A Parametrized Algorithm that Implements Sequential, Causal, and Cache Memory Consistency*

Ernesto Jiménez[1] Antonio Fernández[2] Vicente Cholvi[3]

[1] Universidad Politécnica de Madrid, 28031 Madrid, Spain, ernes@eui.upm.es
[2] Universidad Rey Juan Carlos, 28933 Móstoles, Spain, afernandez@acm.org
[3] Universitat Jaume I, 12071 Castellón, Spain, vcholvi@inf.uji.es

Abstract. In this paper we present an algorithm that can be used to implement sequential, causal, or cache consistency in distributed shared memory (DSM) systems. For this purpose it has a parameter that allows to choose the consistency model to be implemented. As far as we know, this is the first algorithm proposed that implements the cache consistency model.
In our algorithm, when implementing causal and cache consistency all read and write operations are executed locally (i.e., are *fast*). It is known that no sequential algorithm has only fast memory operations. However, in our algorithm, when implementing sequential consistency all write operations and some read operations are fast.

1 Introduction

Distributed shared memory (DSM) is a well-known mechanism for inter-process communication in a distributed environment. One of the main properties of a DSM system is the semantic of its read and write operations, which is commonly denoted as its *consistency model.* Two of the most popular consistency models proposed are the sequential [6] and causal consistency models. The former is close to what programmers expect from a shared memory, while the later is considered to be powerful enough to allow easy programming, but at the same time allows for inexpensive implementations. As a consequence, a number of algorithms have been proposed in the literature implementing sequential [1, 3, 4] and causal consistency [2, 7, 8]. A third consistency model proposed in the literature is the cache model [5], for which to our knowledge, no algorithm has been proposed.

An interesting property of any algorithm implementing a consistency model is the time a memory operation takes. If a memory operation does not need to wait for any communication to finish, and can be completed based only on the local state of the process that issued it, it is said to be *fast*, which is a very desirable feature. All the above mentioned algorithms for causal consistency are fast.

* This work is partially supported by the CICYT under grant TEL99-0582 and the Comunidad Autónoma de Madrid under grant CAM-07T/00112/1998.

However, in [3], Attiya and Welch have shown that no sequential consistency algorithm can guarantee the fast executions of all its operations. This impossibility result restricts the efficiency of any implementation of sequential consistency.

In general, in order to increase concurrency, most DSM protocols support *replication* of data. With replication, there are copies (replicas) of the same variables in the local memories of several processes of the system, which allows these processes to use the variables simultaneously. However, in order to guarantee the consistency of the shared memory, the system must control the replicas when the variables are updated. That control can be done by either *invalidating* outdated replicas or by *propagating* the new variable values to update the replicas. When propagation is used, a replica of the whole shared memory is usually kept in each process.

Our Results In this paper, we introduce a parametrized algorithm that implements sequential, causal, and cache consistencies. The algorithm has the convenience that we can change the model it implements with a single parameter. Furthermore, as far as we know, this is the first algorithm proposed to implement cache consistency.

Our algorithm uses propagation and replication. With this algorithm, each process in the system has a copy of the complete set of variables that constitute the shared memory. A write operation is propagated from the process that issued it to the rest of processes so they can apply it locally. However, write operations are not propagated immediately. The algorithm works on a cyclic turn fashion, with each process broadcasting one message in its turn. The latest write operation on each variable issued by the process since it sent the previous message are included in this single message. This scheme allows a very simple control of the number of messages in the network due to this algorithm, since only one message is sent periodically by each process. Furthermore, it compares very favorably with most algorithms that use propagation (e.g. [1, 3]), since they send one message for each write operation issued, while ours does not propagate some write operations, and the rest is grouped in single messages, with the corresponding savings in bandwidth (by avoiding the overhead of many messages).

When implementing causal and cache consistency, all the operations in our algorithm are fast. When implementing sequential consistency, from the results in [3] is derived the impossibility of having all the memory operations fast. However, in our algorithm, all write operations are fast. Furthermore, all read operations are fast unless a specific condition on the process issuing the read operation occurs. This condition is the following: since the latest time it sent a message, the process has not issued write operations on the variable being read and has issued write operations on other variables. An example of these non fast read operations is one on a variable x issued by a process that previously (but after it sent the previous message) issued a write operation on a different variable y and did not issue a write operation on x. A read operation in which this condition happens has to block until the process that issued it has the turn.

It is very interesting to compare our sequential consistency algorithm with the sequential cache coherence algorithm proposed by Afek et al. [1]. First, as we said, we do not send each variable update in a single message as they do and we are able to control the number of messages sent. However, both algorithms have some features in common. In their algorithm, like in ours, all write operations are fast. Also read operations are also fast unless a given condition occurs. In their case, a read operation blocks if there are local write operations still not applied in the shared memory. Compared with our condition, we also block a read operation if there are local write operations still not propagated, but only if the variable to be read was not written in one of these write operations, which makes our condition slightly more interesting. However, the algorithm in [1] could be simply modified to use the technique we present here and, hence, have the same condition as ours. It is worth to mention here that the time a read operation is blocked with our algorithm is bounded if the communication delay is bounded (since it only depends on the number of processes and the maximum communication delay). In the algorithm of Afek et al. a blocked read operation may need to wait for an arbitrary number of write operations to be applied.

Another aspect in which both algorithms differ has to do with the model assumed. In the model of [1] there is a communication medium among all processes (and with the shared memory) that guarantees total order among concurrent writes. In our case, we do not have such a device, and must enforce the order of the operations with the cyclic turn technique described above.

2 Definitions

In this paper we assume a distributed system that consists of a set of n processes (each uniquely identified by a value in the range $0...n-1$). Processes do not fail and are connected by a reliable message passing subsystem. These processes use their local memory and the message passing subsystem to implement a shared memory abstraction. This abstraction is accessed through read and write operations on variables of the memory. The execution of these memory operations must be consistent with the particular *memory consistency model*.

Each memory operation acts on a named variable and has an associated value. A write operation by process p, denoted $w_p(x)v$, stores the value v in the variable x. Similarly, a read operation, denoted $r_p(x)v$, reports to the process p that issued it that v is stored in the variable x.

In this paper we present an algorithm that uses replication and propagation. We assume each process holds a copy of the whole set of variables in the shared memory. We use x_p to denote the local copy of variable x in process p. Different copies of the same variable can hold different values at the same time.

A *computation* α is a sequence of read and write operations (usually observed in some execution of the memory algorithm). We denote with $\xrightarrow{\alpha}$ the order in which the operations in α happen.

Definition 1 (Legal Computation). *A computation* α *is* legal *if* $\forall op = r(x)v \in \alpha, \exists op' = w(x)v \in \alpha : op' \xrightarrow{\alpha} op$ *and* $\nexists op'' = w(x)u : op' \xrightarrow{\alpha} op'' \xrightarrow{\alpha} op$.

Definition 2 (Causal Order). *Let $op, op' \in \alpha$, op precedes op' in the* causal order *($op \prec^{\alpha}_{cau} op'$) if*
1. op and op' are operations from the same process and $op \overset{\alpha}{\rightarrow} op'$,
2. $op = w_p(x)v$ and $op' = r_q(x)v$, or
3. $\exists op'' \in \alpha : op \prec^{\alpha}_{cau} op'' \prec^{\alpha}_{cau} op'$

We denote by α_p the computation obtained by removing from α all read operations issued by processes other than p. We also denote by $\alpha(x)$ the computation obtained by removing from α all the operations on variables other than x.

Definition 3 (Causal Computation). *We say that a computation α is* causal *if, for each process p, the computation α_p has a causal view β_p which is a permutation of α_p that preserves the causal order $\prec^{\alpha}_{cau}$, and such that each prefix of β_p is legal.*

Definition 4 (Sequential Computation). *We say that a computation α is* sequential *if it has a sequential view β, which is a permutation of α such that operations from the same process appear in the same order as in α, and each prefix of β is legal.*

Definition 5 (Cache Computation). *We say that computation α is* cache *if, for each variable x, the computation $\alpha(x)$ has a cache view $\beta(x)$, which is a permutation of $\alpha(x)$ such that operations from the same process appear in the same order as in $\alpha(x)$, and each prefix of $\beta(x)$ is legal.*

From these definitions, we say that an algorithm implements *causal, sequential, or cache consistency* if all the computations observed in its executions are causal, sequential, or cache, respectively.

3 The Algorithm

In this section we present the parametrized algorithm $\mathcal{A}$ that implements causal, cache and sequential consistency. Figure 1 presents the algorithm in detail. As it can be noted, it is run with a parameter *model*, which defines the consistency model that the algorithm must implement. Hence, the parameter must take one of the values *causal*, *sequential*, or *cache*.

In Figure 1 it can be seen that all write operations are fast. When a process p issues a write operation $w_p(x)v$, the algorithm changes the local copy of variable x (which we denote by x_p) to the value v, includes the pair (x, v) in a local set of variable updates (which we call $updates_p$), and returns control. This set $updates_p$ will later be asynchronously propagated to the rest of processes. Note that, if a pair with the variable x was already in $updates_p$, it is removed before inserting the new pair, since it does not need to be propagated anymore.

Processes propagate their respective sets $updates_p$ in a cyclic turn fashion, following the order of their identifiers. To maintain the turn, each process p uses a variable $turn_p$ which contains the identifier of the process whose set must be propagated next (from p's view). When $turn_p = p$, process p itself uses

```
Initialization ::
begin
  $turn_p \leftarrow 0$
  $updates_p \leftarrow \emptyset$
end

$r_p(x)$ :: atomic function
begin
  if ($model = sequential$) and
     ($updates_p \neq \emptyset$) and
     ($(x,\cdot) \notin updates_p$) then
       wait until $turn_p = p$
  return($x_p$)
end

$send\_updates()$ :: atomic task activated
whenever $turn_p = p$
begin
  /* send to all processes, except itself */
  broadcast($updates_p$)
  $updates_p \leftarrow \emptyset$
  $turn_p \leftarrow (turn_p + 1)\ mod\ n$
end

$w_p(x)v$ :: atomic function
begin
  $x_p \leftarrow v$
  if ($(x,\cdot) \in updates_p$) then
      remove $(x,\cdot)$ from $updates_p$
  include $(x,v)$ in $updates_p$
end

$apply\_updates()$ :: atomic task activated
whenever $turn_p = q$, $p \neq q$, and the set
$updates_q$ from process $q$ is in the receiving
buffer of process $p$
begin
  take $updates_q$ from the receiving buffer
  while $updates_q \neq \emptyset$ do
      extract $(x,v)$ from $updates_q$
      if ($model = causal$) or
         ($(x,\cdot) \notin updates_p$) then
           $x_p \leftarrow v$
  $turn_p \leftarrow (turn_p + 1)\ mod\ n$
end
```

Fig. 1. The algorithm $\mathcal{A}(model)$ for process p. It is invoked with the parameter *model*, which defines the consistency model that it must implement.

the communication channels among processes to send to the rest of processes its local set of updates $updates_p$. This is done in the algorithm with a generic broadcast call, which could be simply implemented with $n-1$ point-to-point messages sends if the underlying message passing subsystem does not provide a more appropriate communication primitive. All this is done by the atomic task $send\_updates()$, which also empties the set $updates_p$. The message sent implicitly passes the turn to the next process in order $(turn_p + 1)\ mod\ n$.

The atomic task $apply\_updates()$ is the one in charge of applying the updates received from another process q in $updates_q$. This task is activated whenever $turn_p = q$ and the set $updates_q$ is in the receiving buffer of process p. Note that, when implementing sequential and cache consistency, after a local write operation has been performed in some variable, this task will stop applying the write operations on the same variable from other processes. That allows the system to "view" those write operations as if they were overwritten with the value written by the local process.

Read operations are always fast with causal and cache consistencies. When implementing sequential consistency, a read operation $r_p(x)u$ is fast unless $updates_p$ does not contain a pair with variable x but contains a pair with a variable different from x. That is, the read operation is not fast only if, since the latest time it held the turn, process p has not issued write operations on x and has issued write operations on other variables. In this case, and only in this case, it is necessary to delay such a read operation until $turn_p = p$ for the next time. Note that this condition is the same as the activation condition of task *send_updates*(). We enforce a blocked read operation to have priority over the task *send_updates*(). Hence, when $turn_p = p$, a blocked read operation finishes before *send_updates*() is executed.

We have labeled the code of the read operation as atomic because we do not want it to be executed while the variable $updates_p$ is manipulated by some other task. However, if the read operation blocks, other tasks are free to access the algorithm variables. In particular, it is necessary that *apply_updates*() updates the variable $turn_p$ for the operation to finish eventually.

We omit the proof that this algorithm implements causal, sequential, and cache consistencies (depending on the value of the parameter *model*) due to space limitation.

4 Future Work

We are currently studying how to extend this work so that the system can switch on the fly from one consistency model to another. We are also evaluating the efficiency of our algorithm in real applications, and comparing it with that of other proposed algorithms.

References

1. Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.
2. M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, August 1995.
3. H. Attiya and J.L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.
4. Alan Fekete, M. Frans Kaashoek, and Nancy Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 45(1):35–69, January 1998.
5. J.R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.
6. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.
7. R. Prakash, M. Raynal, and M. Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *Journal of Parallel and Distributed Computing*, 41:190–204, 1997.
8. M. Raynal and M. Ahamad. Exploiting write semantics in implementing partially replicated causal objects. In *Proceedings of the 6th EUROMICRO Conference on Parallel and Distributed Computing*, pages 157–163, Feb 1998.