An Efficient Logging Scheme for Lazy Release Consistent Distributed Shared Memory Systems

Taesoon Park
Department of Computer Engineering
Sejong University
Seoul 143-747, KOREA
tspark@cs.sejong.ac.kr

Heon Y. Yeom
Department of Computer Science
Seoul National University
Seoul 151-742, KOREA
yeom@arirang.snu.ac.kr

Abstract

We propose a low-overhead logging scheme for the distributed shared memory system based on the lazy release consistent memory model. In the proposed scheme, stable logging is performed when a lock grant causes an actual dependency relation between the processes, which significantly reduces the logging frequency. Also, instead of making a stable log of the accessed data items, a process logs stably only some access information, and the accessed data items are saved in the volatile log. For the recovery from a failure, the correct version of the accessed data items can be effectively traced by using the logged access information. As a result, the amount of logged information is also reduced.

Index Terms – *Checkpointing, Distributed shared memory system, Fault tolerance, Message logging, Lazy release consistency, Rollback-recovery.*

1. Introduction

Distributed shared memory(DSM) systems[6] provide a simple mean of programming for the networks of workstations, which are gaining popularity due to their cost-effective high computing power. However, as the number of workstations participating in a DSM system increases, the probability of failure also increases. Hence, for long-running applications, it is important for the system to be recoverable so that the processes do not have to restart from the beginning when there is a failure [12].

An approach to provide fault-tolerance for the system is to use checkpointing and rollback-recovery. By periodically saving the intermediate system state into a safe storage, the system can resume from one of the checkpointed states, after a failure, instead of restarting from the initial state. In the DSM systems, the computational states of the processes become dependent on one another by accessing common data items. Such computational dependency makes the processes roll back together to a consistent recovery line, when a process rolls back after a failure. Hence, to prevent the *domino effect*[8] in which the processes have to roll back recursively to reach a consistent recovery line, the checkpointing activities of the related processes should be carefully designed.

One solution to cope with the domino effect is to log the accessed data items on a stable storage in addition to the checkpoint [9]. Hence, a process affected by a failure can regenerate the same computation by replaying the logged data items; that is, the rollback of one process does not affect other processes. However, since the logging itself may cause nonnegligible overhead, many solutions to reduce the logging overhead have been suggested for the sequentially consistent DSM system[4, 7, 11].

The memory consistency model of the DSM system is an important factor to characterize the inter-process dependency. Hence, for the lazy release consistency(LRC) model, which is one of the relaxed memory models[1, 3, 5], other requirements are needed to reduce the logging overhead. In [11], the process should make a stable log of the accessed data items and their usage information before it releases any lock or transfers any data items to other processes. Another scheme proposed in [2] requires the data items and their usage information to be logged in the volatile memory of the writer process. Hence, this scheme removes the overhead of the stable logging, however, it can not handle the concurrent failures which affect both of the reader and the writer processes.

In this paper, we propose a new logging scheme for a LRC based DSM system, which incurs much low logging overhead compared to the scheme in [11] and tolerates multiple site failures unlike the scheme in [2]. In the LRC model, the dependency relation between the processes occurs when the write and the read operations are explicitly

synchronized by lock operations. Hence, we suggest that the logging should be performed when the actual dependency is formed by synchronization operations, which reduces the frequency of the stable logging. Also, the data items are logged into the volatile memory of the writer process, and only the access information such as the vector time used in LRC model is logged stably. As a result, the amount of the stable log can also be drastically reduced.

2. Background

A DSM system consisting of a number of nodes connected through a communication network is considered. Each node consists of a fail-stop processor[10], a volatile main memory and a stable secondary memory. The processors in the system do not share any physical memory, and communicate by message passing. The communication subsystem is assumed to be reliable. The failures considered in the system are assumed to be transient and a number of concurrent node failures may happen in the system.

The computation of a process in the system is assumed to be *piece-wise deterministic*; that is, the sequence of computational states of a process is fully determined by the data values provided for a sequence of read and write operations. We assume the invalidation-based lazy release consistency(LRC) memory model[5], which allows copies of the same data on different nodes to be temporarily inconsistent during the computation. To guarantee the correct execution order between the conflicting operations, synchronization operations are used. The lock acquire and lock release operations are used to make the execution order of the conflicting operations sequential, and a barrier operation is used to synchronize the execution timing of every process.

We define a *state interval*, denoted by $I(i,a)$, as the computation sequence between the $(a-1)^{th}$ and the a^{th} synchronization operations of a process p_i, where $a > 1$ and the 0^{th} synchronization operation means the initial state of p_i. Then, in the LRC based DSM system, the computational dependency between the state intervals is defined as follows:

Definition: A state interval $I(i,a)$ is dependent on another state interval $I(j,b)$ if one of the following conditions are satisfied:
(1) $i = j$ and $a = b + 1$.
(2) $I(j,b)$ ends with a $release(x)$ and $I(i,a)$ begins with an $acquire(x)$, and p_i in $I(i,a)$ accesses a data value written by p_j in $I(j,b)$.
(3) $I(j,b)$ ends with a $barrier(x)$ and $I(i,a)$ begins with a $barrier(x)$, and p_i in $I(i,a)$ accesses a data value written by p_j in $I(j,b)$.
(4) $I(i,a)$ is dependent on $I(k,c)$ and $I(k,c)$ is dependent on $I(j,b)$. □

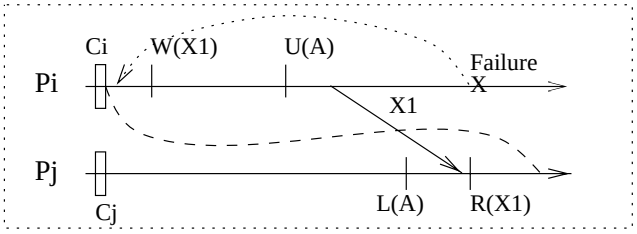

Figure 1. An Inconsistent Recovery Line

Such dependency relation between the state intervals may cause possible inconsistency problems during the rollback and recovery. Figure 1 shows a typical example of inconsistent rollback recovery case[7]. The notations, $R(X_1)$ and $W(X_1)$, in the figure represent the read and the write operations on a data page X_1, respectively, and the notations $U(A)$ and $L(A)$ represent the release and the acquire operations on a lock A, respectively. Now, suppose the process p_i in Figure 1 rolls back to its latest checkpoint C_i due to its failure but it cannot regenerate the same computation for $W(X_1)$. Then, the consistency between p_i and p_j becomes violated since p_j's current computation depends on p_i's computation invalidated by a rollback. Such a case is called an *orphan message* case and a process is said to recover to a *consistent recovery line*, if any process in the system is not involved in the orphan message case after the rollback recovery.

3. Protocol Description

3.1. Logging Protocol

Processes performing the piece-wise deterministic computation can regenerate the same computation sequence, if the data values used for the shared memory accesses can be logged and replayed at the same access points [9]. Hence, for the consistent recovery, each process must log two types of information; one is the contents of the accessed data page and the other is the information indicating the computational point at which the page has been accessed.

The data page can be logged either at the process which accessed it (a *reader process*) or at the process which produced it (a *writer process*). If the reader process logs the page, the log must be stable since it has to tolerate the reader's own failure. However, if the writer process logs the page, the volatile log can be used, since the page is to be retrieved after the reader's failure not for its own failure. In case that the writer process fails, it can regenerate the same contents of the page if it performs correct recovery. Moreover, since a data value written by a process is usually read by many processes, it is more efficient for one writer to log the value, instead of many readers logging it.

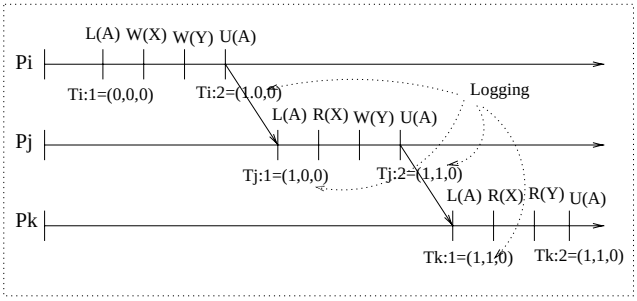

Figure 2. An Example of Logging Operations

For the volatile logging of data pages, the *diff* structure which the LRC memory model provides can be used. In the LRC model, when a process writes on a data page, it first creates a copy of the page, called a *twin*, and then performs the write operation. Later, when another process requests the data page, the modified page is compared with its twin, and the modified portion of the page, called a *diff*, is sent to the requesting process. The requesting process collects the diffs from every process which has written on that page, and applys them in the chronological order to create the up-to-date version of the page. Such a diff structure is maintained at the volatile storage of the writer process until the system periodically discards the diffs which are no longer required, which is called *garbage collection*. Hence, in the proposed scheme, the diff structure maintained in the volatile storage of the writer process is used as the volatile log of data values and only the diffs discarded by the garbage collection are saved into the stable storage as parts of the checkpoint.

Another information to be logged is the association of each data access point with the correct version of the data. For the efficient implementation, the vector time employed in the LRC model is directly used to represent the information. A vector time T_i of process p_i is an array of integers, $T_i = (t_{i1}, ..., t_{ii}, ..., t_{in})$, where n is the number of processes in the system. The value of t_{ii} in T_i is incremented by one when p_i releases a lock following any write operation and each t_{ik} is updated as maximum of t_{ik} and t_{jk} in T_j when p_i acquires a lock from the last releaser of the lock, p_j. As a result, the vector time associated with the synchronization operation reflects the causal ordering between those operations.

When a process accesses a data page at vector time T, the page must reflect all and only the diffs made before the time T. Hence, if the values of the vector time associated with the diffs and the data access points are logged, the correct version of the shared data can be retrieved during the recovery. Moreover, since the vector time can be updated only when the synchronization operation is performed, the logging needs to be performed only for the synchronization operation. To log the vector time associated with each synchronization operation, each process p_i maintains the *synchronization operation counter*, denoted by Syn_i, which indicates the number of synchronization operations happened at p_i. When p_i performs a synchronization operation, the new vector time value is logged with the value of Syn_i, if the vector time of p_i has been updated.

Figure 2 shows an example of the logging for the system consisting of three processes, p_i, p_j and p_k. $T_{i:a} = (i, j, k)$ in Figure 2 denotes the vector time of p_i with Syn_i value, a. Since the vector time needs to be logged only when the value is updated, there are four logging operations performed in the system. From the figure, we can easily see how the processes can recover from a failure. For example, suppose that process p_k fails and it has to recover up to the release operation of lock A (denoted by $U(A)$). Then, from the log, p_k can retrieve the information that its first lock acquire operation (denoted by $L(A)$) is associated with the vector time (1,1,0), and hence it can know that for R(X) and R(Y), it has to fetch the diffs made before the vector time (1,1,0). Since each diff in the system carries the vector time of its creation, p_k can safely include the diffs made by p_i and p_j before vector time (1,1,0) for its read operations.

Unlike the data page contents (diffs), the access information must not be logged on the volatile storage of the writer process, since such information cannot be regenerated after the writer's failure. In case that the reader process makes a stable log of such information, the frequency of the logging operation must be an important performance factor, even though the amount of information to be logged is very small. To reduce the logging frequency, in the proposed scheme, the vector time associated with the $Sync$ value is temporarily logged on the volatile storage of the reader process and stable logging is performed only when the process has a new dependent.

If a process has lost some state intervals due to a failure, the processes dependent on the lost state intervals have to roll back together to reach a consistent recovery line[7]. However, if a process has lost some state intervals but there is no process dependent on them, then arbitrary recomputation of the process does not cause any inconsistency problem with other processes; that is, there is no need for the logging of those state intervals. Hence, a process can delay the logging of the access information for some state intervals until having a process dependent on those intervals. From the definition, a state interval $I(i, a)$ of process p_i can have any dependent state interval $I(j, b)$ on another process p_j only if the following sequence of operations are performed: $W(X)$ in $I(i, a)$, $U(A)$ (or $barrier(A)$) following $I(i, a)$, $A(A)$ (or $barrier(A)$) at p_j following $U(A)$ of p_i, $I(j, b)$ following $A(A)$ at p_j and then $R(X)$ in $I(j, b)$.

To efficiently perform stable logging, in the proposed scheme, each process p_i maintains the *write-since-last-logging* flag, denoted by WSL_i, which indicates whether

there has been a write operation since the last stable logging of p_i. The WSL_i is set to one when p_i performs a write operation and reset to zero when p_i performs a stable logging. Process p_i performs stable logging only when p_i grants a lock to another process and WSL_i is equal to one. Hence, in Figure 2, volatile logging is performed at four synchronization points, however, stable logging is performed only at two synchronization points, such as $U(A)$ of p_i and $U(A)$ of p_j.

3.2. Checkpointing and Recovery

Each process periodically takes a checkpoint to reduce the amount of recomputation during the recovery. A checkpoint includes the intermediate state of the process and the contents of data structures required for the memory consistency and the logging protocols. The checkpointing activities among the related processes need not to be performed in a coordinated way.

When a process recovers from a failure, it first restores its state from the latest checkpoint and begins the recomputation as follows:

At the lock or barrier operation time: The process increments its Syn value by one and resets its current vector time as the one logged with the new Syn value.

At the data page access miss: The process broadcasts the data page request to the other processes with its current vector time. Each of the other processes replies with the diffs of the page it has produced before the broadcast vector time. The recovering process applys the collected diffs on the data page in the order to create an up-to-date version. The created version can be used until it is invalidated. For the efficient invalidation, the write notices obtained at the lock acquire time is used. The write notices include the identifiers of the data pages which have been invalidated since the last acquire operation of the process. Hence, by logging the write notices with the vector time at the acquire point, the process can invalidate only the necessary pages during the recomputation. Logging the write notices may slightly increase the amount of the log, however, it does not affect the logging frequency.

Since for each synchronization operation, the process can retrieve the same vector time as the one created before the failure, it can retrieve the same diffs from other processes and create the same data page for its read and write operations.

Theorem: The recovery line produced under the proposed scheme is consistent.

Proof: For the recovery line to be consistent, there must not be an orphan message case. In the LRC based DSM system, the vector time associated with each state interval can be used to identify the diffs created during that interval and also to identify the access point of the diffs. If the vector time value is saved with a unique identifier for each state interval, the correct diffs can be retrieved after the failure. Hence, it is enough to prove that any vector time value related to a diff can be stably logged before the diff is actually accessed by another process. In the proposed scheme, a process performs the stable logging when it receives an acquire request and it has performed a write operation since its last stable logging. Hence, every access information related to a diff can be stably logged before any acquire is granted, after which point the diff can be accessed. Therefore, for any dependency relation, the correct version of the data value can be retrieved after a failure and the recovery line is consistent. □

4 Performance Study

To evaluate the performance of the proposed scheme, we have performed the simulation with a set of traces obtained from the real parallel program execution, and compared the behavior of the proposed scheme with the existing logging schemes.

Shared-read logging(SRL)[9] : Each process logs the data value to the volatile memory at each access point, and the volatile log is flushed into the stable storage, when the process transfers a data value to another process.

Shared-access tracking(SAT)[11]: Each process logs the data value to the volatile memory when it is transferred from another process and logs the write notices at the lock acquire time. Stable logging is performed, when the process transfers a data value or a lock grant message to another process.

Reduced-stable logging(RSL): It is what we propose in this paper and each process makes the volatile log of the vector time and write notices at each synchronization point. Stable logging is performed, when a new dependency relation is actually formed.

Two performance measures are used; one is the amount of diffs which have to be logged at the processes and the other is the frequency of stable logging. The traces used in the simulation contain references produced by a 32-processor MP, running the following four programs: fft, barnes-hut, mp3d and radix.

Figure 3 shows the frequency of stable logging, which indicates the number of accesses to the stable storage not only for the data values, but also for the access information. Comparing the three schemes, SAT scheme shows the highest logging frequency in most cases, since in this scheme, not only the data value transfer but also the write notice transfer causes the logging. In SRL scheme, only the data value transfer causes the logging and in the proposed scheme, only the write notice transfer causes the logging. Comparing the data value transfer, the frequency of the write notice transfer is much low. Moreover, in the pro-

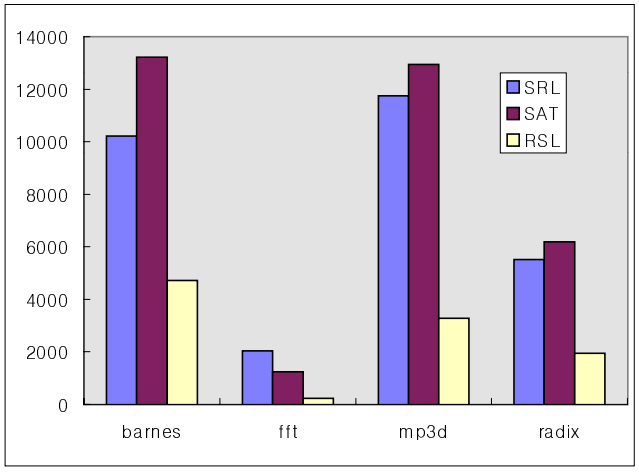

Figure 3. The Logging Frequency

posed scheme, stable logging is not performed for every write notice transfer, but performed only when there has been a write operation. Hence, compared with the other scheme, about 50% to 90% of the logging frequency has been reduced in the proposed scheme.

Table 1 shows the amount of stably logged data values. In SRL scheme, all data values used for read and write operations are logged. Meanwhile, in SAT scheme, only the data values transferred after the access misses are logged, and hence the amount of data log can be much smaller. However, the logging amount in SAT scheme is still non-negligible compared to the one in the proposed scheme, in which no stable logging is required for the data values.

5. Conclusions

In this paper, we have proposed a new logging scheme for a LRC based DSM system, which tolerates multiple failures with minimal logging overhead. In the proposed scheme, the data value produced by a write operation is logged at the volatile storage of the writer process. To efficiently trace the correct version of the data value during the recomputation, the vector time provided by the LRC model is used. Stable logging is performed only for the vector time value when the dependency relation is actually formed. Hence, the amount and the frequency of the stable logging is significantly reduced.

	SRL	*SRL*	*RSL*
barns	1907757	14756	0
fft	6818013	6657	0
mp3d	4351950	42641	0
radix	5001040	10602	0

Table 1. The Logging Amount

References

[1] S.V. Adve and M.D. Hill. Weak ordering–A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture*, May 1990.

[2] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight logging for lazy release consistent distributed shared memory. In *Proc. of the USENIX 2nd Symp. on Operating Systems Design and Implementation*, October 1996.

[3] K. Gharachorloo, D.E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture*, May 1990.

[4] S. Kanthadai and J.L. Welch. Implementation of recoverable distributed shared memory by logging writes. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems*, May 1996.

[5] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 18th Annual Int'l Symp. on Computer Architecture*, May 1992.

[6] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, September 1986.

[7] T. Park, S.B. Cho, and H.Y. Yeom. An efficient logging scheme for recoverable distributed shared memory systems. In *Proc. of the 17th Int'l Conf. Distributed Computing Systems*, May 1997.

[8] B. Randell, P.A. Lee and P.C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165, June 1978.

[9] G.G. Richard III and M. Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proc. of the 12th Symp. on Reliable Distributed Systems* , October 1993.

[10] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. on Computer Systems*, 1(3):222–238, August 1983.

[11] G. Suri, B. Janssens, and W.K. Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. In *Proc. of the 25th Annual Int'l Symp. on Fault-Tolerant Computing* , June 1995.

[12] K.L. Wu and W.K. Fuchs. Recoverable distributed shared memory. *IEEE Trans. on Computers*, 39(4):460–469, April 1990.