A Node Count-independent Logical Clock for Scaling Lazy Release Consistency Protocol

Luciana Bezerra Arantes *, Bertil Folliot, and Pierre Sens

LIP6 Laboratory.
Université Pierre et Marie Curie.
4, Place Jussieu 75252 Paris Cedex 05, France.
[Luciana.Arantes, Bertil.Folliot, Pierre.Sens]@lip6.fr

Abstract. The use of per processor vector logical clocks in lazy release consistency (LRC) protocol implementation may restrict its scalability since the size of these clocks depends on the number of nodes of the system. We propose a new logical clock, the *barrier-lock*, whose concept is based on the causality of synchronization operations. Its size is proportional to the number of synchronization variables used by the application, being not affected by the number of nodes of the system.

1 Introduction

Distributed shared memory systems (DSM) simulate a shared-memory address space on top of loosely coupled multiprocessor systems. The physical distribution of data among the nodes as well as the consistency of the shared memory is made by the DSM layer, being completely transparent for the application. One of the most efficient DSM protocol for memory consistency is the lazy release consistency (LRC) [1]. By relaxing memory consistency model, LRC reduces the number of messages and data transferred among the processors. In LRC, synchronization operations set up the ordering of memory accesses and the propagation of shared data coherence information.

This paper presents our proposal for a LRC protocol whose implementation does not depend on the number of nodes of the system. This approach can be quite interesting for platforms with a large number of nodes, i.e., large-scale DSM systems. In these systems, scalability is an important feature.

One of the limitations for scaling current LRC implementations is the fact that, for controlling shared memory updates causality, they use logical clocks to timestamp synchronization operations [6] [4]. These logical clocks consist of vector structures which have one entry for each node (process) of the system. Hence, their size is proportional to the total number of nodes. We propose a new logical clock, whose size is independent of the number of nodes. We have named it the *barrier-lock* clocks. Its concept is based on operations on locks and barriers, i.e., the basic synchronization variables provided by most LRC DSM

* Ph.D. scholar from CAPES (Brazil)

systems. Its size is proportional to the number of synchronization variables used by the DSM application.

Section 2 of this paper gives an overview of the lazy release consistency protocol. Section 3 presents the *barrier-lock* logical clocks and some performance measures obtained with a first *barrier-lock* LRC DSM prototype. In section 4 some related works are discussed, while the last section summarizes the contributions of this work.

2 Lazy Release Consistency Overview

In lazy release consistency memory model, originally defined by TreadMarks [1], ordinary memory accesses are distinguished from synchronization ones. Synchronization operations are divided into acquire and release ones. The updates made by a process on its local shared data copy are propagated out to a second one only when the latter performs an acquire operation. This postponement reduces much of the communication required to make shared data consistent.

In LRC, the execution of each process is divided into intervals. A new interval begins at each acquire or release operation. Synchronization operations settle a causal ordering between intervals, which, in their turn, define the causality of shared memory updates. Intervals are partially ordered (**LRC happened-before**) as follows:

- intervals of the same process are totally ordered, i.e., if i and i' are intervals of the same process, and i occurs before i' in program order, then $i \rightarrow i'$;
- if i is an interval on process P_1 and i' is an interval of process P_2 then $i \rightarrow i'$ if i' begins with the acquire operation which is subsequent to the release operation which concluded i.
- if $i \rightarrow i'$ and $i' \rightarrow i''$, then $i \rightarrow i''$.

LRC controls partial order between intervals by using logical vector clocks, as defined by **Mattern** [6] and **Fidge** [4]. A vector clock timestamp is assigned to every interval. Each process keeps a local vector clock variable of N entries, where N is the total number of nodes (processes) of the system. A process j controls the intervals created by itself by using the jth entry of its vector clock variable. The other entries store the current knowledge that this process has of the intervals of the other processes. Thus, process Pj updates its vector clock v_j at each synchronization operation, based on the following rules:

r_1 : If it is an acquire operation and the acquirer P_j is different from the releaser Pr, whose local clock variable value is v_r, then:
$0 \leq k \leq N-1 : v_j[k] = max(v_j[k], v_r[k])$.
r_2 : $v_j[j] = v_j[j] + 1$.

Hence, at a remote acquire, the acquirer P_j sends its current clock value to the releaser P_r. This one sends back all the intervals "covered" by its own local vector clock, but not by P_j's, including the identification of the pages that have been modified in each interval. Each identification is stored in a structure called *write-notice*. When receiving a *write-notice*, the acquiring process invalidates its corresponding local page (invalidate protocol). The first access to an invalid page will cause a page fault. The faulting process will then ask for all the updates that it does not have yet, applying them in the order defined by the causality of the intervals. These updates come in the form of *diffs*, a word-by-word comparison between a copy of the original page and its last version. Each *diff* is associated with a *write-notice* which in its turn is associated with an interval.

LRC DSMs usually offer to application two types of synchronization variables: **locks** and **barriers**. The first ones are used to control accesses to shared memory critical regions, while the second ones to sequence the execution of the program. Both of them can be mapped onto an acquire and/or release operations. Operations on a **lock** can be directly mapped onto acquire or release operations: obtaining a lock corresponds to an acquire operation, while granting it corresponds to a release. A **barrier** is a synchronization point, executed in parallel by all processes, where each process incorporates consistency information (*write-notices*) from all other processes. After the execution of a barrier, the local vector clocks of all processes are set to the same value. Basically, when arriving at a barrier each process performs a release operation, sending to the barrier manager process the intervals that the latter does not have. On the other hand, the departure from the barrier is seen as an acquire, since the manager process sends to all other processes the intervals that they do not have yet. Thereby, if I_{bbar} is the set of intervals that happened *before* a barrier call and I_{abar} is the set of intervals that happened *after* it, we have:

$\forall i_j \in I_{bbar}$ and $\forall i_k \in I_{abar} : i_j \rightarrow i_k$ **(barrier property)**.

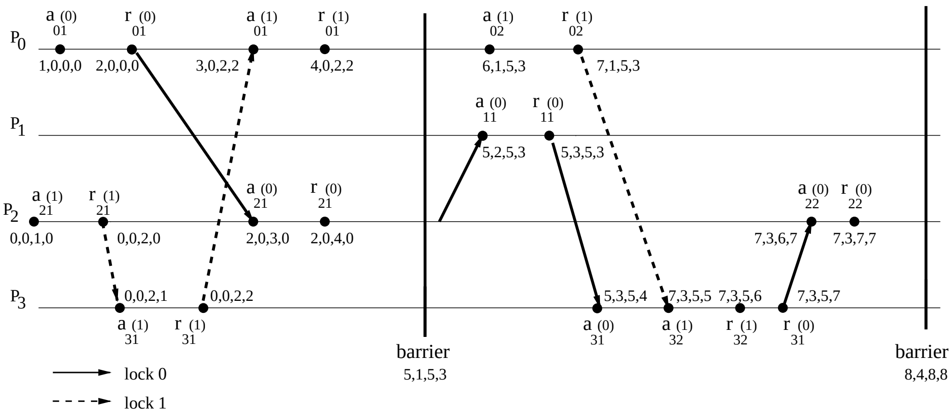

Fig. 1. LRC timing diagram based on vector clocks.

Figure 1 shows the partial ordering timing diagram corresponding to operations on 2 locks made by 4 processors. A barrier is called two times. For simplicity, shared memory operations or data structures (*write-notices*, *diffs*) are not shown. The notations $a_{ij}(l)$ and $r_{ij}(l)$ respectively represent the jth acquire or release operation of processor P_i on lock l. Each synchronization operation is timestamped with $v_i[0], v_i[1], v_i[2], v_i[3]$.

3 LRC Based on Barrier-lock Clocks

Vector clocks precisely control causality of LRC synchronization operations. However, their size is proportional to the number of processes of the system, which can be quite huge in a large-scale DSM. Due to LRC transitive propagation of updates, several of them can be included in a synchronization message.

As explained in the last section, at a barrier, processes set their local vector clocks to the same value. Thus, barriers could be seen as stop points for the restarting of a new set of lock operations, i.e., a program execution could be divided into *barrier-intervals* and each *barrier-interval* would be subdivided into *lock-intervals*. A new *barrier-interval* would begin at each barrier call, while a *lock-interval* at each acquire or release operation on a lock. This is exactly the idea behind the *barrier-lock* clock.

A *barrier-lock* timestamp is represented by the tuple $(b, vl)_i$, where b_i is a barrier call counter and vl_i is a per lock vector. Similar to per processor vector clocks, *barrier-lock* ones precisely control **LRC happened-before** partial ordering. At each barrier call, the counter b_i of all processes is incremented, while their lock vector vl_i is reset. On the other hand, for controlling processes' operations on locks within a single *barrier-interval*, the concept of the *poset-diagram*, as presented by Mattern in [6], is used. This diagram shows the logical relationship of events. If $a \to b$, than it is possible to follow a *path of causality* from a to b, which is easier seen in the *poset-diagram*. The timing diagram and the *poset-diagram* are isomorphic. Figure 2 illustrates the *poset-diagram*, related to the operations on locks 0 and 1 of the first *barrier-interval* of figure 1.

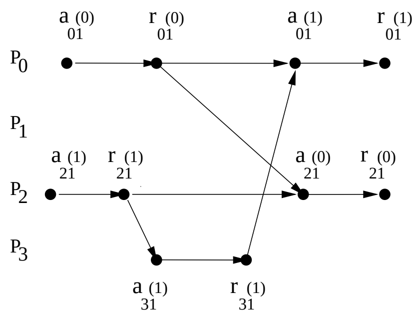

Fig. 2. Lock-intervals poset-diagram within a barrier-interval.

As locks are used to control processes' accesses to shared data critical regions, lazy release consistency protocol assures that a lock can only be held by one process at a time. Its behavior is like a token. This means that if two acquire or release operations on locks are mutually independent, they correspond to operations on different locks. It is guaranteed that two processes are not going to increment the same entry of their local variable vl concurrently. Therefore, for timestamping synchronization operations on locks, we can employ a vector variable where each of its entry is associated with a lock. Notice that the size of the *barrier-lock* clock is $L+1$, L being the number of locks used by the application. Very often this number is quite small (e.g. barrier-like programs). The LRC "lock token" behavior also justifies why, in spite of the fact that **Charon-Bost** [3] has proved that causality can only be characterized by vector clocks with N entries, we have managed to have a clock, that precisely captures causality, but whose size, $L+1$, can be smaller than the total number of processes (nodes) N. In other words, since locks are held in mutual exclusion, operations on them can be represented as the union of L chains (*paths of causality*). Hence, a vector of L entries is sufficient to characterize causality of synchronization operations on locks.

Let N be the number of processes (nodes) of the system and L, the number of locks used by the application. A *barrier-lock* clock timestamp of processor P_j is represented by the tuple $(b, vl)_j$, where vl has dimension L. A new timestamp is computed based on the following rules:

r_1 : If it is assigned to a new *barrier-interval*:
  $a:\ 0 \leq i \leq N-1 : \{0 \leq k \leq L-1 : vl_i[k] = 0\};$
  $b:\ 0 \leq i \leq N-1 : b_i = b_i + 1;$

r_2 : If it is assigned to a new *lock-interval*, corresponding to an acquire or release operation on lock l:
  a : in the case of an acquire operation in which the acquirer process Pj is different from the releaser Pr:
  $0 \leq k \leq L-1 : vl_j[k] = max(vl_j[k], vl_r[k]);$
  $b:\ vl_j[l] = vl_j[l] + 1.$

Figure 3 shows the same figure 1, using *barrier-lock* timestamps. The notation b-vl[0],vl[1] is used for each timestamp assigned to a synchronization operation.

For comparing two timestamps, (b, vl) and (b', vl'), the following relations are defined:

$$- \ (b, vl) < (b', vl') \Leftrightarrow \begin{cases} b < b' \\ or \\ (b = b') \ \ and \ \ (vl < vl') \end{cases}$$

$$- \ (b, vl) \ \| \ (b', vl') \Leftrightarrow \begin{cases} (b = b') \ \ and \ \ (vl = vl' = 0) \\ or \\ \neg(vl < vl') \ \ and \ \ \neg(vl' < vl) \end{cases}$$

The above relations guarantee that the *barrier-lock* clocks are also strongly consistent. So, for any two intervals i and i', respectively identified by (b, vl) and (b', vl') timestamps, we have:

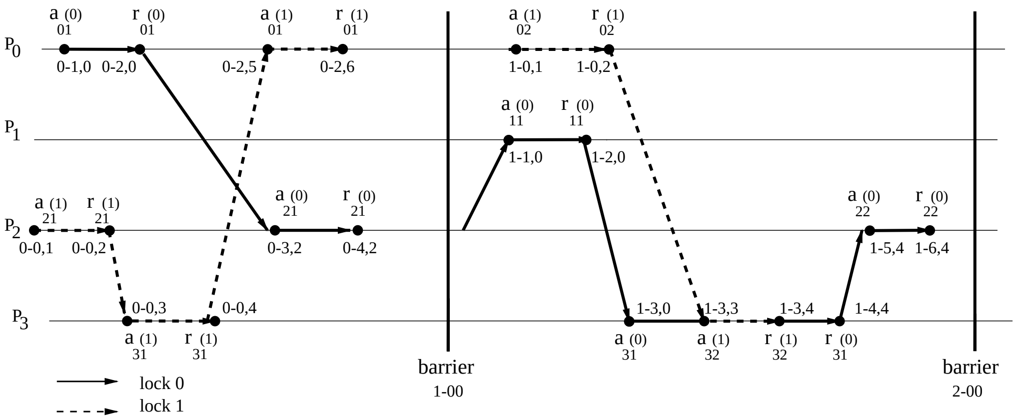

Fig. 3. LRC based on barrier-lock clocks.

$$i \rightarrow i' \Leftrightarrow (b, vl) < (b', vl')$$

Proof. If i and i' belong to different *barrier-intervals*, then $b < b'$ (rule r_1). If they belong to the same *barrier-interval* ($b = b'$), let i'', with (b, vl'') timestamp, be the latest interval that directly precedes i' and l the lock corresponding to the synchronization operation (acquire or release) which started interval i'. If i' and i'' belong to different processes then i' refers to the interval that begins with the acquire of l, which has been released with the ending of i''. Then $vl' = max(vl'', vl') + 1$ (rule $r_2.a$ and $r_2.b$). If they belong to the same process then $vl'[l] = vl''[l] + 1$ (rule $r_2.b$). Hence, in both case $vl'' < vl'$. We can apply this arguments to any pair of intervals belonging to all *paths of causality* that lead to i'. Due to the transitive dependency property of LRC intervals, we have that $vl < vl'$.

Conversely, if $b < b'$, intervals i and i' belong to different *barrier-intervals*. Then, based on the **barrier principle**, described in the previous section, $i \rightarrow i'$. On the other hand, if $b = b'$ then i and i' are *lock-intervals* within the same *barrier-interval*. Let (b, vl'') be the greatest timestamp smaller than vl' within this *barrier-interval*. If these timestamps have been created by different processes, then (b, vl') corresponds to the acquire of lock l, whose releasing operation has been identified by (b, vl''); if they belong to the same process, they correspond to totally order intervals and i'' occurred before i'. Then, based on the **LRC happened-before** partial ordering, i'' directly precedes i', which means that $i'' \rightarrow i'$. Applying this same arguments for each pair of *lock-intervals* belonging to all *paths of causality* that lead to i' and considering the transitive property of LRC protocol, we have $i \rightarrow i'$. □

3.1 Performance

In order to verify the feasibility of the *barrier-lock* clocks, we have implemented a prototype, replacing the traditional vector clocks by the *barrier-lock* ones in

TreadMarks software DSM, version 0.10.1. The tests have been made on top of 8 Sun-sparc-5 stations linked by a 100 Mbit/s Ethernet backbone with 5 well-known applications: SOR and TSP (distributed by TreadMarks [1]), IS and 3D FTT (NAS benchmark [2]) and Barnes-Hut(SPLASH benchmark [7]). The number of locks used by these applications are quite small (from 0 to 2). For showing the scalability of a LRC DSM, we have simulated a platform with a huge number of nodes by increasing the constant that specifies the number of processes in the system. We have then measured the number of bytes exchanged between the processes at synchronization operations. Figure 4 shows the ratio: *number of bytes of synchronization messages on top of barrier-lock LRC prototype / the number of bytes of synchronization messages on top of TreadMarks*.

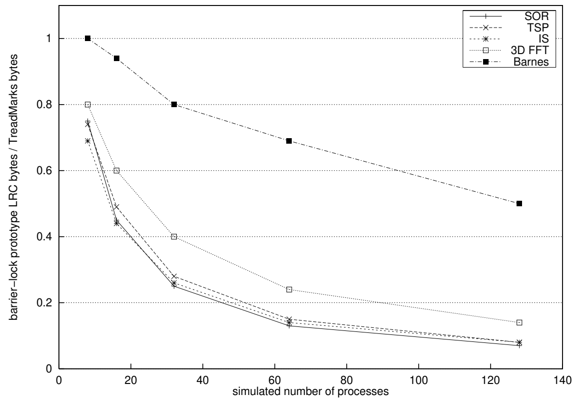

Fig. 4. Ratio of data exchanged at synchronization operations.

We can remark a considerable cut down in the amount of data exchanged at synchronization operations when the *barrier-lock* clocks are used, since all the applications use a small number of locks.

4 Related Works

TreadMarks [1] was the first DSM to implement the LRC memory model. Automatic Update Release Consistency (AURC) [10] and Home-based Lazy Release Consistency (HLRC) [10] are variations of LRC protocol. The three of them employ vector clocks to ensure the causality of synchronization operations. The last two adopt a "home-based" protocol in which updates of a page are eagerly propagated to a home node associated with the page.

Some authors have proposed a different implementation of Mattern and Fidge vector clocks. For instance, in **Singhal** and **Kshemkalyani** technique [8], a process sends to another only those entries of its vector clock that have been modified since they have last communicated to each other. Hence, the size of the message exchanged between processes is reduced. **Fowler** and **Zwaenepoel** [5] have proposed a vector clock implementation where each process only keeps direct dependencies on others. This implies in a considerable cut in memory storage and communication overhead. However, for capturing transitive causal relations, it is necessary to recursively trace causal dependencies.

There are some clocks, as the **plausible** ones [9], that can be constructed with a constant number of entries, independently of the number of nodes of the system. However, even if they offer a high level of ordering accuracy, they do not guarantee that certain pairs of concurrent events will not be ordered. Therefore, they are not appropriate for implementing LRC protocol, as this ordering can lead to unnecessary consistency operations and remote requests.

5 Conclusions

We have presented the *barrier-lock* clocks, whose size is proportional to the number of synchronization variables used by a DSM application. They have been modeled to precisely capture causality of synchronisation operations of lazy release consistency protocol. As their size is not affected by the number of nodes of the system, these clocks are quite appropriate for scaling DSM systems that provide such protocol. The proof that *barrier-lock* clocks can precisely control causal ordering of synchronization operations was presented in section 3, while the results of section 4 validate them.

References

1. C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.
2. D. Bailey, J. Barton, T. Lasinski and H. Simon. The NAS Parallel Benchmark. Technical Report 103863, NASA, July 1993.
3. B. Charon-Bost. Concerning the Size of Logical Clocks. *Information Processing Letters*, 39:11–16, July 1991.
4. C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.
5. J. Fowler and W. Zwaenepoel. Causal Distributed Breakpoints. In *the 10th International Conference on Distribute Computing Systems*, pages 131–41, 1990.
6. F. Mattern. Virtual Time and Global States in Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*, Elsevier (Holland), October 1988.
7. P.Singh, W. Weber and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-memory. Computer Architecture News, 20(1):5–44, March 1992.
8. M. Singhal, M. and A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 33:47–53, August, 1992.
9. F. Torres-Rojas, F. and M. Ahamad, Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. In *the 10th International Workshop on Distributed Algorithms*,Bologna(Italy), Octobre, 1996.
10. Y. Zhou, L. Iftode and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *the 2nd Symposium on Operating Systems Design and Implementation*, Octobre 1996.