ELSEVIER

Information Processing Letters 57 (1996) 243-248

Information Processing Letters

Limitations of fast consistency conditions for distributed shared memories [1]

Hagit Attiya [a], Roy Friedman [b,*]

[a] *Department of Computer Science, The Technion, Haifa 32000, Israel*
[b] *Department of Computer Science, Cornell University, Ithaca, NY 14853, USA*

Received 10 August 1995; revised 22 December 1995
Communicated by F.B. Schneider

Abstract

A consistency condition for distributed shared memory is *fast* if it has a *fast* implementation in which the execution time of every operation is significantly faster than the network delay. These conditions include Pipelined RAM, weak consistency, causal memory, and one interpretation of processor consistency. It is shown that if a condition is fast then it does not support non-centralized solutions for mutual exclusion.

Keywords: Distributed computing; Distributed shared memory; Consistency conditions

1. Introduction

A *consistency condition* describes the allowable behaviors of a virtual shared memory which is distributed over several nodes. These behaviors admit different implementations and may impose different programming styles. Clearly, a more relaxed consistency condition with fewer restrictions on the possible behaviors admits more efficient implementations, since it allows more possible return values for each memory access. On the other hand, it is simpler to write programs assuming a strict consistency condition that restricts the possible behaviors, since the return value of each memory access is more predictable. This presents a "tradeoff" when deciding which type of consistency condition to implement.

We consider distributed systems in which a distributed shared memory abstraction, obeying a certain consistency condition, is implemented on top of a communication network.

An implementation, satisfying a certain consistency condition, is *fast* if the execution time of every operation is significantly faster than the network delay, i.e., it is $o(d)$, where d is the network delay. [2] A consistency condition is *fast* if it has a fast implementation when there are no limitations on the size of local memory. These conditions include the Pipelined RAM

* Corresponding author. Most of this research was performed while the author was at the Department of Computer Science, The Technion.

[1] This research was supported by grant No. 92-0233 from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel, the fund for the promotion of research in the Technion, and by Technion V.P.R. funds.

[2] By network delay we mean the time required to deliver a message from one node to the other, which may include going through several hops in some networks.

PII S0020-0190(96)00007-5

model of [15], weak consistency [5], causal memory [4], and one version of processor consistency [13].[3]

The ability to implement a consistency condition in such a way that the execution time of every operation is significantly faster than the network delay is appealing in terms of performance and scale. However, in this paper we show that such implementations impose a significant limitation on the programming style for solving synchronization problems. Specifically, we show that all solutions for the mutual exclusion problem based on fast conditions must be *cooperative* – certain processes have to participate in the algorithm, whether they wish to enter the critical section or not. The proof depends on the assumption that the message delay is unbounded; a similar proof holds when there is no lower bound on the step time of processes (cf. [10]).

This extends an earlier result on weak consistency [6,10]. Note that centralized solutions (like the one presented in [15]) are cooperative, since the process moderating the entry to the critical section must participate in the algorithm, even if it never wishes to enter the critical section. In contrast, all classical solutions to the mutual exclusion problem based on sequential consistency are not cooperative (cf. [8,16,17]).

This impossibility result suggests that some degree of cooperation must exist, either in the implementation of the consistency condition or in the application program, in order to solve synchronization problems. Thus, fast implementations come at a cost: Programmers may need to work harder in developing their code since the environment does not give them an adequate protection from concerns relating to synchronization.

2. The system

We use the model of [7,5]. We assume a system consisting of a collection of nodes P connected via a communication network. The virtual shared memory consists of a collection of *objects*. The shared memory abstraction is implemented by a *memory consistency system* (MCS), which uses local memory and some protocol executed by the MCS processes (one at each node). The MCS must provide the proper semantics when the values of the responses to calls are considered, throughout the network.

The following *events* may occur at the MCS process on node i:

- *Call events*: The initiation of operations. For example, in the case of read/write objects, the call events are $\text{Read}_i(X)$ and $\text{Write}_i(X, v)$ for all objects X and values v.
- *Response events*: The response of the MCS to operations. For example, in the case of read/write objects, the response events are $\text{Return}_i(X, v)$ and $\text{Ack}_i(X)$ for all objects X and values v.
- *Message send events*: $\text{Send}(i, m, j)$ for all messages m and MCS processes p_i and p_j: the MCS process on node i sends message m to the MCS process on node j.
- *Message receive events*: $\text{Receive}(i, m, j)$ for all messages m and MCS processes p_i and p_j: the MCS process on node i receives message m from the MCS process on node j.

The call and message-receive events are *interrupt events*.

An *MCS process* (or simply *process*) is an automaton with a (possibly infinite) set of states, including an initial state, and a transition function. Each interrupt event causes an application of the transition function. The transition function is a function from states and interrupt events to states, sets of response events and sets of message-send events. That is, the transition function takes as input the current state and an interrupt event, and produces a new state, a set of response events for the application process, and a set of messages to be sent. A *step* of a process p_i is a tuple (s, i, s', R, M), where s and s' are states, i is an interrupt event, R is a set of response events, M is a set of message-send events, and s', R and M are the result of p's transition function acting on s and i. A *history* of a process p is a mapping h from $\mathbb{R}$ (real time) to finite (possibly empty) sequences of steps such that

(1) for each real time t, there are only a finite number of times $t' < t$ such that the corresponding sequence of steps $h(t')$ is nonempty (thus the concatenation of all the sequences in real-time order is a sequence);
(2) the old state in the first step is p's initial state; and

[3] Note that, as shown in [9], synchronization operations like read-modify-write, test&set, and compare&swap cannot be implemented faster than the network delay, even with a very weak ordering semantics. Hence, these consistency conditions have fast implementations only when these kinds of operations are not supported.

(3) the old state of each subsequent step is the new state of the previous step.

An *execution* of an MCS is a set of histories, one for each process in P, in which there is a one-to-one correspondence from the messages received by p_i from p_j onto the messages sent by p_j to p_i, for any processes p_i and p_j. An *infinite execution* is an execution in which every history is infinite. We use the message correspondence to define the *delay* of any message in an execution to be the real time of receipt minus the real time of sending. (The network is not explicitly modeled, although the constraints on executions, defined below, imply that the network reliably delivers all messages sent.)

An execution σ is *d-admissible* if the following hold:

(1) for every i and j, every message in σ from p_i to p_j has a fixed, but unknown and unbounded, delay d;
(2) for every i, at most one call at p_i is pending at a time in σ.

From now on, we assume all executions are *admissible*, i.e., they are d-admissible for some $d > 0$.

A history h' is a *partial history* of a history h if there exists a time t_0 such that for all times $t < t_0$, $h'(t) = h(t)$, $h'(t_0)$ is a prefix of $h(t_0)$ and for all times $t > t_0$, $h'(t)$ is empty. A *partial execution* is a collection $\mathcal{E}$ of partial histories, one for every process, in which there is a one-to-one correspondence between the messages received by p_j from p_i in $\mathcal{E}$ and the messages sent by p_i to p_j in $\mathcal{E}$, for every two processes p_i and p_j. (Note that it is possible that some messages are not received in the partial execution.) Given a partial history h of a process p_i, let t_0 be the largest time for which $h(t_0)$ is not empty; the *last state* of p_i in h is the local state of p_i at the end of the sequence of steps $h(t_0)$. A *global state* is the collection of last states of all processes in a partial execution. From now on, we use the word *state* for global states, unless specifically said otherwise.

Each pair of a *call* event and a subsequent matching *response* event forms an *operation*. The call event marks the start of the operation, while the response event marks the end of that operation. An operation *op* is *invoked* when the application process issues the appropriate call for *op*; *op terminates* when the MCS process issues the appropriate response for *op*. An operation is *pending* if it is invoked and is not terminated. Note that since we assumed that all executions are admissible, there can be at most one pending operation per process, i.e., an application program does not invoke a new operation before the previous one has terminated.

The *sequential specification* (cf. [14]) of an object describes the operations that can be applied and the set of allowable operation sequences. A sequence τ of operations for a collection of processes and objects is *legal* if, for every object X, the restriction of τ to operations of X is in the sequential specification of X.

Given an execution σ, let $ops(\sigma)$ be the sequence of call and response events appearing in σ in real-time order, breaking ties by ordering all events of the same process in the order they appear in that process and then using process ids.

We sometimes identify a consistency condition with the set of executions it allows, $\mathcal{C}$. A collection of MCS processes $\mathcal{E}$ is an *implementation* of a consistency condition $\mathcal{C}$ if every execution generated by $\mathcal{E}$ is in $\mathcal{C}$.

3. The mutual exclusion problem

An algorithm for mutual exclusion consists of four disjoint sections – entry, critical, exit and remainder (cf. [17]). In the *entry* section, a process tries to gain access to the critical section; the *exit* section is executed by each process upon leaving the critical section; the *remainder* section is the rest of the code. A mutual exclusion algorithm should guarantee:

- *mutual exclusion*: no two processes are inside the critical section at the same time, and
- *deadlock freedom*: in every infinite execution, if there is a process in the entry section, then eventually there is some process in the critical section.

To prohibit centralized solutions to the mutual exclusion problem, we require that processes that are in the remainder section do not write to any object that is used in the entry or exit sections. Only processes that are trying to enter or leaving the critical section participate in the algorithm.

The *exclusion set* of a mutual exclusion algorithm $\mathcal{A}$ is the set of shared objects read from inside the entry or exit sections of $\mathcal{A}$; this set is denoted $exc(\mathcal{A})$.

Definition 1. A mutual exclusion algorithm $\mathcal{A}$ is *non-cooperative* if every process which executes the re-

mainder section of $\mathcal{A}$ does not write any object in $exc(\mathcal{A})$; otherwise, the algorithm is *cooperative*.

In particular, all solutions in which a process wishing to enter the critical section must communicate with a fixed subset of the processes, even if these processes are not trying to enter (or are inside) the critical section, are cooperative. Note that Definition 1 does not put any restrictions on reads during the remainder section, which only strengthen our impossibility result.

4. The impossibility result

The *execution time* of an operation is the time that elapses from the point the operation is invoked by the program until the point a response for it is generated by the MCS. An implementation $\mathcal{E}$ is *fast* if the execution time of every operation in $\mathcal{E}$ is $o(d)$. A consistency condition is *fast* if it has a fast implementation when there are no restrictions on local memory. Our main result is:

Theorem 2. *If $\mathcal{A}$ is an algorithm for the mutual exclusion problem which assumes a fast consistency condition C, then $\mathcal{A}$ is cooperative.*

Proof. Consider a system with two processes p_1 and p_2, and assume, by way of contradiction, that there exists a non-cooperative mutual exclusion algorithm $\mathcal{A}$ which is based on a fast consistency condition C. Let $\mathcal{E}$ be a fast implementation of C. Since the worst-case execution time of an operation in $\mathcal{E}$ is $o(d)$ it follows that for every constant $k > 0$, there exists d_0 such that for every $d > d_0$, the execution time of every operation in $\mathcal{E}$ is less than d/k (in executions where the message delay is at most d).

For any $d > 0$, consider the following execution σ_1^d of $\mathcal{A}$ on $\mathcal{E}$. In σ_1^d, p_1 and p_2 invoke their first operation at time 0, and then invoke a new operation each time an operation they invoked terminates. The delay of each message in σ_1^d is d. At time 0, p_1 makes the transition from the remainder section to the entry section. On the other hand, p_2 remains in the remainder section. Since $\mathcal{A}$ guarantees that there is no deadlock, after some finite number k_1 of its own operations, p_1 makes the transition from the entry section to the critical section. Assume that this happens at real time t_1. Since p_2 is in the remainder section, and since $\mathcal{A}$ is non-cooperative, p_2 does not write to any object in $exc(\mathcal{A})$ during σ_1^d. Note that regardless of the value of d, in σ_1^d p_2 does not write any object in $exc(\mathcal{A})$ while p_1 reads only objects from $exc(\mathcal{A})$. Thus, for every value d, p_1 enters the critical section after exactly k_1 operations.

Now for any $d > 0$, consider the symmetric execution σ_2^d of $\mathcal{A}$ on $\mathcal{E}$. In σ_2^d, p_1 and p_2 invoke their first operation at time 0, and then invoke a new operation each time an operation they invoked terminates. The delay of each message in σ_2^d is d. At time 0, p_2 makes the transition from the remainder section to the entry section. On the other hand, p_1 remains in the remainder section. Since $\mathcal{A}$ guarantees that there is no deadlock, after some finite number k_2 of its own operations, p_2 makes the transition from the entry section to the critical section. Assume that this happens at real time t_2. Since p_1 is in the remainder section, and since $\mathcal{A}$ is non-cooperative, p_1 does not write to any object in $exc(\mathcal{A})$ during σ_2^d. Note that regardless of the value of d, in σ_2^d p_1 does not write any object in $exc(\mathcal{A})$ while p_2 reads only objects from $exc(\mathcal{A})$. Thus, for every value d, p_2 enters the critical section after exactly k_2 operations.

Let $k = \max(k_1, k_2)$. It follows that there exists some d_0 such that such that for every $d > d_0$, the execution time of every operation in $\mathcal{E}$ is at most d/k (in executions where the message delay is at most d). That is, the total execution time of the operations at p_1 or p_2 is at most d time.

Since for any $d > d_0$ no message is ever received in σ_1^d and σ_2^d, the partial execution σ obtained from σ_1^d by replacing p_2's history with p_2's history in σ_2^d can be extended to an execution. However, at the last state of σ both p_1 and p_2 are in the critical section. A contradiction. □

Note that our proof does not rely on any sort of fairness. Also, the proof holds for both deterministic and randomized algorithms; this only makes the result stronger.

5. Examples of fast conditions

Pipelined RAM. Lipton and Sandberg defined Pipelined RAM and present the following implemen-

tation in [15]: Every process maintains a copy of the entire memory. Reads are serviced immediately by returning the value from the local copy of the memory. Writes also cause an immediate update to the local copy, followed by sending an `update` message to all other processes in the system. Whenever a process receives an `update` message, it immediately updates its local copy of the memory. Clearly, the Pipelined RAM has a fast implementation, and therefore Theorem 2 implies that it does not allow non-cooperative mutual exclusion algorithms.

Weak consistency. Weak consistency requires that for every process, all operations can be ordered in a way that is consistent with this process' view. However, it is possible that different processes see different orderings. Weak consistency implies that writes performed by a certain process will eventually be seen by all processes. It does not guarantee that the order in which updates are seen is the same at all processes or respects the order at which they were issued. Formally:

Definition 3 (*Weak consistency*). An execution σ is *weakly consistent* if for every process p_j there exists a legal serialization τ_j of σ such that $\tau_j \mid j = \sigma \mid j$.

Note that τ_j represents the ordering of operations that is consistent with the view of p_j. Since the Pipelined RAM of [15] provides weak consistency (for reads and writes), weak consistency is a fast condition when only reads and writes are used. Therefore, Theorem 2 implies that it does not allow non-cooperative mutual exclusion algorithms. It was shown in [9] that there are no fast implementations of weak consistency with read-modify-write operations. Therefore, weak consistency is not a fast condition if read-modify-write operations are allowed. Indeed, there are non-cooperative mutual exclusion algorithms based on weak consistency with read-modify-write operations.

Causal memory. *Causal memory* was suggested in [3], and is strictly stronger than the Pipelined RAM model. An operation op_i^1 *precedes* op_j^2 (denoted $op_i^1 \rightarrow op_j^2$) if either $i = j$ and op_i^1 was invoked before op_j^2, or $op_i^1 = w_i(x,v)$ for some memory location x and value v and $op_j^2 = r_i(x,v)$.[4] Let $\stackrel{*}{\rightarrow}$ denote the transitive closure of $\rightarrow$; op_i^1 and op_j^2 are *concurrent* if $op_i^1 \not\stackrel{*}{\rightarrow} op_j^2$. Given a pair of operations op_i^1 and op_j^2, such that op_i^1 precedes op_j^2, we say that a third operation op_k^3 is *intervening* if op_i^1 precedes op_k^3 and op_k^3 precedes op_j^2.

Definition 4. Given a read operation $r(x)$ and a write operation $w(x,v)$, the value v is *live* for $r(x)$ if either:
(1) $w(x,v)$ is concurrent with $r(x)$, or
(2) $w(x,v)$ precedes $r(x)$ with no intervening read or write of x with value $v' \neq v$.

Definition 5 (*Causal consistency*). An execution is *causally consistent* if the value returned by any read operation in the execution is live for that read.

A fast implementation for causal memory appears in [3,4]. Hence, causal memory is fast and Theorem 2 implies that it does not allow non-cooperative mutual exclusion algorithms. The above definition of causal memory includes only read and write operations. As in the case of weak consistency, if we add read-modify-write operations, then causal memory is no longer fast, and it supports non-cooperative mutual exclusion algorithms.

Processor consistency. Informally, processor consistency requires that the operations of each process will appear to be executed by all the processes in the order they were invoked [13]. For the formal definition, we need some more notation. Let σ be an execution or a sequence of operations. Denote by $\sigma \mid j$ the restriction of σ to operations invoked by p_j; denote by $\sigma \mid x$ the restriction of σ to operations on object x; denote $\sigma|writes$ the sequence of all write operations in a sequence of operations σ.

Given an execution σ, a sequence of operations σ is a *serialization* of σ if it is a permutation of $ops(\sigma)$.

Definition 6 (*Processor consistency*). An execution σ is *processor consistent* if for every process p_i there exists a legal serialization τ_i of $ops(\sigma)$ such that
(1) $\tau_i|i = \sigma|i$.

[4] This definition assumes that no value is written twice to the same memory location.

(2) for every process p_j,

$$(\tau_i|writes)|j = (\sigma|writes)|j.$$

The Pipelined RAM model of [15] provides processor consistency according to the above definition. Hence, processor consistency is fast and Theorem 2 implies that it does not allow non-cooperative mutual exclusion algorithms. Two alternative definitions of processor consistency, PCG and PCD,[5] were suggested in [2], together with non-cooperative mutual exclusion algorithms. Therefore, Theorem 2 implies that PCG and PCD are not fast, i.e., they do not have a fast implementation, in which the execution time of operations is $o(d)$. There is an explicit proof [10] that in any implementation of PCG and PCD, the execution time of operations is $O(d)$.

6. Discussion

Our results indicate that in order to solve fundamental synchronization problems, some sort of cooperation must exist, either at the implementation level, or within the programming model. Although both alternatives may be acceptable, designers must be aware of this tradeoff. In particular, for certain applications it is possible that the performance gains of the implementation would be out-weighted by the need to use more restrictive programming styles. In addition, our result highlights the benefits of mixed consistency conditions, e.g., hybrid consistency [5], weak ordering [1], and release consistency [12].

References

[1] S. Adve and M. Hill, Weak ordering - A new definition, in: *Proc. 17th Internat. Symp. on Computer Architecture* (1990) 2–14.

[2] M. Ahamad, R. Bazzi, R. John, P. Kohli and G. Neiger, The power of processor consistency, in: *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures* (1993) 251–260.

[3] M. Ahamad, P. Hutto and R. John, Implementing and programming causal distributed shared memory, Tech. Rept. TR GIT-CC-90-49, Georgia Institute of Technology, 1990.

[4] M. Ahamad, G. Neiger, P. Kohli, J. Burns and P. Hutto, Causal memory: Definitions, implementation and programming, *Distributed Comput.* **9** (1).

[5] H. Attiya and R. Friedman, A correctness condition for high-performance multiprocessors, in: *Proc. 24th ACM Symp. on the Theory Of Computing* (1992) 679–690; Revised version: Tech. Rept. #767, Dept. of Computer Science, The Technion.

[6] H. Attiya and R. Friedman, A correctness condition for high-performance multiprocessors, Tech. Rept. #767, Dept. of Computer Science, The Technion, 1993.

[7] H. Attiya and J. Welch, Sequential consistency versus linearizability, *ACM Trans. Comput. Systems* **12** (2) (1994) 91–122.

[8] E.W. Dijkstra, A solution of a problem in concurrent programming control, *Comm. ACM* **8** (1965) 569.

[9] R. Friedman, Implementing hybrid consistency with high-level synchronization operations, *Distributed Comput.* **9** (2).

[10] R. Friedman, Consistency conditions for distributed shared memories, Ph.D. Thesis, Dept. of Computer Science, The Technion, 1994.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, in: *Proc. 17th Internat. Symp. on Computer Architecture* (1990) 15–26.

[12] P. Gibbons, M. Merritt and K. Gharachorloo, Proving sequential consistency of high-performance shared memories, in: *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures* (19910 292–303.

[13] R.J. Goodman, Cache consistency and sequential consistency, Tech. Rept. 1006, Computer Science Dept., University of Wisconsin, Madison, WI, 1991.

[14] M. Herlihy and J. Wing, Linearizability: A correctness condition for concurrent objects, *ACM Trans. Programming Languages Systems* **12** (1990) 463–492.

[15] R. Lipton and J. Sandberg, PRAM: A scalable shared memory, Tech. Rept. CS-TR-180-88, Computer Science Dept., Princeton University, 1988.

[16] G.L. Peterson, Myths About the Mutual Exclusion Problem, *Inform. Process. Lett.* **12** (1981) 115–116.

[17] M. Raynal, *Algorithms for Mutual Exclusion* (MIT Press, Cambridge, MA, 1986).

[5] PCD is supported by the Stanford DASH multiprocessor [11]; the D in PCD stands for DASH and the G in PCG stands for Goodman.