Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software xxx (2007) xxx–xxx
The Journal of Systems and Software
www.elsevier.com/locate/jss

A parametrized algorithm that implements sequential, causal, and cache memory consistencies

Ernesto Jiménez [b], Antonio Fernández [a], Vicent Cholvi [c,*]

[a] Universidad Rey Juan Carlos, 28933 Móstoles, Spain
[b] Universidad Politécnica de Madrid, 28031 Madrid, Spain
[c] Departamento de Lenguajes y Sistemas Informáticos, Universitat Jaume I, Campus de Riu Sec, 12071 Castellón, Spain

Received 16 August 2006; received in revised form 28 February 2007; accepted 1 March 2007

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 includes a parameter that allows us to choose the consistency model to be implemented. If all processes run the algorithm with the same value in this parameter, the corresponding consistency is achieved. (Additionally, the algorithm tolerates that processes use certain combination of parameter values.) This characteristic allows a concrete consistency model to be chosen, but implements it with the more efficient algorithm in each case (depending of the requirements of the applications). Additionally, as far as we know, this is the first algorithm proposed that implements cache coherence.

In our algorithm, all the read and write operations are executed locally when implementing causal and cache consistency (i.e., they are *fast*). It is known that no sequential algorithm has only fast memory operations. In our algorithm, however, all the write operations and some read operations are fast when implementing sequential consistency. The algorithm uses propagation and full replication, where the values written by a process are propagated to the rest of the processes. It works in a cyclic turn fashion, with each process of the DSM system, broadcasting one message in turn. The values written by the process are sent in the message (instead of sending one message for each write operation): However, unnecessary values are excluded. All this permits the amount of message traffic owing to the algorithm to be controlled.

Keywords: Distributed shared memory; Causal consistency; Sequential consistency; Distributed systems; Coherency models

1. Introduction

Distributed shared memory (DSM) is a well-known mechanism for inter-process communication in a distributed environment (Steinke and Nutt, 2004; Manovit and Hangal, 2005). Roughly speaking, it consists of using read and write operations for inter-process communication, thus hiding the particular communication technique employed by the programmers to avoid the need of involvement in the management of messages.

In general, most DSM protocols support *replication* of data in order to increase concurrency. 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, the system must control the replicas when the variables are updated to guarantee the *consistency* of the shared memory. This control can be performed by either *invalidating* outdated replicas or *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.

An interesting property of any algorithm implementing a consistency model is how long a memory operation can

[*] Corresponding author.
E-mail address: vcholvi@uji.es (V. Cholvi).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.03.012

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*, that is, a most desirable feature. An algorithm is fast if all its operations are fast. For instance, one of the most widely known memory models, the causal consistency model (Ahamad et al., 1995), has been implemented in a fast way on several occasions (Ahamad et al., 1995; Prakash et al., 1997; Raynal and Ahamad, 1998). On the contrary, it has been shown how another of the widely known memory models, the sequential (Lamport, 1979), cannot be implemented in a fast manner (Attiya and Welch, 1994). This impossibility result restricts the efficiency of any implementation of sequential consistency (Raynal, 2002; Raynal and Vidyasankar, 2004).

1.1. Our work

In this paper, we introduce a parametrized algorithm that implements sequential, causal, and cache consistency (Cholvi and Bernabéu, 2004), and allows us to change the model it implements on-line. We now go on to provide the main reasons to choose these three models of consistency. It has been shown that many practical distributed applications require competing operations (Attiya and Friedman, 1992) (i.e., operations that need synchronization among them). We have chosen to implement the sequential consistency model because it is the most popular model proposed that provides competing operations (other than the atomic consistency model (Misra, 1986), which is more restrictive). However, it has also been shown that there are several classes of applications which when executed with algorithms that implement causal consistency behave as sequentially consistent (Ahamad et al., 1995; Raynal and Schiper, 1996). Hence, we have also chosen to implement the causal consistency model with an algorithm where all memory operations are fast, thus avoiding the efficiency problems of sequential consistency algorithms (Attiya and Welch, 1994). The cache model is also included, even though it is not as popular, because of the extreme simplicity of its integration into our algorithm and its interest (at least theoretical) for applications that require competing operations, but only on the same variable. Furthermore, and as far as we know, this is the first algorithm proposed to implement cache consistency.

Our algorithm is implemented by using full propagation and broadcasts to perform such a task. On one hand, it is known that propagation is more expensive than invalidation (in terms of network traffic) since, in addition to the invalidating messages, data must be sent. In turn, by using invalidation, when a request cannot be locally served (because the local replica has been invalidated), this leads to starting a process that will create a new replica, thus increasing latency. Therefore, each mechanism has its advantages and drawbacks. However, it must be taken into account that, in the case of using propagation, the transfer of data are carried out concurrently with the application program, and when the memory operations can be immediately served, the network traffic does not affect the system's performance. This occurs in the 100% of cases when the consistency model is causal or cache, and in the 99% of the cases when the consistency model is sequential (see our experimental results in Appendix).

The algorithm works as follows: a write operation is propagated from the process that issues it to the rest of processes so that they can apply it locally. However, write operations are not propagated immediately. The algorithm works in a cyclic turn fashion, with each process broadcasting one message in turn. This scheme allows a very simple control of the load of messages in the network, since only one message is sent by each process in turn. This enables several write operations to be grouped in a single propagation message, thus reducing the network load.

When implementing causal and cache consistency, all the operations in our algorithm are fast. Obviously, this is not the case for the sequential model (given the results in Attiya and Welch (1994), we are reminded that the impossibility of all memory operations being fast is derived). However, all write operations are always fast even in the case of the sequential model. Conversely, this does not occur with read operations, but there is only one situation where read operations must be non-fast: when the process that issues that read operation has not issued, since its last turn, any write operations on the variable being read, but has issued some write operation on another variable. In this case, the process must wait until it reaches its turn.

1.2. Comparison with previous work

From the set of algorithms that implement DSM, two of them have features similar to those presented in this paper. In the first one (proposed by Yehuda Afek and Merritt (1993) for sequential memory), the algorithm also ensures that write operations will be fast. Additionally, read operations are fast except for some situations, but these situations are more general than that in our algorithm, which makes our algorithm faster. Furthermore, we do not send each variable update in a single message as achieved in Yehuda Afek and Merritt (1993) and we can also bound the number of messages sent. Finally, in Yehuda Afek and Merritt (1993) it is assumed that there is a communication medium among all processes (and with the shared memory) that guarantees the total order among concurrent write operations. In our case, we have no such restriction and enforce the order of the operations by using a cyclic turn technique.

On the other hand, the authors in Raynal and Schiper (1996) propose an algorithm that implements three consistency models (sequential, causal, and a hybrid between both models). Such an algorithm can dynamically switch among these three consistency models. However, there are a number of differences with the algorithm we propose. Firstly, their algorithm is designed by separating the propagation mechanism from the consistency policy. In our algorithm on the other hand, the propagation mechanism is sufficient to maintain the consistency model. Furthermore, in their implementation, an adaptation of vector clocks is used (Singh, 1996) (called *version vectors*) which not only results in a large waste of memory in each node, but also in larger messages to be sent through the network. Finally, this implementation also forces several restrictions to achieve a total order: (a) two update transactions cannot be executed concurrently, and (b) no update transaction is allowed whenever query transactions are ongoing.

The rest of the paper is arranged as follows. In Section 2, we introduce basic definitions. In Section 3, we introduce the algorithm we propose. We prove the correctness of our algorithm in Sections 4–6. In Section 7, we provide an analysis of the complexity of our algorithm. We show consistency in Section 8 when not all the processes are executing our algorithm with the same parameter. We finally present our concluding remarks in Section 9.

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 $0, \ldots, n-1$). 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 the value v stored in the variable x by a write operation to the process p that issued it. Whenever the process that performs this operation is of no particular importance, we will simply will denote them as $w(x)v$ and $r(x)v$. 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 the process that performs the operation is not important, we simply denote as x_p the local copy of variable x in process p. Different copies of the same variable can hold different values at the same time.

We use α to denote the set of read and write operations obtained in an execution of the memory algorithm.

Now we define an order among the operations observed by the processes in an execution of the memory algorithm.

Definition 1 (Execution Order). Let op and $op' \in \alpha$. Then op precedes op' in the execution order, denoted $op \prec op'$, if:

(1) op and op' are operations from the same process and op is issued before op',
(2) $op = w(x)v$ and $op' = r(x)v$, or
(3) $\exists op'' \in \alpha{:}op \prec op' \prec op'$

From this last definition, we also derive the non-transitive execution order (denoted as $\prec_{\mathrm{nt}}$) as a restriction of the execution order if the transitive closure (i.e., the third condition) is not applied. Note that if $op \prec_{\mathrm{nt}} op'$, then op has been executed before op'. Hence, if $op \prec op'$, then op has also been executed before op'. If $op \prec op'$, we define by a *related sequence* between op and op' a sequence of operations $op^1, op^2, \ldots, op^m$ such that $op^1 = op$, $op^m = op'$, and $op^j \prec_{\mathrm{nt}} op^{j+1}$ for $1 \leqslant j < m$.

We state that α_p is the set of operations obtained by removing all read operations issued by processes other than p from α. We also say that $\alpha(x)$ is the set of operations obtained by removing all the operations on variables other than x from α.

Definition 2 (*View*). We denote by *system view* β, *process view* β_p or it *variable view* $\beta(x)$ a sequence formed with all operations of α, α_p or $\alpha(x)$, respectively, such that this sequence preserves the execution order $\prec$.

Note that owing to the existence of operations that are not affected by the execution order, a lot of sequences on α, α_p or $\alpha(x)$, and not only β, β_p or $\beta(x)$, may preserve $\prec$.

We use $op \to op'$ to denote that op precedes op' in a sequence of operations. By abusing the notation, we will also use $set_1 \to set_2$, where set_1 and set_2 are a set of operations, to denote that all the operations in set_1 precede all the operations in set_2.

Definition 3 (*Legal View*). A view β on α is legal if $\forall r(x)v \in \alpha$, $\nexists w(x)u \in \alpha$: $w(x)v \to w(x)u \to r(x)v$ in β.

Definition 4 (*Sequential*, *Causal or Cache Algorithm*).

- An algorithm implements *sequential consistency* if for each execution α a legal view of it exists.
- An algorithm implements *causal consistency* if for each execution α a legal view of α_p exists, $\forall p$.
- An algorithm implements *cache consistency* if for each execution α a legal view of $\alpha(x)$ exists, $\forall x$.

3. The algorithm

In this section, we present the parametrized algorithm A that implements causal, cache and sequential consistency. Fig. 1 presents the algorithm in detail. As 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 following values *causal*, *sequential*, or *cache*.

We can see that all the write operations in Fig. 1 are fast. When a process p issues a write operation $w_p(x)v$, the

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

$r_p(x)$ :: 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($updates_p$)
  $updates_p \leftarrow \emptyset$
  $turn_p \leftarrow (turn_p + 1) \bmod n$
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
      $x_p \leftarrow v$
  $turn_p \leftarrow (turn_p + 1) \bmod n$
end
```

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

algorithm changes the local copy of variable x (denoted 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 included in *updates$_p$*, it is removed before inserting the new pair, since it does not need to be propagated anymore.

Processes propagate their respective *updates$_p$* sets 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 its local set of updates *updates$_p$* to the rest of processes. 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 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 $(turn_p + 1) \bmod n$ (see Fig. 2).

The atomic task *apply_updates*() is 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 $r_p(x)$ is fast unless *updates$_p$* contains a pair with a variable different to x. That is, the read operation is not fast only if process p, since the most recent time it took

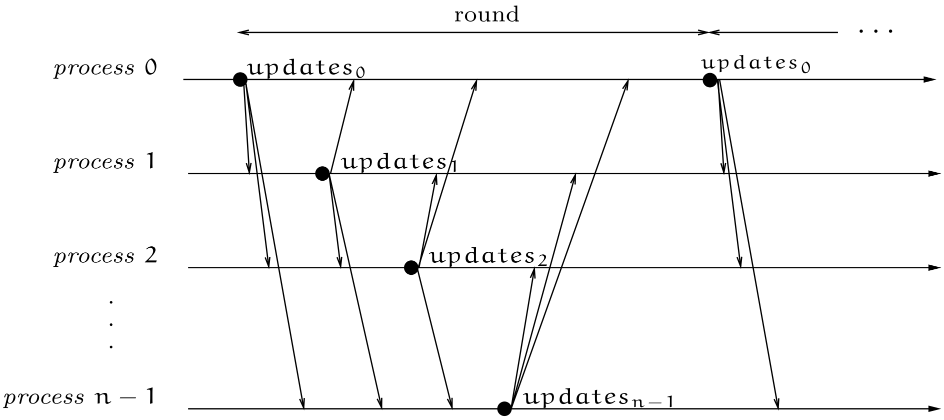

Fig. 2. Cyclic turn fashion.

its turn, has not issued write operations on x, but 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). Otherwise, if process p issues $w_p(x)v$ (and the pair (x, v) is included in p's update queue) then, according to the algorithm, all subsequent read operations issued by p on x will return v (note that a pair (x, u) broadcasted from another process is only applied by p if $(x,\cdot) \notin updates_p$). Therefore, the algorithm can return immediately v, instead of delaying the read operation. As it will be formally justified later, this ''interpretation'' of the execution still allows to find a sequence β that preserves the order $\prec$ and is legal (which are the conditions to have a sequential execution). On the contrary, if $w_p(x)v$ does not exist, then the subsequent read operation does not know, in advance, which will be the write operation on the same variable that will immediately precede it, and hence, has to wait until $turn_p = p$. 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 wish 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. *A*(*causal*) implements causal consistency

In this section, we show that Algorithm A, executed with the parameter *causal*, implements causal consistency. In the rest of this section we assume that α is the set of operations obtained in the execution of Algorithm $A(causal)$, and α_p is the set of operations obtained by removing all read operations issued by processes other than p from α.

Definition 5. The ith writes of process q, denoted $writes_q^i$, $i > 0$, is the sequence of all write operations of process q in α_p, in the order they are issued, after *send_updates*() is executed for the ith time in this process q, and before it is executed for the $i+1$st time.

To simplify, 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 $A(causal)$ that the $i+1$st $updates_q$ 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 sequence β_p which we will show in the following lemmas that preserves $\prec$ and is legal.

Definition 6. We denote β_p to the sequence formed with all operations 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 sequence point at which *apply_updates*() is executed with the $updates_q$ for the $i+1$st time, for all $q \neq p$ and $i \geqslant 0$.

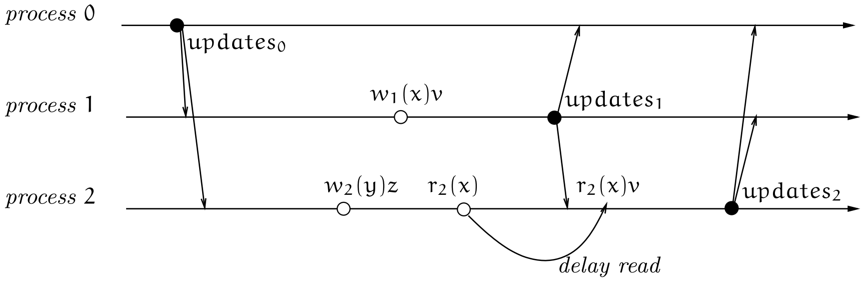

Fig. 3. An example of a "non-fast" read operation.

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.

Lemma 1. *Let op and op′ be two write operations in* α_p *issued by different processes. If* $op \prec op'$, *then* $op \rightarrow op'$ *in* β_p.

Proof. From Definition 1, we know there is a related sequence of operations $op^1, op^2, \ldots, op^m$ such that $op^1 = op$, $op^m = op'$, and $op^j \prec_{\text{nt}} op^{j+1}$ for $1 \leqslant j < m$. This sequence can be divided into r subsequences of operations, $s_1, \ldots, s_r$, such that the operations in each subsequence s_i are issued by the same process, the first operation of s_1 is op, the last operation of s_r is op', and the last operation of s_i writes the value read by the first operation of s_{i+1} for two consecutive subsequences s_i and s_{i+1}. From Algorithm $A(causal)$ it can be seen that the last operations of two consecutive subsequences s_i and s_{i+1}, $i \leqslant r-1$, belong to sequences $write_q^i$ and $write_s^j$ (from Definition 6) in such a way that either $j > i$, or $j = i$ and $s > q$. Then, $op \rightarrow op'$ in β_p. □

Lemma 2. β_p *preserves the order* $\prec$.

Proof. Let op and op' be two operations of β_p so that $op \prec op'$. Let us assume by way of contradiction that $op' \rightarrow op$.

Case 1. op and op' are issued by the same process. Let us assume that they are issued by process q. Recall that α_p only contains write operations of a process different to p. From Definition 5, if $op' \rightarrow op$ it is because op' is executed before op. However, we know from Definition 1 that if $op \prec op'$, op must be executed before op'. Hence, we reach a contradiction. Similarly, let us assume that op and op' are issued by p. From Definition 6, if $op' \rightarrow op$ is because op' is executed before op; but, from Definition 1, if $op \prec op'$, then op must be executed before op'. Hence, we also reach a contradiction.

Case 2. op and op' are issued by different processes. First, let us suppose that op and op' are operations issued by processes other than p, with differences between them. Then, as α_p only contains read operations of process p, op and op' must be write operations. Therefore, from Lemma 1, $op \rightarrow op'$, and we reach a contradiction.

Now, let us suppose that op is a read operation issued by p, and, as α_p only contains write operations of processes other than p, op' must be a write operation of a process different to p. We know that if $op \prec op'$, we have a related sequence $op = op^1 \prec_{\text{nt}} op^2 \prec_{\text{nt}} \ldots, \prec_{\text{nt}} \; op^n = op' = w(y)v$. If op^i is the first write operation after op then, from Definition 1, op^i has to be issued by p, and op has to be issued before op^i. Therefore, from Definition 6, $op \rightarrow op^i$, and, from Lemma 1, $op^i \rightarrow op'$. Hence, $op \rightarrow op'$, and a contradiction is reached.

Finally, let us suppose that op' is a read operation issued by p, and, as α_p only contains write operations of processes other than p, so op must be a write operation of a process different to p. We know that if $op \prec op'$, we have a related sequence $op = op^1 = w(x)v \prec_{\text{nt}} \ldots, \prec_{\text{nt}} \; op^{n-1} \prec_{\text{nt}} op'$. If $op^j = w_q(y)v$ is the last write operation before op', from Definition 1, op^j has to be executed before op'. With Algorithm $A(causal)$, it is implied that op^j is propagated to process p before op' is issued. This is because *apply_updates*() is executed in process p with $updates_q$ containing op^j before op' is issued. Therefore, we obtain $op^j \rightarrow op'$ from Definition 6, and $op \rightarrow op^j$ from Lemma 1. Hence, $op \rightarrow op'$, and a contradiction is reached. □

Lemma 3. β_p *is legal.*

Proof. Let us assume, by way of contradiction, that β_p is illegal because $op' = w_q(x)u \rightarrow op'' = w_s(x)v \rightarrow op = r_p(x)u$ exists in β_p. We know, from Definition 6, that if op' precedes op'' and op'' precedes op, then in process p we find: firstly, op' is issued (or applied if $q \neq p$), next, op'' is issued (or applied if $s \neq p$), and finally, op is issued. From Algorithm $A(causal)$ we can see that, owing to these write operations op' and op'', the local copy x_p of x will have the value u and will later take the value v. We can also see that in $A(causal)$ a read operation always returns the value of the local copy of a variable. It is not therefore possible to have op in β_p after op'', since it would mean that op would have found the value v in x_p, instead of the value u. Hence, a contradiction is reached and β_p is legal. □

Theorem 1. *Algorithm* $A(causal)$ *implements causal consistency.*

Proof. We know from Lemmas 2 and 3 that every execution of Algorithm $A(causal)$ has a view β_p of α_p, $\forall p$, that preserves $\prec$ and is legal. Hence, from Definition 4, Algorithm $A(causal)$ is causal. □

5. *A*(*sequential*) implements sequential consistency

In this section, we show that Algorithm A, executed with the parameter *sequential*, implements sequential consistency. In the rest of this section we assume that α is the set of operations obtained with the execution of Algorithm $A(sequential)$. Any time reference in this section is related to the time at which the operations of α are executed. We now firstly introduce some definitions of the subsets of α.

Definition 7. The ith *iteration* of process p, denoted it_p^i, $i > 0$, is the subset of α that contains all the operations issued by process p after *send_updates*() is executed for the ith 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*().

Please cite this article in press as: Jiménez, E. et al., A parametrized algorithm that implements sequential, causal, and ..., J. Syst. Software (2007), doi:10.1016/j.jss.2007.03.012

Definition 8. The ith *iteration tail* of process p, denoted $tail_p^i$, is the subset 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, then $tail_p^i$ 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 $A(sequential)$ 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 9. The ith *iteration header* of process p, denoted $head_p^i$, is the subset 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 $tail_p^i$ in the execution of A. We now use the time instants sets received from other processes which are applied to partition the sequence $head_p^i$. Note that between the ith 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 $(p+1) \bmod n, \ldots, n-1, 0, \ldots, (p-1) \bmod n$ (in this order).

Definition 10. The *iteration subheader* q of $head_p^i$, denoted $subhead_{p,q}^i$, is the subset 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 sets of operations.

With these definitions, we now divide the set of operations α into slices. This division is done in such a way that it preserves the order of the execution of α (see Fig. 4).

Definition 11. The ith *slice* of α, denoted α^i, $i \geqslant 0$, is the subset of α formed by the sets of operations $tail_p^i, \forall p$, $subhead_{p,q}^i, \forall p, q : p > q$, and $subhead_{p,q}^{i+1}, \forall p, q : p < q$.

Note that, if we consider α^0 the first slice, every operation in α is in one and only one slice. There are subheaders of iteration 0 that are not assigned to any slice, but since they are empty by definition, 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 hereby present the sequential order for each slice separately. The order for the whole execution is obtained by simply concatenating the slices in their numerical order. To complete this sequential order however, we still need to define an order into each subset of operations in tails and subheads that constitute the slice α^i. Thus, from now onward in the rest of this section, we assume that the operations in any $tail_p^i$ and $subhead_{q,p}^i$ are placed into order as they were issued by process p. Hence, we define the sequence β^i which contains all the operations of the slice in the sequential order, for each slice α^i.

Definition 12. Sequence β^i is obtained by placing the operations into each $tail_p^i$ and $subhead_p^j$ of α^i in the order as they were issued by process p, and by concatenating the set of tails and subheads of α^i as follows:

$$
\begin{aligned}
& tail_0^i \to subhead_{0,0}^{i+1} \to subhead_{1,0}^i \to subhead_{2,0}^i \\
& \quad \to \ldots \to subhead_{n-1,0}^i \to \\
& tail_1^i \to subhead_{0,1}^{i+1} \to subhead_{1,1}^{i+1} \to subhead_{2,1}^i \\
& \quad \to \ldots \to subhead_{n-1,1}^i \to \\
& \ldots
\end{aligned}
$$

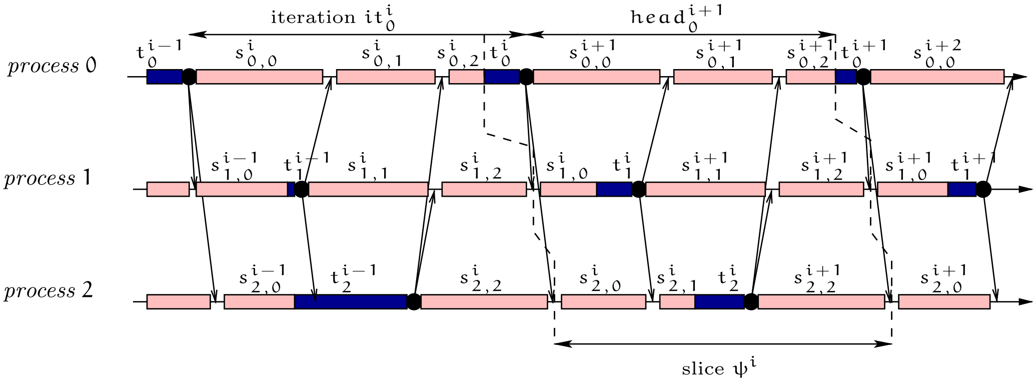

Fig. 4. Iterations and slices. We have abbreviated tail with *t* and subhead with *s*.

$$\mathit{tail}^i_p \rightarrow \mathit{subhead}^{i+1}_{0,p} \rightarrow \ldots \rightarrow \mathit{subhead}^{i+1}_{p,p}$$
$$\rightarrow \mathit{subhead}^i_{p+1,p} \rightarrow \ldots \rightarrow \mathit{subhead}^i_{n-1,p} \rightarrow$$
$$\ldots$$
$$\mathit{tail}^i_{n-1} \rightarrow \mathit{subhead}^{i+1}_{0,n-1} \rightarrow \mathit{subhead}^{i+1}_{1,n-1}$$
$$\rightarrow \mathit{subhead}^{i+1}_{2,n-1} \rightarrow \ldots \rightarrow \mathit{subhead}^{i+1}_{n-1,n-1}$$

In fact, this is only one of the many ways to order the sequences of the slice to obtain a sequential order. All the subheaders that appear in the same line above could be permuted in any possible way, since they only contain read operations and each contains operations from a different process. We chose the above order for simplicity.

We now go on to define the sequence β.

Definition 13. $\beta$is the sequence of α obtained by the concatenation of all sequences β^i in order (i.e., $\beta^i \rightarrow \beta^{i+1}, \forall i \geqslant 0$).

From the above definitions, we find that $\mathit{tail}^i_p \rightarrow \mathit{tail}^j_q$ 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.

In the following Lemmas, we show that β is in fact a view that preserves the order $\prec$ and is legal.

Lemma 4. *β preserves the order $\prec$.*

Proof. Let op and op' be two operations of β in such a way that $op \prec op'$. From Definition 1, we know that there is a related sequence of operations $op^1, op^2, \ldots, op^m$ so that $op^1 = op$, $op^m = op'$, and $op^j \prec_{\mathrm{nt}} op^{j+1}$ for $1 \leqslant j < m$. If β preserves $\prec$, then $op^j \rightarrow op^{j+1}$, $\forall j$, and, hence, $op \rightarrow op'$. We consider several cases.

Case 1. op^j and op^{j+1} are operations issued by the same process. If from Definition 1, $op^j \prec op^{j+1}$, then op^j must be issued before op^{j+1}. Therefore, 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 they were issued. Then, $op^j \rightarrow op^{j+1}$, $\forall j$. Hence, $op \rightarrow op'$.

Case 2. op^j and op^{j+1} are a write operation and a read operation, respectively, issued by different processes. Let us assume, from Definition 1, that $op^j = w_q(x)u$ and $op^{j+1} = r_s(x)u$. We know that if $op^j \prec op^{j+1}$, then op^j must be executed before op^{j+1}. From Algorithm *A*(*sequential*) and from the above definitions of β and β^i, we can see that op^j always belongs to tail^i_q. We have two possibilities: (a) op^{j+1} belongs to $\mathit{subhead}^j_{s,l}$, $i < j$, or if $i = j$ $q \leqslant l$; or (b) op^{j+1} belongs to tail^j_p, $i < j$. In both cases, $op^j \rightarrow op^{j+1}$, $\forall j$. Hence, $op \rightarrow op'$. □

Lemma 5. *β is legal.*

Proof. Let us suppose that $op' = w(x)v \rightarrow op = r(x)v$ exists in β. From Definition 3, β is legal if $\nexists op'' = w(x)u \in \alpha$ so that $op' \rightarrow op'' \rightarrow op$ in β. Therefore, this is equivalent of saying that β is legal if $op = r(x)v$ in β for every read operation, the nearest previous write operation in β on the variable x is $op' = w(x)v$.

Let us assume that op is issued by process p. Firstly, note that the order in which the iteration tails appear in β is exactly the order imposed by the token passing procedure. Then, in p, the order in β reflects the exact order in which the sets $\mathit{updates}_q$ are applied in the local memory of p. The only exceptions are the sets $\mathit{updates}_p$, since the write operations of p itself, are applied in its local memory immediately, and do not wait until p holds the token. However, also note that any update from other processes on a variable written locally is not applied (see *apply_updates*()). This gives the idea that the local write operations have in fact been applied at the time of p's token possession. Then, we can consider several cases.

Case 1. Both op and op' belong to the same iteration tail tail^i_p. When issued by p, op' sets the value of the local copy x_p of x. After op' is executed, $(x,\cdot) \in \mathit{updates}_p$, and no update applied from other process changes this value (see *apply_updates*()). Hence, if op returns the value v it is because op' wrote the value v in x.

Case 2. op belongs to an iteration subheader $\mathit{subheader}^i_{p,q}$. The value v returned by op is the value of x_p after applying the write operations locally in the following tails:

- If $p > q$, tail^j_r for each $j < i$, and for each $r \leqslant q$ when $j = i$.
- If $p \leqslant q$, tail^j_r for each $j < i - 1$, and for each $r \leqslant q$ when 0 $j = i - 1$.

These are the tails that precede $\mathit{subheader}^i_{p,q}$ in β As already mentioned, these tails are applied in the order they appear in β. Then, v has to be the value written by the nearest write operation on x that precedes op in β, which by definition is op'.

Case 3. op belongs to an iteration tail tail^i_p, while op belongs to a different iteration tail. Then the read operation op was issued when p had already issued a write operation (since it belongs to a tail) on a variable different to x (by definition of op'). Then, op was blocked until the token was assigned to p. The value v returned by op is the value of x_p after applying the write operations locally in the tails tail^j_q for each $j < i$ and for each $q < p$ when $j = i$, which are the tails that precede tail^i_p in β. As stated previously, these tails are applied in the order they appear in β. Then, v has to be the value written by the nearest write operation on x that precedes op in β, which by definition is op'.

Thus, in the above three cases we have shown that $op' = w(x)v$ is the nearest write operation on variable x previous to $op = r(x)v$ in β. Hence, β is legal. □

Theorem 2. *Algorithm A*(*sequential*) *implements sequential consistency.*

Proof. From Lemmas 4 and 5, every execution of Algorithm *A*(*sequential*) has a view β of α that preserves the order $\prec$ and is legal. Hence, from Definition 4, Algorithm A(*sequential*) is sequential. □

6. A(*cache*) implements cache consistency

In this section, we show that Algorithm A, executed with the parameter *cache* in each process, implements cache consistency. In the rest of this section, we assume that α is a set of operations produced in the execution of Algorithm A(*cache*), and $\alpha(x)$ is a set of operations formed by all the operations in α on the variable x.

The proof of correctness follows the same lines as the proof of correctness for 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 similarly to how sequence β was defined in Section 5. A version for $\beta(x)$ of Lemma 4 is directly derived. Case 3 disappears in a version for $\beta(x)$ of Lemma 5 with the above sequences. Hence, we see that $\beta(x)$ is a view of $\alpha(x)$ that preserves the order α and is legal. Since this is true for any variable x, we obtain the following theorem.

Theorem 3. *Algorithm* A(*cache*) *implements cache consistency.*

Proof. From Lemmas 4 and 5 (but only with operations of $\alpha(x)$), every execution of Algorithm A(*cache*) has a view $\beta(x)$ of $\alpha(x)$, $\forall x$, that preserves the order $\prec$ and is legal. Hence, from Definition 4, Algorithm A(*cache*) is cache. □

7. Complexity measures

7.1. 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 Algorithm A executed with either parameter *causal* or *cache* all operations are executed locally, while all write and some read operations are also executed locally with the parameter *sequential*. Therefore, the response time for them is always 0.

Let us now consider a read operation that is blocked in Algorithm A(*sequential*). We will consider the worst case to obtain the maximum response time for such a read operation. 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 this process takes its turn again, which can take up to n message transmissions. Therefore, a process will have to wait nd time units for the worst case.

The previous analysis assumes that the messages are never delayed in the processes. However, the protocol allows the processes to control when messages are sent. For instance, it is possible for a process p, when $turn_p = p$, to wait a time T before executing its task to send *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 $T + d$.

7.2. 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 the same as the number of shared variables, since we only hold at most one pair for each variable, at the most.

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

Furthermore, note that most algorithms that implement propagation and full replication send a message for *every* write operation 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.

7.3. 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. Consistency in A with different parameters

In this section, we show that Algorithm A, executed in some processes with the parameter *sequential* and with parameter *causal* in the rest of processes, implements causal consistency. In this section, we also prove that if there are processes executing A(*sequential*) and others A(*cache*), then Algorithm A implements cache consistency.

In this part of this section, we assume that α is the set of operations obtained in the execution of Algorithm A(*sequential*) and Algorithm A(*causal*), and α_p is the set of operations obtained by removing all read operations issued by processes other than p from α.

Theorem 4. *Algorithm A implements causal consistency when some processes execute* A(*sequential*) *and the rest* A(*causal*).

Proof. As seen in Fig. 1, $A(causal)$ and $A(sequential)$ have the same mechanism to propagate the write operations. Besides, read operations are always performed locally in both algorithms (without generating messages) in the process where they were issued.

It should be remembered that a sequence β_p is formed with every write operation issued by any process and with every read operation issued only by process p. Hence, from the point of view of each process executing $A(causal)$, the existence of processes executing $A(sequential)$ neither includes nor modifies the set of operations α_p to order in β_p different to other process executing $A(causal)$. Therefore, if we now construct β_p for each process p executing $A(causal)$, as we did in Definition 6, β_p will remain legal and preserve the order $\prec$.

In the following two definitions, we redefine β_p for each process p that executes $A(sequential)$. We will then show that this new sequence β_p preserves $\prec$ and is legal.

For construct β_p we use the notation of slice from Section 5. However, from this point onward, we use α_p instead of α to see how the set of operations of α_p is divided. We also use $writes^i_p$ and $subhead^i_{p,q}$ as we defined in Sections 4 and 5, respectively, to see how the set of operations of α^i_p is ordered.

Definition 14. The ith *slice* of α_p, denoted α^i_p, $i \geqslant 0$, is the subset of α_p formed by the sets of operations $writes^i_q, \forall q$, $subhead^i_{q,p}, \forall q : q > p$, and $subhead^{i+1}_{q,p}, \forall q : q < p$.

β^i_p denotes the sequence of all operations of the slice α^i_p, and β_p denotes the sequence of α_p formed by the concatenation of β^i_p in an increasing numerical order.

Definition 15. The sequence β^i_p, $\forall p$ executing $A(sequential)$, is obtained by ordering the operations into each $writes^i_p$ and $subhead^j_p$ of α^i_p in the order as they were issued by process p, and by concatenating the set of writes and subheads of α^i_p as follows:

$writes^i_0 \rightarrow subhead^i_{p,0} \rightarrow$

$writes^i_1 \rightarrow subhead^i_{p,1} \rightarrow$

$\ldots$

$tail^i_p \rightarrow subhead^{i+1}_{p,p} \rightarrow$

$\ldots$

$writes^i_{n-1} \rightarrow subhead^{i+1}_{p,n-1}$

Definition 16. The sequence β_p, $\forall p$ executing $A(sequential)$, is the sequence of α obtained by the concatenation of all sequences β^i_p in order (i.e., $\beta^i_p \rightarrow \beta^{i+1}_p, \forall i \geqslant 0$).

By comparing Definitions 12 and 15, we see that β^i_p is the same sequence as β^i (and, therefore, β_p and β) but with the following two differences:

- $writes^i_q$, $q \neq p$, is the same sequence as it^i_q where we have removed every read operation issued by q.
- We have removed every subhead with operations other than process p. That is to say, every $subhead^j_{q,n}$ such as $q \neq p$, $j = i$ or $j = i + 1$, and $\forall n$.

As shown, we have constructed β_p in a similar way to the sequence β that was defined in Section 5. Then, all the operations belonging to β_p are in the same order as in the definition of β, and a version for β_p of Lemma 4 is directly derived. Hence, β_p preserves the order $\prec$.

Similarly, we know that all the write operations which are in β_p are in the same order as in the definition of β. Then, a version for β_p of Lemma 5 is also directly derived. Hence, β_p is legal.

Thus, we have for each process p executing $A(causal)$ a β_p is formed as described in Section 4. A β_q is also formed for each process q executing $A(sequential)$, as in Definition 16. As in both cases, β_p is legal and preserves $\prec$, and we can affirm that A implements causal consistency when there are processes executing $A(sequential)$ and others $A(causal)$. □

In this part of this section, we assume that α is the set of operations obtained in the execution of Algorithm $A(sequential)$ and Algorithm $A(cache)$, and $\alpha(x)$ is the set of operations of α on variable x.

Theorem 5. *Algorithm A implements cache consistency when some processes execute* $A(sequential)$ *and the rest* $A(cache)$.

Proof. We can see in Fig. 1 that $A(cache)$ and $A(sequential)$ are the same algorithm except in one case: when a read operation is not fast. Therefore, they use the same mechanism to propagate the write operations, and read operations do not add new messages because they are managed locally in the process where it is invoked.

Hence, from the point of view of each process that executes $A(cache)$, the existence of processes executing $A(sequential)$ neither includes nor modifies the set of operations which are ordered in $\beta(x)$ and differ from the other processes that execute $A(cache)$. Therefore, if we now construct $\beta(x)$ for each process p that executes $A(cache)$ as we did in Section 6, $\beta(x)$ will remain legal and will preserve $\prec$.

Now, we will use the same definition and way of construction of $\beta(x)$ from Section 6 for all the processes that execute $A(sequential)$. Then, the difference between the sequence $\beta(x)$ for processes executing $A(sequential)$ with regard to those executing $A(cache)$ is what happens when a non-fast read operation occurs. To analyse this case, we use the same notation from Section 6 to define a slice $\alpha^i(x)$, a tail $tail^i_p(x)$, and a subhead $subhead^j_{q,p}(x)$. Then, let us assume that a non-fast read operation occurs in a process p that executes $A(sequential)$ in the slice $\alpha^i(x)$. We know, by the definition of $\beta(x)$, that this read operation belongs to $tail^i_p(x)$. We will also assume that the first operation of $tail^i_p(x)$ takes place just after $subhead^j_{q,p}(x)$, $i = j$ if $q > p$, or $j = i + 1$ if $q \leqslant p$. Hence, in this case, the unique difference

Please cite this article in press as: Jiménez, E. et al., A parametrized algorithm that implements sequential, causal, and ..., J. Syst. Software (2007), doi:10.1016/j.jss.2007.03.012

with regard to an execution in a process $A(cache)$ is that each subhead subsequent to $subhead^{j}_{q,p}(x)$ in $\alpha^{i}(x)$ is always empty. Therefore, as with the sequence $\beta(x)$ of a process executing $A(cache)$ from Section 6, a version of Lemmas 4 and 5 for $\beta(x)$ of a process executing $A(sequential)$ is directly derived. Hence, $\beta(x)$ of a process executing $A(sequential)$ is legal and preserves the order $\prec$.

Thus, we have shown for each process p that executes $A(cache)$ or $A(sequential)$ that there is a $\beta(x)$, $\forall x$, formed as described in Section 6, such that the order $\prec$ is preserved and is legal. Therefore, A implements cache consistency when processes executing $A(sequential)$ and others $A(cache)$, exist. □

9. Conclusions and future work

In this paper, we have presented a parametrized algorithm that implements sequential, causal, and cache consistency in 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 Attiya and Welch (1994) that it is impossible to obtain a sequential algorithm whose total number of operations are fast. The algorithm presented in this paper guarantees fast writes in its sequential execution and reduces the reads to only one case that cannot be executed locally.

By considering possible extensions of this work for the sequential version, we wish 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 deals with the scalability of the protocol. The worst-case response time is linear in 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 ideal to eliminate this dependency. Finally, the protocol works in a token passing fashion, which can prove 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.

Acknowledgements

This work was partially supported by the Spanish Ministry of Science and Technology under Grants TIN2005-09198-C02-0, TIN2004-07474-C02-01 and TSI2004-02940, by Bancaixa under Grant No. P1-1B2003-37 and the Comunidad de Madrid under grant S-0505/TIC/0285.

Appendix A. Experimental results

In Section 7, we have seen that the worst-case response time can be linear in the number of processes. We know that all memory operations in A are fast except for some read operations of $A(sequential)$. This worst case can only occur in blocked read operations under sequential consistency. However, we cannot analytically evaluate how many read operations block in $A(sequential)$ irrespectively of the application that uses it. In this section, we evaluate the number of read operations that actually block during the executions of three typical parallel processing applications with our algorithm $A(sequential)$.

processes	FD	MM	FFT
2	0.47%	0.07%	0.65%
4	0.06%	0.01%	0.05%
8	0.14%	0.01%	0.03%

Fig. A.1. Percentage of blocking read operations per process in $A(sequential)$.

Then, we have implemented our algorithm $A(sequential)$ and the following three typical parallel processing applications: finite differences for 16,384 × 1024 elements (FD), Matrix Multiplication of 1600 × 1600 matrices (MM), and Fast Fourier Transform for 26,2144 coefficients (FFT). FD and MM have been implemented as in Wilkinson and Allen (1999), and FFT as in Akl (1989). We then executed the resulting system.

The three applications FD, MM and FFT have been executed in an experimental environment formed by a cluster of 2, 4, and 8 computers connected via a network. Each computer is a PC running Linux Red-Hat with a 1,5-GHz AMD CPU and 512 Mbytes of RAM memory. The network that interconnects the computers of the cluster is a switched, full-duplex 1-Gbps Ethernet. We have implemented one process per computer and the messages have been restricted to carry at most 100 write operations. The language used was C with *sockets* with the UDP/IP protocols for computer intercommunication.

Fig. A.1 shows the percentage of read operations that block in each process in relation to the total number of read operations issued by the process when FD, MM and FFT have been executed using $A(sequential)$ in our experimental environment. As we can observe, almost all read operations are fast for each case.

References

Ahamad, M., Neiger, G., Burns, J., Kohli, P., Hutto, P., 1995. Causal memory: definitions, implementation and programming. Distributed Computing 9 (1), 37–49.

Akl, S.G., 1989. The Design and Analysis of Parallel Algorithms. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Attiya, H., Friedman, R., 1992. A correctness condition for high-performance multiprocessors. In: Proceedings of the 24th Annual ACM Symposium on the Theory of Computing, pp. 679–690.

Attiya, H., Welch, J., 1994. Sequential consistency versus linearizability. ACM Transactions on Computer Systems 12 (2), 91–122.

Cholvi, V., Bernabéu, J., 2004. Relationships between memory models. Information Processing Letters 90 (2), 53–58.

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

Manovit, C., Hangal, S., 2005. Efficient algorithms for verifying memory consistency. SPAA'05: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures. ACM Press, New York, NY, USA, pp. 245–252.

Misra, J., 1986. Axioms for memory access in asynchronous hardware systems. ACM Transactions on Programming Languages and Systems 8 (1), 142–153.

Prakash, R., Raynal, M., Singhal, M., 1997. An adaptive causal ordering algorithm suited to mobile computing environments. Journal of Parallel and Distributed Computing 41, 190–204.

Raynal, M., 2002. Token-based sequential consistency. Computer Systems Science and Engineering 17 (6), 359–365.

Raynal, M., Ahamad, M., 1998. Exploiting write semantics in implementing partially replicated causal objects. In: Proceedings of the Sixth EUROMICRO Conference on Parallel and Distributed Computing, pp. 157–163.

Raynal, M., Schiper, A., 1996. From causal consistency to sequential consistency in shared memory systems, Tech. Rep. 926, IRISA (May).

Raynal, M., Vidyasankar, K., 2004. A distributed implementation of sequential consistency with multi-object operations. In: ICDCS, pp. 544–551.

Singh, A., 1996. Bounded timestamps in process networks. Parallel Processing Letters 6 (2), 259–264.

Steinke, R.C., Nutt, G.J., 2004. A unified theory of shared memory consistency. Journal of ACM 51 (5), 800–849.

Wilkinson, B., Allen, M., 1999. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Yehuda Afek, G.B., Merritt, M., 1993. Lazy caching. ACM Transactions on Programming Languages and Systems 15 (1), 182–205.