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

Ernesto Jiménez
Universidad Politécnica de Madrid, 28031 Madrid, Spain
ernes@eui.upm.es

Antonio Fernández
Universidad Rey Juan Carlos, 28933 Móstoles, Spain
afernandez@acm.org

Vicente Cholvi
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 cache coherence.

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.*

The algorithm uses propagation and full replication, where values written by a process are propagated to the rest of processes. It works in a cyclic turn fashion, with each process of the DSM system broadcasting one message in its turn. The values written by the process are sent in the message (instead of sending one message for each write operation), but unnecessary values are excluded. All this allows to control the amount of message traffic due to the algorithm.

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*.

However, while the semantic of read and write operations in sequential programs is clear, the situation is different for concurrent accesses to shared variables. This is more evident if the shared memory is not centralized but distributed among a number of processors, i.e. we have distributed shared memory (DSM). Two of the most popular consistency models proposed are the sequential [9] and causal consistency models. The former is close to what programmers expect from a shared memory, while the later is powerful enough to allow easy programming but allows inexpensive implementations. As a consequence, a number of algorithms have been proposed in the literature implementing sequential [2, 4, 6] and causal consistency [3, 11, 12]. A third, less popular, consistency model proposed in the literature is the cache model [7]. To our knowledge, no algorithm to implement the cache consistency model has been proposed.

An interesting property of any algorithm implementing a consistency model is how long can a memory operation take. 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 that the operation is *fast*, which is a very desirable feature. An algorithm is fast if all its operations are fast. All the above mentioned algorithms for causal consistency are fast. However, in [4], Attiya and Welch have shown that no sequential algorithm can guarantee the fast executions of all its operations. This impossibility result restricts the efficiency of any implementation of sequential consistency.

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

Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (EUROMICRO-PDP'02)
1066-6192/02 $17.00 © 2002 IEEE

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 consistency. 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 is included in this single message. This scheme allows a very simple control of the load of messages in the network due to this protocol, since only one message is sent periodically by each process. Furthermore, it compares very favorably with most algorithms that use propagation (e.g. [2, 4]), 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 [4] 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 algorithm with the sequential cache coherence algorithm proposed by Afek et al. [2]. 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 algorithm more interesting. However, the algorithm in [2] 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 delays are 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 appplied.

A second aspect in which both algorithms differ has to do with the model assumed. In the model of [2] there is a communication medium among all processes (and with the shared memory) that guarantees total order among concurrent write operations. 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.

The rest of the paper is organized as follows. In Section 2 we introduce basic definitions. In Section 3 we introduce the algorithm we propose. In Sections 4, 5, and 6 we prove the correctness of our algorithm. In Section 7 we provide an analysis of the complexity of our algorithm and in Section 8 we present our concluding remarks.

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 system 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. To simplify the analysis, we assume that a given value is written at most once in any given variable and that the initial values of the variables are set by using write operations.

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. When needed, 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 $\stackrel{\alpha}{\rightarrow}$ the order in which the operations in α happen. Abusing the notation, we will also use $seq_1 \stackrel{\alpha}{\rightarrow} seq_2$, where seq_1 and seq_2 are sequences of operations, to denote that all the operations in seq_1 precede all the operations in seq_2 in computation α.

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

Definition 2 (Causal Order) *Let op and $op' \in \alpha$, op precedes op' in the* causal order *($op \prec^{\alpha}_{cau} op'$) if:*
1. op and op' are operations from the same process and $op \stackrel{\alpha}{\rightarrow} op'$,
2. $op = w(x)v$ and $op' = r(x)v$, 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 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 a 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 in $\alpha(x)$, and each prefix of $\beta(x)$ is legal.*

Definition 6 (Sequential, Causal or Cache Algorithm) *An algorithm implements* sequential, causal *or* cache consistency *if all the computations observed in its executions are sequential, causal, 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 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 by sending $n - 1$ point-to-point messages if the underlying message passing subsystem does not provide a more appropiate 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 $(turn_p + 1) \bmod n$ (see Figure 2).

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 writes as if they were overwritten with the write value issued 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 $updates_p$ contains a

```
Initialization ::
begin
  turn_p ← 0
  updates_p ← ∅
end

w_p(x)v :: atomic function
begin
  x_p ← v
  if ((x, ·) ∈ updates_p) then
     remove (x, ·) from updates_p
  include (x, v) in updates_p
end

r_p(x) :: atomic function
begin
  if (model = sequential) and (updates_p ≠ ∅) and
((x, ·) ∉ 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 ← ∅
  turn_p ← (turn_p + 1) mod n
end

apply_updates() :: atomic task activated whenever
turn_p = q, p ≠ 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 ≠ ∅ do
     extract (x, v) from updates_q
     if (model = causal) or ((x, ·) ∉ updates_p) then
        x_p ← v
  turn_p ← (turn_p + 1) mod n
end
```

Figure 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.

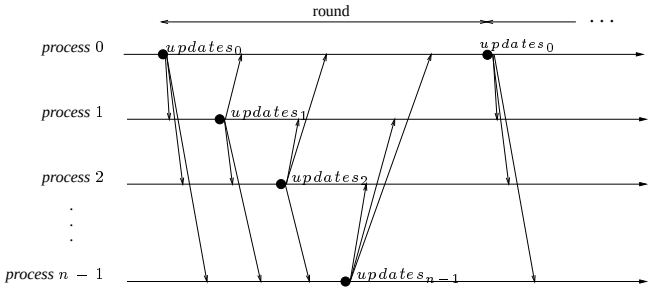

Figure 2. cyclic turn fashion.

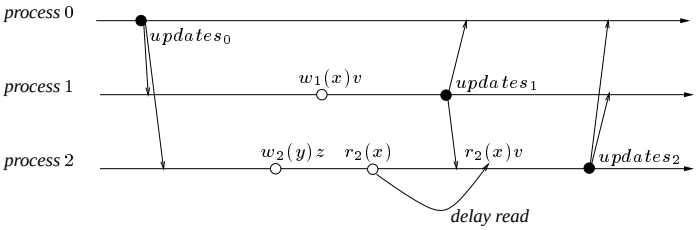

Figure 3. An example of "non fast" read operation.

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 (see Fig. 3). Note that this condition is the same as the condition to execute the 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 finished 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.

4. $\mathcal{A}(causal)$ Implements Causal Consistency

In this section, we show that the algorithm $\mathcal{A}$, executed with the parameter $causal$, implements causal consistency. In the rest of this section we assume that α is a computation observed in the execution of the algorithm $\mathcal{A}(causal)$ and α_p is the sequence obtained by removing from α all read operations issued by processes other than p.

Definition 7 *The i^{th}* writes *of process q, denoted $writes_q^i$, $i > 0$, is the subsequence of α that contains all the write operations issued by process q after $send\_updates()$ is executed for the i^{th} time, and before it is executed for the $i+1^{st}$ time.*

For simplicity, we assume that no write operation is issued by any process before it executes $send\_updates()$ for the first time. This allows us to consider $writes_p^0$ as the empty sequence. Observe in $\mathcal{A}(causal)$ that the $i + 1^{st}$ set $updates_q$ broadcasted by process q contains, for each variable, the last (if any) write operation in $writes_q^i$ on that variable.

Then, we construct a permutation β_p of α_p as follows. Given the sequence of operations issued by p, in the order they are issued, we insert the sequence $writes_q^i$ in the point of the sequence in which $apply\_updates()$ is executed with the set $updates_q$ for the $i + 1^{st}$ time, for all $q \neq p$ and $i \geq 0$. Since the execution of $apply\_updates()$ is atomic, it does not overlap any of the operations issued by p, and the placement of every sequence $writes_q^i$ can be easily found.

We show in the following lemmas that β_p is in fact a causal view of α_p. Some proofs are omitted due to space limitation.

Lemma 1 *Each prefix of β_p preserves the causal order $\prec_{ca\,u}^{\alpha}$*

Lemma 2 *Each prefix of β_p is legal.*

Proof: Let γ_p be a prefix of β_p. Let us consider a read operation $op = r_p(x)v$ in γ_p. From the algorithm $\mathcal{A}(causal)$ it can be seen that the read operation returns the value of the local copy x_p of x, and that this value is only set by a local write or the execution of $apply\_updates()$ with a set $updates_q$ from some process $q \neq p$. In either case, the corresponding write operation $op' = w(x)v$ must precede op in γ_p. Furthermore, the existence of another write operation $op'' = w(x)u$ in β_p between op' and op is not possible, since it would mean that op would have found the value u in x_p, instead of the value v. ■

Lemma 3 *β_p is a causal view of α_p.*

Proof: From Lemma 1, each prefix of β_p preserves the causal order $\prec_{ca\,u}^{\alpha}$, and from Lemma 2, each prefix of β_p is legal. Hence, from Definition 3, β_p is a causal view of α_p. ■

Theorem 1 *The algorithm $\mathcal{A}(causal)$ implements causal consistency.*

Proof: From Lemma 3, every computation α observed in the execution of the algorithm $\mathcal{A}(causal)$ has a causal view β_p of α_p, $\forall p$. Hence, from Definition 3, every α is causal, and, from Definition 6, the algorithm $\mathcal{A}(causal)$ is causal. ■

5. $\mathcal{A}(sequential)$ Implements Sequential Consistency

In this section, we show that the algorithm $\mathcal{A}$, executed with the parameter $sequential$, implements sequential consistency. In the rest of this section we assume that α is a computation observed in the execution of the algorithm $\mathcal{A}(sequential)$. Also, any time reference in this section has to do with the time in the execution in which α was observed. We first introduce some definitions of subsequences of α. In all of them their operations follow the same order as they have in α.

Definition 8 *The i^{th}* iteration *of process p, denoted it_p^i, $i > 0$, is the subsequence of α that contains all the operations issued by process p after $send\_updates()$ is executed for the i^{th} time, and before it is executed for the $i + 1^{st}$ time.*

Observe that any operation in it_p^i finishes before $send\_updates()$ is executed for the $i + 1^{st}$ time, since all write and most read operations are fast, and we assume that blocked read operations have priority over the execution of $send\_updates()$.

Definition 9 *The i^{th}* iteration tail *of process p, denoted $tail_p^i$, is the subsequence of it_p^i that includes all the operations from the first write operation (included) until the end of it_p^i. If it_p^i does not contain any write operation, $tail_p^i$ is the empty sequence.*

Observe that all write operations in it_p^i are in $tail_p^i$. Furthermore, it is easy to check in $\mathcal{A}(sequential)$ that the $i + 1^{st}$ set $updates_p$ broadcasted by process p contains, for each variable, the last (if any) write operation in $tail_p^i$.

Definition 10 *The i^{th}* iteration header *of process p, denoted $head_p^i$, is the subsequence of it_p^i that contains all the operations in it_p^i that are not in $tail_p^i$.*

It should be clear that all the operations in $head_p^i$ precede all the operations in $tail_p^i$. We use now the time instants sets received from other processes are applied to partition the sequence $head_p^i$. Note that between the i^{th} and the $i + 1^{st}$ execution of $send\_updates()$ by p (which defines the operations that are in it_p^i, and hence in $head_p^i$) the task $apply\_updates()$ is executed $n - 1$ times, with sets from processes $(p+1)\ mod\ n, ..., n-1, 0, ..., (p-1)\ mod\ n$ (in this order).

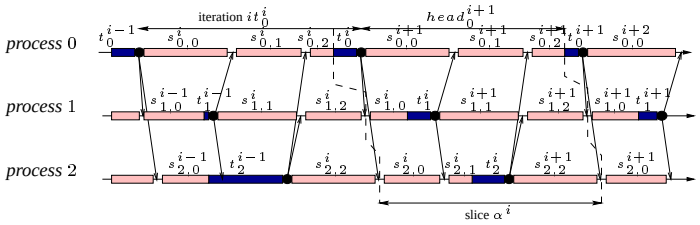

Figure 4. Iterations and slices. We have abbreviated $tail$ with t and $subhead$ with s.

Definition 11 *The* iteration subheader q *of* $head_p^i$*, denoted* $subhead_{p,q}^i$*, is the subsequence of* $head_p^i$ *that contains the following operations.*

- *If* $q = p$*, then* $subhead_{p,p}^i$ *contains all the operations issued before* $apply\_updates()$ *is executed with the set* $updates_{(p+1) \bmod n}$*.*
- *If* $q = (p-1) \bmod n$*, then* $subhead_{p,q}^i$ *contains all the operations issued after* $apply\_updates()$ *is executed with the set* $updates_q$*.*
- *Otherwise,* $subhead_{p,q}^i$ *contains all the operations issued after* $apply\_updates(mess_q)$ *is executed with the set* $updates_q$ *and before it is executed with the set* $updates_{(q+1) \bmod n}$*.*

Clearly, if the first write operation in it_p^i *is issued before* $apply\_updates()$ *is executed with the set* $updates_q$*, then* $subhead_{p,q}^i$ *is the empty sequence (see* it_2^{i-1} *in Fig. 4).*

To simplify the notation and the analysis, we assume that no operation is issued by any process before it executes $send\_updates()$ for the first time. This allows us to define, for any p and q, the sequences it_p^0, $tail_p^0$, $head_p^0$, and $subhead_{p,q}^0$ as empty sequences.

With these definitions, we divide now the computation α in slices (see Fig. 4).

Definition 12 *The* i^{th} slice *of computation* α*, denoted* α^i*,* $i \geq 0$*, is the subsequence of* α *formed by the sequences* $tail_p^i, \forall p$*,* $subhead_{p,q}^i, \forall p, q : p > q$*, and* $subhead_{p,q}^{i+1}, \forall p, q : p \leq q$*.*

Note that, if we consider α^0 the first slice, every operations in α is in one and only one slice. There are subheaders of iteration 0 that are not assigned to any slice, but since by definition they are empty, they do not need further consideration. The slice is the basic unit that we will use to define the sequential order that our algorithm enforces. We present now the sequential order for each slice separately. The order for the whole computations is obtained by simply concatenating the slices in their numerical order. Hence, we define now, for each slice α^i, the permutation β^i which contains all the operations of the slice in the sequential order.

Definition 13 *The sequence* β^i *is obtained by concatenating the sequences that form the slice* α^i *as follows.*

$tail_0^i \rightarrow subhead_{0,0}^{i+1} \rightarrow subhead_{1,0}^i \rightarrow subhead_{2,0}^i \rightarrow$
$... \rightarrow subhead_{n-1,0}^i \rightarrow$
$tail_1^i \rightarrow subhead_{0,1}^{i+1} \rightarrow subhead_{1,1}^{i+1} \rightarrow subhead_{2,1}^i \rightarrow$
$... \rightarrow subhead_{n-1,1}^i \rightarrow$
$\ldots$
$tail_p^i \rightarrow subhead_{0,p}^{i+1} \rightarrow$
$... \rightarrow subhead_{p,p}^{i+1} \rightarrow subhead_{p+1,p}^i \rightarrow$
$... \rightarrow subhead_{n-1,p}^i \rightarrow$
$\ldots$
$tail_{n-1}^i \rightarrow subhead_{0,n-1}^{i+1} \rightarrow subhead_{1,n-1}^{i+1} \rightarrow$
$subhead_{2,n-1}^{i+1} \rightarrow ... \rightarrow subhead_{n-1,n-1}^{i+1}$

In fact, this is only one of many ways to order the sequences of the slice to obtain a sequential order.

We define now the sequence β, which we claim is a sequential view of α.

Definition 14 *The sequence* β *is the permutation of* α *obtained by the concatenation of all sequences* β^i *in order (i.e.,* $\beta^i \xrightarrow{\beta} \beta^{i+1}, \forall i \geq 0$*).*

From the above definitions, in β, we have that $tail_p^i \xrightarrow{\beta} tail_q^j$ if and only if either $i < j$ or $i = j$ and $p < q$. This is exactly the order in which the sets associated with each tail are processed and applied in the algorithm.

We show in the following lemmas that β is in fact a sequential view of α.

Lemma 4 *Each prefix of* β *preserves the program order* $\prec^\alpha$*.*

Proof: Let op and op' be two operations of some prefix γ of β such that $op \prec^\alpha op'$. Then, from Definition 4, op and op' must belong to the same process and $op \xrightarrow{\alpha} op'$. It is easy to check from the above definitions of β and β^i that operations from the same process appear in the same order in β as in α. Hence, $op \xrightarrow{\beta} op'$, and therefore $op \xrightarrow{\gamma} op'$. ■

The proof of the following lemma is omitted due to space limitation.

Lemma 5 *For every read operation* $op = r(x)v$ *in* β*, the nearest previous write operation in* β *on the variable* x *is* $op' = w(x)v$*.*

Lemma 6 *Each prefix of* β *is legal.*

Proof: Let γ be a prefix of β. Let us consider a read operation $op = r(x)v$ in γ. From Lemma 5, the previous write on x in β, and hence in γ, is $op' = w(x)v$. Therefore, from Definition 1 γ has to be legal. ■

Lemma 7 *β is a sequential view of α and, therefore, α is a sequential computation.*

Proof: From Lemma 4, each prefix of β preserves the program order $\prec^{\alpha}$. Also, from Lemma 6, each prefix of β is legal. Hence, from Definition 4, β is a sequential view of α. Then, α is a sequential computation. ■

Theorem 2 *The algorithm $\mathcal{A}(sequential)$ implements sequential consistency.*

Proof: From Lemma 7, every computation α observed in the execution of the algorithm $\mathcal{A}(sequential)$ is sequential. Hence, from Definition 6, the algorithm is sequential. ■

6. $\mathcal{A}(cache)$ Implements Cache Consistency

In this section, we show that the algorithm $\mathcal{A}$, executed with the parameter $cache$ in each process, implements cache consistency. In the rest of this section we assume that α is a computation observed in the execution of the algorithm $\mathcal{A}(cache)$, and $\alpha(x)$ is a sequence formed by all the operations in α on the variable x.

The proof of correcteness follows the same lines as the proof of correctness for $\mathcal{A}(sequential)$, but on $\alpha(x)$ instead of α. First we define the sequences $it(x)_p^i$, $tail(x)_p^i$, $head(x)_p^i$, $subhead(x)_{p,q}^i$, and the slice $\alpha(x)^i$ of $\alpha(x)$. Then we construct the sequence $\beta(x)$ from these sequences in a similar way as the sequence β was defined in Section 5. A version for $\beta(x)$ of Lemma 4 is directly derived. In a version for $\beta(x)$ of Lemma 5 with the above sequences the case 3 dissapears, while the version of Lemma 6 is basically the same. Hence we have that $\beta(x)$ is a cache view of $\alpha(x)$, and therefore $\alpha(x)$ is a cache computation. Since this is true for any variable x, we have the following theorem.

Theorem 3 *The algorithm $\mathcal{A}(cache)$ implements cache consistency.*

The details are ommitted due to space limitation.

7. Complexity Measures

Worst-Case Response Time In this section we consider that local operations are executed instantaneously (i.e., in 0 time units) and that any communication takes d time units. In the algorithm $\mathcal{A}$ executed with parameter $causal$ or $cache$ all operations are executed locally, while when executed with parameter $sequential$ all write and some read operations are also executed locally. Therefore, the response time for them is always 0.

Let us now consider a read operation that is blocked in algorithm $\mathcal{A}(sequential)$. To obtain the maximum response time for such a read operation, we will consider the worst case. This can happen if the operation blocks (almost) immediately after the process that issued it sent a message. Then, the read operation will be blocked until the turn of this process again, which can take up to n message transmissions. Therefore, in the worst case, a process will have to wait nd time units.

The previous analysis assumes that the messages are never delayed at the processes. However, the protocol allows the processes to control when to send the messages. For instance, it is possible for a process p, when $turn_p = p$, to wait a time T before executing its task $send\_updates()$ (see Fig. 1). Thus, we can reduce the number of messages sent by this process per unit of time. Obviously, this can increase the response time, since in this case the delay time of a message sent by p, in the worst case, will be $T + d$.

Message Size It is easy to check in Fig. 1 that the size of the list $updates_p$ of process p depends on the number of write operations performed by p during each round, which can be very high. However, the number of pairs (x, v) in $updates_p$ will be, at most, the same as the number of shared variables, since we only hold at most one pair for each variable.

The bound obtained may seem extremely bad. However, note that the real number of pairs in a set $updates_p$ really depends on the frequency f of write operations and the rotation time nd. Hence if we have a write operation on a variable every milisecond, in a system with 100 processes and 1 milisecond of delay, we will have at most 100 pairs in the set $updates_p$ broadcasted, which is a reasonable number.

Furthermore, note that most algorithms that implement propagation and full replication send a message for *every* write operation that is performed. This would mean that 100 messages would have to be sent. With our algorithm, only one pair per variable is sent, and all of them are grouped into one single message. With the overhead per message in current networks, this implies a significant saving in bandwidth.

Memory Space Finally, note that we do not require the communication channels among processes to deliver messages in order. Hence, a process could have received messages that are held until the message from the appropriate process arrives. It is easy to check that the maximum number of messages that will ever be held is $n - 2$.

8. Conclusions and Future Work

In this paper, we have presented a parametrized algorithm that implements sequential, causal, and cache consistency on a distributed system. To our knowledge, this is the first algorithm that implements cache consistency.

The algorithm presented in this paper guarantees fast operations in its causal and cache executions. It is proven in [10] and [4] that it is imposible to have a sequential algorithm with all operations fast. The algorithm presented in this paper guarantees in its sequential execution fast writes and reduces to only one case the reads that can not be executed locally.

Considering possible extensions of this work for the sequential version, we would like to know how many read operations are fast in real applications with several system parameters. Our belief is that most read operations will be fast. A second line of work has to do with the scalability of the protocol. The worst case response time is linear on the number of processes. Hence, it will not scale well, since it may become high when the system has a large number of processes. It would be nice to remove this dependency. Finally, the protocol works in a token passing fashion, which can be very risky in an environment with failures, since a single failure can block the whole system. It would be interesting to extend the protocol with fault tolerance features.

References

[1] S.V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993.

[2] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[3] 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.

[4] H. Attiya and J.L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[5] V. Cholvi. Specification of the behavior of memory operations in distributed systems. *Parallel Processing Letters*, 8(4):589–598, December 1998.

[6] 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.

[7] J.R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[8] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[10] R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[11] 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.

[12] 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.

[13] A.K. Singh. Bounded timestamps in process networks. *Parallel Processing Letters*, 6(2):259–264, 1996.

[14] H.S. Sinha. *Mermera: Non-Coherent Distributed Shared Memory for Parallel Computing*. PhD thesis, Computer Science Department, Boston University, April 1993.