANTE DU POLYTOPE ASSOCIE AU PROBLEME DU VOYAGEUR DE COMMERCE ASYMETRIQUE A 6 SOMMETS
Reinhardt EULER, Hervé LE VERGE
Octobre 1992, 30 pages.

PI 685 MISE EN CORRESPONDANCE DE SEGMENTS DANS UNE SEQUENCE D'IMAGES PAR UNE APPROCHE LOCALE
Samia BOUKIR, Patrick BOUTHEMY, François CHAUMETTE, Didier JUVIN
Octobre 1992, 30 pages.

PI 686 FROM EQUATIONS TO HARDWARE. TOWARDS THE SYSTEMATIC MAPPING OF ALGORITHMS ONTO PARALLEL ARCHITECTURES
François CHAROT, Patrice FRISON, Eric GAUTRIN, Dominique LAVENIER, Patrice QUINTON, Charles WAGNER
Octobre 1992, 18 pages.

PI 687 THE COMPILATION OF PROLOG and its Execution with MALI
Pascal BRISSET, Olivier RIDOUX
Novembre 1992, 90 pages.

PI 688 GENERALISATION DE L'ANALYSE FACTORIELLE MULTIPLE A L'ETUDE DES TABLEAUX DE FREQUENCE ET COMPARAISON AVEC L'ANALYSE CANONIQUE DES CORRESPONDANCES
Lila ABDESSEMED, Brigitte ESCOFIER
Novembre 1992, 34 pages.

PI 689 OVERVIEW OF THE KOAN PROGRAMMING ENVIRONMENT FOR THE iPSC/2 AND PERFORMANCE EVALUATION OF THE BECAUSE TEST PROGRAM 2.5.1..
François BODIN, Thierry PRIOL
Décembre 1992, 16 pages.

PI 690 CONCURRENT PROGRAMMING NOTATIONS IN THE OBJECT-ORIENTED LANGUAGE ARCHE
Marc BENVENISTE, Valérie ISSARNY
Décembre 1992, 34 pages.

PI 691 COMMUNICATION EFFICIENT DISTRIBUTED SHARED MEMORIES
Masaaki MIZUNO, Michel RAYNAL, Gurdip SINGH, Mitchell L. NEILSEN
Décembre 1992, 24 pages.

ISSN 0249-6399