An Efficient Causal Logging Scheme for Recoverable Distributed Shared Memory Systems

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

Inseon Lee and Heon Y. Yeom
Department of Computer Science
Seoul National University
Seoul 151-742, KOREA
{inseon, yeom}@arirang.snu.ac.kr

ABSTRACT

This paper presents a causal logging scheme for the lazy release consistent distributed shared memory systems. Causal logging is a very attractive approach to provide the fault tolerance for the distributed systems, since it eliminates the need of stable logging. However, since inter-process dependency must causally be transferred with the normal messages, the excessive message overhead has been a drawback of this approach. In order to achieve an efficient implementation of causal logging for the distributed shared memory system, data structures and operations supported by the lazy release consistency memory model are utilized. As a result, the causal logging for the lazy release consistent distributed shared memory system can be implemented by adding the minimum information for the dependency tracking.

To evaluate the performance of the proposed scheme, the proposed logging scheme has been implemented on top of the CVM distributed shared memory system. The experimental results show that the logging operation requires only 1% - 4.4% increases in the execution time.

Keywords—Checkpointing, Distributed shared memory systems, Fault tolerant systems, Message logging, Lazy release consistency, Rollback-recovery.

An earlier version of this work has appeared in the Proceedings of the ACM Symposium on Applied Computing, 2000.
1 Introduction

Distributed shared memory (DSM) systems [12] transform the existing network of workstations into a powerful shared-memory parallel computer and due to the cost-effective high computing power, the DSM systems are gaining popularity. However, as the number of workstations participating in a DSM system increases, the probability of failure also increases and even a single failure could result in the loss of total computation for the long-running applications. Hence, for the DSM systems to be of any practical use, it is important for the system to be recoverable so that the processes do not have to restart from the beginning even in case of failure occurrences [26].

An approach to provide fault-tolerance for a DSM system is to save intermediate states of a process periodically into the stable storage, which is called a checkpoint, so that the system can recover from one of the checkpoints after a system failure occurs. However, in a DSM system, the computational states of the processes become dependent on one another by accessing the common data item. In the sequentially consistent DSM systems[12], the process accessing a data item becomes dependent on the process which has produced that item, and in the DSM systems based on the relaxed memory models[1, 7, 10], the dependency relation is explicitly synchronized by the lock or barrier operations. Such inter-process dependency may have to make a group of related processes roll back recursively to reach a consistent recovery line in case of a failure, which is called a domino effect[17]. In the worst case, the processes eventually have to roll back to their initial states, in spite of the checkpointing efforts.

One solution to cope with the domino effect is to coordinate the checkpointing activities for the related processes. One way of checkpointing coordination is to synchronize the related processes to take consistent checkpoints together[4, 8, 14]. Another way of coordination is the communication-induced checkpointing, in which a process takes a checkpoint whenever a new dependency relation occurs with another process [23, 25, 26]. Since the checkpointing coordination under both approaches results in the system-wise consistent recovery line, the processes may not be involved in any domino effect. However, the approaches may have to suffer from the overhead caused by the computation blocking during the coordination or too frequent checkpointing activities.

Another solution is to use the logging in addition to the independent checkpointing [18]. The logging technique is to stably save the data items which have been accessed by a process, in the accessing order. Hence, after a failure occurs, the process can restore the latest checkpoint and re-build the same computational states with the logged data items. Employing the logging with periodic checkpointing, a
process can recover from a failure with the small amount of recomputation, and the failure-recovery of
a process does not affect other dependent processes for the piece-wise deterministic computation. Many
logging schemes have been suggested, which can be categorized into three approaches.

In the reader-based logging schemes [18, 24], each process logs the data items it has accessed into
the stable storage in the access order, so that the process can retrieve the logged data items at the same
computational points after the failure-recovery. However, these schemes may require the high logging
overhead during the failure-free operations. The writer-based logging schemes [9, 15, 16] utilize the
volatile log space of the writer process for each data item, and only the related access information is
logged at the stable storage of the corresponding reader as in [9] or the writer itself as in [15, 16]. As a re-
result, the logging overhead can be reduced, however, the overhead of stable logging is still non-negligible.
One solution suggested in [6] uses only the volatile log of the reader to store the access information.
This scheme removes the overhead of stable logging, however, concurrent failures of multiple processes
cannot be tolerated.

Causal logging is another logging approach which is gaining a lot of attention for the message-
passing based distributed computing systems [2]. In the causal logging technique, the writer-based
volatile logging of data items is performed and the access information is logged at the volatile storage
of the dependent processes. Since this scheme completely eliminates the needs for stable logging, the
logging overhead can significantly be reduced compared to the stable logging approaches. Also, since
the storage of the dependent processes are utilized, concurrent and multiple failures can be handled.
However, since the log of the access information has to be causally spread over the dependent processes
for the log to be stable, the processing overhead for each message carrying the log must be significant.

This paper presents a new logging scheme for the DSM system based on the lazy release consis-
tent(LRC) memory model[10]. The proposed scheme adopts the causal logging approach and to reduce
the message processing overhead incurred by the causal logging, the data structures and the operations
supporting the LRC memory model are utilized for the causal logging operations. Especially, the facil-
ities for the dependency tracking in the LRC memory model, such as diff structure, write notices, and
vector clocks, can support the causal logging very well. Moreover, the logging is performed only at the
synchronization points instead of every message passing point.

However, if the existing vector clock is used to trace the access information in the causal logging,
the message overhead must be high for the system with a large number of processes, because of the
large size of the vector clock. Also, for the barrier operations, the dependency relation between every pair of processes must be logged in the form of a vector clock, and hence, the logging overhead must be significant. In the proposed scheme, instead of logging the vector clock for each synchronization operation, the sufficient and necessary information to recreate the corresponding vector clock is inserted into the existing write notice structures. As a result, the additional information carried by a message becomes less than two integers for each synchronization interval, and hence, a very efficient causal logging scheme can be implemented by adding the minimum information. Moreover, the size of the added information is independent of the number of processes in the system, and hence, the new scheme can be very efficient for the systems of a very large size.

To evaluate the performance of the proposed scheme, we have implemented the logging scheme on top of CVM distributed shared memory system. The experimental results show that the logging operation requires only 1% - 4.4% of the execution time increases, and the logging overhead with the optimized vector clock achieves 35% - 69% reduction in the logging cost compared with the scheme employing the existing vector clocks.

The rest of this paper is organized as follows: In Section 2, the system model and the definition of stable logging for the consistent recovery are presented and in Section 3, the proposed causal logging and recovery schemes are presented with their correctness. The performance of the proposed scheme is discussed with the experimental results in Section 4 and Section 5 concludes the paper.

2 Preliminaries

2.1 System Model

A DSM system consisting of a number of fail-stop nodes [19], connected through a communication network, is considered. Each node consists of a processor, a volatile main memory and a non-volatile secondary storage. The processors in the system do not share any physical memory and communicate by message passing. However, the system can be viewed as a set of processes communicating through a logically shared memory space, which is actually composed of a set of distributed local memory spaces of the nodes. The distributed shared memory space is assumed to consist of a set of fixed-size pages. The communication subsystem is assumed to be reliable, however, no assumption is made on the message delivery order between two processes. The failures considered in the system are transient and a number
of concurrent node failures may happen in the system.

The computation of a process is assumed to be \textit{piece-wise deterministic}; that is, the computational states generated by a process is fully determined by a sequence of data values provided for the sequence of read and write operations. For the memory consistency, the invalidation-based LRC memory model is assumed. A number of different memory semantics for the DSM have been proposed, however, in this paper, we focus on the LRC memory model for its enhanced performance. In the LRC memory model, the synchronization operations, such as the \textit{acquire}, \textit{release}, and \textit{barrier}, are used to guarantee the correct execution order between the conflicting operations. Hence, when a process writes on a data page and releases a lock, another process can read the updated page after it acquires the corresponding lock and invalidates the old copy of the page. The LRC memory model also employs a multiple-reader, multiple-writer protocol, in which more than one process may have writable copies in their local memory if the writing portions of the page are not overlapped.

\subsection{2.2 Consistent Recovery}

A process can logically be viewed as a sequence of state transitions from the initial state to the final state, each of which is caused by an event. Let $e^\alpha_i$ denote the event which causes the $\alpha$-th state transition of a process $p_i$. Then, a sequence of events, denoted by $E_i = (e^1_i, e^2_i, \ldots, e^\omega_i)$, is called a computation performed by $p_i$. Also, $S_i = (s^0_i, s^1_i, \ldots, s^\omega_i)$ denotes the sequence of states generated by $E_i$, where $s^0_i$ and $s^\omega_i$ are the initial and the final states of $p_i$, respectively, and each $s^\alpha_i$ is the state generated by $e^\alpha_i$. Under the assumption of piece-wise deterministic computation, only the external events, which affect other processes or are affected by other processes, are considered for the consistent recovery. In the DSM systems, \textit{reads} and \textit{writes} are those events; and two types of interactions can be considered regarding those events. One is the \textit{dependency relation} and the other is the \textit{precedency order}.

A process $p_i$ becomes dependent on another process $p_j$ by reading a data item written by $p_j$. Let $W(x^n)$ denote the $n$-th write event on a data item $x$ regardless of the writer process, and $R(x^n)$ denote the read event on the same data item with the value written by $W(x^n)$. Then, the dependency relation can formally be defined as follows:

\textbf{Definition 1:} A state $s^\alpha_i \in S_i$ is \textit{dependent on} another state $s^\beta_j \in S_j$ if one of the following conditions is satisfied, and the relation is denoted by $s^\beta_j \rightarrow s^\alpha_i$.

(a) $i = j$ and $\alpha = \beta + 1$. 

4
Definition 1.(a) states the dependency regarding the event time order within a process, Definition 1.(b) denotes the inter-process dependency caused by writing and reading a common data item, and Definition 1.(c) states that the dependency relation is transitive. The relation between two events, $e_i$ and $e_j$, is also denoted by $e_j \rightarrow e_i$, if the relation, $s_j \rightarrow s_i$, holds. The precedency order is the relation between two write events, which indicates the actual timing order of the write events on the same data item, and the relation can formally be defined as follows:

**Definition 2:** A write event $e_i = W(x^n)$ precedes another write event $e_j = W(y^{n'})$ if one of the following conditions is satisfied, and the relation is denoted by $W(x^n) \rightarrow W(y^{n'})$.

(a) $x = y$ and $n' = n + 1$.
(b) For any $W(z^{n''})$, $W(x^n) \rightarrow W(z^{n''})$ and $W(z^{n''}) \rightarrow W(y^{n'})$. □

The dependency relation between the processes may cause the inconsistency problem when a process recovers from a failure. Figure 1 shows the typical example of the inconsistent recovery stated in [5]. Suppose that the process $p_i$ in the figure should roll back to its latest checkpoint $C_i$ after a failure, however, $p_i$ cannot regenerate the same data item value for $W(x^1)$. This situation may happen if $p_i$ cannot retrieve the same data item value for $R(x^1)$. Then, the consistency between $p_i$ and $p_j$ becomes violated since $p_j$’s current computation depends on the $p_i$’s invalidated states. Such a case is called an orphan state case and a process is said to recover to a consistent recovery line, if any process in the system is not involved in any orphan state case after a rollback recovery.

Another problem of the page-based DSM system is that the precedency order may lead to a potential dependency relation, which may not be noticed by the processes and may cause the inconsistent recovery
as shown in Figure 2. In the figure, two dependency relations can be noticed; one is the explicit dependency from \( p_j \) to \( p_k \), and the other is the implicit dependency from \( p_i \) to \( p_k \). Since a write operation on a data page does not mean the creation of the whole page, \( p_k \) has a chance to read a data value updated by \( p_i \) not by \( p_j \). In case that both of \( p_i \) and \( p_j \) has updated the same portion of the page, the data value updated by \( p_j \) must be read by \( p_k \), and there may not be any dependency between \( p_i \) and \( p_k \). The problem here is that such potential dependency between \( p_i \) and \( p_k \) may not be noticed by observing only the dependency relation between the processes. However, if \( p_i \) fails and cannot regenerate the same states after \( W(x^{1}) \), \( p_j \) also cannot regenerate the correct data page in case of \( p_j \)'s failure and the recovery between \( p_j \) and \( p_k \) can be inconsistent.

Hence, we define the **potential dependency**, which includes the dependency relation and the precedence order, as follows:

**Definition 3:** A state \( s_{i}^{\alpha} \in S_i \) is potentially dependent on another state \( s_{j}^{\beta} \in S_j \) if one of the following conditions is satisfied, and the relation is denoted by \( s_{j}^{\beta} \rightarrow_{p} s_{i}^{\alpha} \).

(a) \( s_{j}^{\beta} \rightarrow s_{i}^{\alpha} \).

(b) \( e_{j}^{\beta} = W(x^{n}) \) and \( e_{i}^{\alpha} = W(y'') \) and \( W(x^{n}) \rightarrow w W(y'') \).

(c) For any \( s_{k}^{\gamma}, s_{j}^{\beta} \rightarrow_{p} s_{k}^{\gamma} \) and \( s_{k}^{\gamma} \rightarrow_{p} s_{i}^{\alpha} \). □

In the LRC based DSM system, the potential dependency between the processes is more explicitly restricted by the synchronization operations and the information regarding the synchronization operation can be used for the efficient recovery. Let \( I_{i}^{\alpha} \) denote the \( \alpha \)-th state interval of a process \( p_i \), which is the sequence of states between the \((\alpha-1)\)-th and the \( \alpha \)-th synchronization operations of \( p_k \), where \( \alpha > 1 \) and the 0-th synchronization operation means the initial state of \( p_i \). Then, in the LRC based DSM system, the effect of a write operation performed in \( I_{i}^{\alpha} \) can be viewed by another process \( p_j \) in the state interval
either after a release and an acquire operations on the same lock are properly performed between the two state intervals or after $p_i$ and $p_j$ performs the same barrier operation between them. Hence, the dependency relation between the state intervals can be defined as follows:

**Definition 4:** A state interval $I_i^\beta$ is dependent on another state interval $I_j^\beta$ if one of the following conditions is satisfied, and the relation is denoted by $I_j^\beta \rightarrow I_i^\alpha$.

(a) $i = j$ and $\alpha = \beta + 1$.

(b) $I_j^\beta$ ends with $\text{rel}(L)$ and $I_i^\alpha$ begins with $\text{acq}(L)$ and for any $s_j^{\beta'} \in I_j^\beta$, $s_i^{\alpha'} \in I_i^\alpha$, $s_j^{\beta'} \rightarrow_p s_i^{\alpha'}$.

(c) $I_j^\beta$ ends with $\text{barrier}(L')$ and $I_i^\alpha$ begins with $\text{barrier}(L')$ and for any $s_j^{\beta'} \in I_j^\beta$, $s_i^{\alpha'} \in I_i^\alpha$, $s_j^{\beta'} \rightarrow_p s_i^{\alpha'}$.

(d) For any $I_k^\gamma, I_j^\beta \rightarrow I_k^\gamma$ and $I_k^\gamma \rightarrow I_i^\alpha$ hold. □

In the definition, $\text{acq}(L)$ and $\text{rel}(L)$ indicate the acquire and the release operations on a lock $L$, respectively; and $\text{barrier}(L')$ indicates the barrier operation on a lock $L'$. Now, we define the orphan interval and the consistent recovery more formally. For a process $p_f$ recovering from a failure, let $I_f^{BF}$ be the sequence of state intervals between the latest checkpointing of $p_f$ and the failure of $p_f$; and let $I_f^{AF}$ be the sequence of state intervals generated by $p_f$ after the failure. Let the invalidated state intervals, $I_f^{INV}$, be the sequence of states which are in $I_f^{BF}$ but not in $I_f^{AF}$.

**Definition 5:** A state interval $I_i^\alpha$ is an orphan interval, if for any state interval $I_f^\beta \in I_f^{INV}, I_f^\beta \rightarrow I_i^\alpha$. □

**Definition 6:** The recovery from a failure of a process $p_f$ is consistent if no orphan interval remains in the system after the rollback. □

### 2.3 Stable Logging

To achieve the consistent recovery, either the orphan intervals have to be traced and discarded; or the states in $I_f^{BF}$ must be regenerated so that no orphan interval can remain in the system. For the state regeneration, the causal logging scheme [2] can be used. Let $\text{Log}(e_i^\alpha)$ be the information logged to regenerate the exactly same event $e_i^\alpha$. Under the assumption of the piece-wise deterministic computation, only the read and the write events are the non-deterministic events. Hence, if every $\text{Log}(e_i^\alpha)$ for each read and write event, $e_i^\alpha$, can be retrieved during the recovery, the same states in $I_f^{BF}$ can be regenerated and the consistent recovery can be guaranteed.

For the correct regeneration of an event $e_i^\alpha$, $\text{Log}(e_i^\alpha)$ must include the data page which has been
provided for \( e^a_i \); or to properly retrieve that data page, the identifier of the write event which creates that data page and the identifier of the event \( e^a_i \) must be retained. In the causal logging, the data page and its write event identifier are logged at the volatile storage of the writer process, and the pair of the write event identifier and the event identifier for \( e^a_i \) is causally logged at the volatile storage of the potentially dependent processes. Hence, when a process fails, it can retrieve the information, regarding which data page should be retrieved for each event, from the potentially dependent processes. If all of the dependent processes also fail, the events need not be regenerated, since there remains no dependents. If the \( \text{Log}(e^a_i) \) can be retrieved even in case of the concurrent system failures, the log is said to be stable; and if for every event \( e^a_i \) which may cause some orphan intervals, \( \text{Log}(e^a_i) \) is stable, the consistent recovery can be guaranteed.

**Definition 7:** A log, \( \text{Log}(e^a_i) \), is stable if \( \text{Log}(e^a_i) \) can be retrieved even in case of the concurrent system failures. □

**Lemma 1:** If for every \( e^a_i \) satisfying the following conditions, \( \text{Log}(e^a_i) \) is stable, the recovery for a process \( p_i \) is consistent.

C1. \( s^a_i \in I_i^{BF} \).

C2. For any state \( s^\beta_j \), where \( j \neq i \), \( s^a_i \rightarrow_p s^\beta_j \) holds.

**Proof:** If for every relation, \( s^a_i \rightarrow_p s^\beta_j \), where \( s^a_i \in S_i^{BF} \), \( \text{Log}(e^a_i) \) can be retrieved, there cannot be any orphan interval (by Definition 5), and the rollback-recovery of \( p_i \) can be consistent (by Definition 6). □

3 Protocol Description

3.1 Overview

Under the piece-wise deterministic computation, a process can regenerate the same sequence of computational states, if the data values used for the shared memory accesses are logged and replayed in the same order[18]. Hence, for the consistent recovery, a process must have two types of information; one is the content of the data page it has accessed and the other is the identifier of the computational point at which the page has been accessed. In the causal logging approach, the data page content is logged at the volatile memory space of the process which produced it (a writer process), since the writer process can always regenerate the same content for the page, assuming that the writer process can perform the
correct recovery if it fails. Therefore, a process can always retrieve the same content of the data page for a specific computational point, if it maintains the version identifier of the corresponding page and the identifier of the access point.

To avoid the stable logging of such access information, the causal logging utilizes the following property: If a process has lost some state intervals due to a failure, the processes dependent on the lost intervals have to roll back together, to reach a consistent recovery line. However, if a process has lost some state intervals but there is no process dependent on those intervals, then arbitrary recomputation of the interval does not cause any inconsistency problem with other processes; that is, there is no need for the logging of those intervals[16]. Hence, whenever a process makes a new dependent process, it has to transfer its log of the access point and the corresponding page identifiers to the new dependent, and any log entries inherited from another process also have to be transferred. As a result, the log of a process’s access information is causally spread to the dependent processes, so that the process can recover to a consistent recovery line as long as any dependent survives the failure.

3.2 Data Structures

Some of the data structures and operations supported by the LRC memory model are described and it is discussed how they can be utilized for the causal logging.

3.2.1 diff Structure for Writer-based Data Page Logging

In the LRC memory 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 on the page. 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 created. Only the diff with its version identifier is sent to the requesting process. The requesting process collects the diff from every process which has written on that page, and applys the diffs into the page in the time order so that the most recent version of the data page can be created. Such a diff is maintained at the volatile storage of its writer process, with the version identifier for the write operation, until the system periodically discards the diffs which are no longer required by other processes, which operation is called a garbage collection.

In the proposed scheme, the diff structure maintained in the volatile storage of the writer process is utilized as the volatile log of data pages and only the diffs discarded by the garbage collection are saved.
into the stable storage as a part of the checkpoints. Hence, any \textit{diff} required for the recovery can be retrieved from the volatile log or from the checkpoint of the writer process. The version identifier saved with each \textit{diff} is considered as the identifier of the write event. The proposed scheme guarantees that the same version identifier is to be regenerated under the correct recovery.

3.2.2 Vector Clock for Potential Dependency Tracking

In the LRC memory model, processes are synchronized using \textit{acquire}, \textit{release}, and \textit{barrier} operations. And, any \textit{diff} \textit{x} written by a process \textit{p}\textsubscript{i} can be viewed by another process \textit{p}\textsubscript{j} only after a sequence of synchronization operations is performed. That is either after a sequence of the operations, \textit{write of x by p}\textsubscript{i}, \textit{release of a lock L by p}\textsubscript{i}, and \textit{acquire of the same lock L by p}\textsubscript{j}, or after \textit{p}\textsubscript{k} and \textit{p}\textsubscript{j} performs the same \textit{barrier} operations. To identify the legally accessible \textit{diffs}, the logical vector clock, similar to the one suggested in [20], is used. A vector clock \textit{T}\textsubscript{i} of process \textit{p}\textsubscript{i} is an array of \textit{N} integers, where \textit{N} is the number of processes in the system. \textit{T}\textsubscript{i}[\textit{k}] is incremented by one when \textit{p}\textsubscript{i} releases a lock or reaches a barrier, following any write operation. Each \textit{T}\textsubscript{i}[\textit{k}] is updated as the maximum of \textit{T}\textsubscript{i}[\textit{k}] and \textit{T}\textsubscript{j}[\textit{k}] when \textit{p}\textsubscript{i} acquires a lock from another process \textit{p}\textsubscript{j} or leaves the same barrier with \textit{p}\textsubscript{j}.

The vector clock is associated with each synchronization operation and is also associated with each \textit{diff}. According to the memory access rule of the LRC memory model, a process can legally access only the \textit{diffs} with the vector clock less than or equal to its current vector clock. Note that a vector clock \textit{T}\textsubscript{i} is less than or equal to \textit{T}\textsubscript{j} if and only if for every \textit{k}, \textit{T}\textsubscript{i}[\textit{k}] ≤ \textit{T}\textsubscript{j}[\textit{k}]. For example, in Figure 3, the vector clock \textit{T}\textsubscript{i:2} = (1, 0, 0) of \textit{p}\textsubscript{k} becomes the version identifier for the \textit{diffs} created by \textit{W(X)} and \textit{W(Y)}.

From the definition of the vector clock and the data access rule in the LRC memory model, it can be noticed that the vector clock also reflects the potential dependency relation between the read and write operations defined earlier. If a process accesses a data page when its vector clock is \textit{T}, the page must include all and only the updates (the \textit{diffs}) created before the vector clock \textit{T}. Hence, if the vector clock for each read or write operation is carefully logged, it can be used as the event identifier. Since a \textit{state interval} is defined as a sequence of the states between any two consecutive synchronization operations, for any
read or write operation within a state interval, the same vector clock can be applied. This means that the vector clock of each synchronization operation is enough to be logged for the potential dependency tracking.

### 3.2.3 Write Notices for Causal Logging

The write notice is a data structure which is used to invalidate the out-of-dated data pages. Under the multiple-reader, multiple-writer protocol, when a process writes on a data page, the old copies of the page must be invalidated. However, in the LRC memory model, the invalidation can be delayed until a process acquires a proper lock. When a process acquires a lock, the last releaser of the lock sends the write notice records, each of which includes the identifier of a data page which has to be invalidated, with the identifier of its writer process and the vector clock of the write operation. Since the write notices are also causally propagated as the synchronization operations are performed, the causal logging operations of the vector clock can be incorporated into the write notice handling operations.

For example, in Figure 3, when \( p_j \) acquires back a lock from \( p_i \), it receives the write notice records for the write operations, \( W(X) \) and \( W(Y) \) of \( p_i \). When \( p_k \) acquires a lock from \( p_j \), it receives the write notice records for \( W(Y) \) of \( p_j \) as well as the records for \( W(X) \) and \( W(Y) \) of \( p_i \). Hence, each process must maintain the write notices for its own write operation and also the ones received from other processes. Moreover, to reduce the message overhead incurred by the write notices, when a process sends the lock acquire request, it attaches its current vector clock into the request message, so that the releaser can send only the write notices which have not been known by the new acquirer. As a result, the write notices are not redundantly sent to the processes. For example, in Figure 3, if \( p_l \) acquires the lock...
A released by \( p_k \), the write notices do not include the records for \( W(X) \) and \( W(Y) \) performed by \( p_i \), since the current vector clock of \( p_i \), \( T_{i;2} = (1,0,0) \), indicates that the vector clock, \( (1,0,0) \), associated with the writes, \( W(X) \) and \( W(Y) \), has been known to \( p_i \).

### 3.3 Dependency Tracking

Causal logging consists of two parts; one is the writer-based logging of the data contents, and the other is the causal logging of the data access information by the potentially dependent processes. For the logging of the data, the diff structure maintained by each process is utilized, since the diff can be regenerated from a correct recovery even after a system failure. As for the event identifiers to trace the data access, the vector clock associated with each synchronization operation is used. To causally log the vector clocks, the operations handling the write notice structures are used. One notable point of the write notice handling operations is that the write notice records are causally propagated to the processes in the potential dependency relation. Another notable point is that the propagation is not redundant. If a write notice record is transferred to a process, the same record is not delivered to the same process, again.

**Lemma 2:** Let \( w.n_i^a \) be the write notice record for a write event \( e_i^a \), and \( W.N_j \) be the set of write notice records maintained by \( p_j \). If the relation, \( e_i^a \rightarrow^p e_j^a \), holds, \( w.n_i^a \) must be included in \( W.N_j \).

**Proof:** In the LRC memory model, the relation \( e_i^a \rightarrow^p e_j^a \) can happen either when a lock release operation follows the event \( e_i^a \) and the acquire operation on the same lock precedes the event \( e_j^a \); or when a barrier operation follows the event \( e_i^a \) and the same barrier operation precedes the event \( e_j^a \). In both cases, the write notice record for \( e_i^a \) is propagated to \( p_j \) with the lock transfer and hence, \( W.N_j \) must include \( w.n_i^a \). \( \Box \)

The write notices, however, reflect only the information regarding the write operations. In other words, only the vector clock for the release operation following any write operation can be included in the write notice structure. In order to include the vector clock for each synchronization operation into the write notice structure, the structure needs to be slightly modified. However, considering the size of the vector clock, it is more efficient to save some information to recreate the vector clock, instead of saving the vector clock itself. Hence, in the proposed scheme, for each synchronization operation, the sufficient and necessary information to recreate the corresponding vector clock is inserted into the write notice structure.
In the LRC memory model, the state intervals between two synchronization operations of a process can partially be ordered according to the dependency relation stated in Definition 4; and for the intervals in the relation, $I_j^\beta \rightarrow I_i^\alpha$, the interval, $I_j^\beta$, is said to precede another interval, $I_i^\alpha$. The vector clock associated with each interval also reflects the precedence order in the dependency relation. For example, in Figure 4.(a), the vector clock for Acq(a) of $p_k$ is calculated as the entry-wise maximum of the vector clocks of its preceding intervals. Hence, the $j$-th entry of a vector clock for an interval of $p_i$, say $T_{i}[j]$, indicates the $p_j$’s latest interval preceding $p_i$’s corresponding interval. In Figure 4, the circled release operations denote the ones following some write operations, and the others denote the release operations without any write. Since the $i$-th entry of a vector clock $T_{i}$ is updated only at the release time after any write, the acquire and the release operations without any circle need not be considered for the vector clock calculation.

Figure 4.(b) represents the precedence order in the dependency relation as a graph, in which the release operations without any write are omitted since they do not affect the vector clock. Let $N_{i}^{\alpha}$ denote a node representing the $\alpha$-th synchronization operation of $p_i$. The arrow from $N_{i}^{\alpha}$ to $N_{j}^{\beta}$ in the figure
denotes that $N_i^\beta$ precedes $N_i^\alpha$. In other words, $I_j^\beta \rightarrow I_j^\alpha$. Now, the vector clock for any acquire operation can be obtained as the entry-wise maximum of the vector clocks of the release operations reachable by that acquire in the graph. Let $P.Set_i^\alpha$ be the set of release operations preceding an acquire operation, $N_i^\alpha$, and $P.Set_i^{*\alpha}$ be the set of release operations which are reachable from $N_i^\alpha$ in the precedence graph and each of which has no other release operation existing in the path from $N_i^\alpha$ to itself. Then, the vector clock obtained from $P.Set_i^{*\alpha}$ is equal to the one obtained from $P.Set_i^\alpha$.

**Lemma 3:** The entry-wise maximum of the vector clocks associated with the release operations in $P.Set_i^\alpha$ is equal to the one of the vector clocks associated with the release operations in $P.Set_i^{*\alpha}$.

**Proof:** Each release operation in $(P.Set_i^\alpha - P.Set_i^{*\alpha})$, say $N_j^\beta$, must have a path from any release operation, say $N_k^\gamma$, in $P.Set_i^{*\alpha}$. Then, the vector clock of $N_j^\beta$ must be less than or equal to the one of $N_k^\gamma$. As a result, the vector clock obtained from $P.Set_i^{*\alpha}$ must be equal to the one obtained from $P.Set_i^\alpha$. □

### 3.4 Causal Logging

Now, we have to slightly modify the write notice structure so that the precedence graph shown in Figure 4 can be reflected in the write notice structure. Figure 5.(a) represents the write notice structure supported by the existing LRC memory model, which consists of an array of $N$ pointers, where $N$ is the number of processes in the system. Each pointer indicates a list of interval records created by the corresponding process, and each interval record contains the identifier of the process and the vector clock of the interval and a list of write notices for the write operations which have happened during the interval. An interval is created only at the release time after some write operations have happened. As shown in the figure, the structure naturally supports the directed edges within a process in the precedence graph, since the intervals are arranged in the descending order of the vector clock.

To represent the edges between the distinct processes, we use a pair of integers, $(i, \alpha)$, which indicates the $\alpha$-th synchronization operation of $p_i$. The inter-process edges are two types; one from an acquire operation to a release operation; and the other is from an acquire operation to another acquire operation. Each release operation of $p_i$ followed by any write can uniquely be identified by the $i$-th entry of $T_i$ ($T_i[i]$), however, each acquire operation may not be distinguishable. Hence, in the proposed scheme, the value of $T_i[i]$ is also incremented by one for each acquire operation. The revised clocks do not violate the semantics of the vector clock[6]. The causal logging can be completed by inserting an interval entry
for each acquire operation, which contains the pair, \((i, \alpha)\), indicating that the lock has been acquired from \(p_i\) and the value of \(T_i[i]\) was \(\alpha\) at the lock release time of \(p_i\).

Figure 5.(b) represents the revised write notice structure supporting the causal logging. The vector clock for each acquire operation can be obtained from the structure as we mentioned before. Note here that the obtained vector clock may not be the same as the actual clock since the value of \(T_i[i]\) is also incremented for each acquire operation. Hence, the number of acquire operations in each path has to be counted and reflected to the calculated vector clock. Now, considering a barrier operation, inter-process dependency can happen between every pair of processes passing the same barrier, which may make the write notice structure very complicated. To reduce the overhead of recording the dependency relation between every pair of processes, we modify the write notice structure for the barrier operation as shown in Figure 6. The order of the processes in the cycle can be determined by the barrier manager, and it can be noticed that in both structures shown in Figure 6, the same set of release operations can be used for the vector clock calculation.

**Lemma 4:** The proposed logging scheme ensures that if for any \(e_i^\alpha\), an event \(e_j^\beta\) in the relation \(e_i^\alpha \rightarrow_p e_j^\beta\) exists, then \(Log(e_i^\alpha)\) is stable.
Proof: For the $Log(e_i^a)$ to be stable, the data contents which has been used for $e_i^a$ with its write event identifier and the dependency relation of the write event to the event $e_i^a$ must be saved. In the proposed scheme, $p_j$ must save the interval record for the write event and the interval record including $e_i^a$ with the relation between the events (by Lemma2). Also, $p_j$ can recalculate the same vector clocks for the write event and the event $e_i^a$ (by Lemma 3). As for the data contents, the same $diff$ can be regenerated since the same vector clocks can be regenerated. As a result, $Log(e_i^a)$ must be stable. $\Box$

3.5 Checkpointing

Each process in the system periodically takes a checkpoint to reduce the amount of recomputation in case of a system failure. A checkpoint includes the intermediate state (context) of the process, the current vector clock, the current contents of the $diff$ structure and the other information required for the memory consistency protocol. The checkpointing activities among the related processes need not to be performed in a coordinated way. However, if the checkpointing activity is incorporated into the barrier operation or the garbage collection in the system, the overhead of the checkpointing can be reduced.
3.6 Rollback-Recovery

For a process $p_i$ to recover from a failure, a recovery process, say $p'_i$, is first created and $p'_i$ broadcasts the recovery message to all the other processes in the system. On the receipt of the message, each process, $p_j$, first determines whether it has write notice records for $p_i$ or not. If so, it replies with its write notice structure, which includes $p_i$’s intervals and their preceding intervals. Otherwise, $p_j$ just replies with a negative acknowledgement. When $p'_i$ collects the reply messages from all the processes in the system, it eliminates the duplicates of the interval records and reconstructs its own write notice structure. The recovery process $p'_i$ then restores the computational state saved as the last checkpoint of $p_i$ and from the reconstructed state, the process $p_i$ begins the recomputation as follows:

At the lock acquire time: $p_i$ calculates the vector clock from the information saved in the write notice structure and sets its current vector clock as the calculated one. $p_i$ also searches the write notice structure with the new vector clock, in order to select the write notices for which the invalidation has to be performed.

At the data page access miss: $p_i$ searches the write notice structure and sends the diff requests to the selected writer processes of the data as before. Each of the writer processes then replies with the diffs of the data page it has produced before the given vector clock. The recovering process arranges the received diffs in the timing order and applies them on the data page to create an up-to-date version, as performed during the normal execution. If the write operation is to be performed, $p_i$ creates a diff with its current vector clock.

At the lock release time or at the barrier: If $p_i$ has performed any new write operation before the synchronization, it increments the value of $T_i[i]$ by one.

Since for each synchronization operation, the process can retrieve the vector clock same as the one which has been used before the failure, it can retrieve the same diffs from the other processes and create the same data page for the read and write operations.

**Theorem:** The rollback-recovery of any process under the proposed scheme is consistent.

**Proof:** If for any state interval $I_i^\alpha \in I_i^{BF}$, an interval $I_j^\beta$ in the relation $I_i^\alpha \rightarrow I_j^\beta$ exists, the proposed logging scheme ensures that for every $e_i^\gamma \in I_i^\alpha$, $Log(e_i^\gamma)$ is stable (by Lemma 4). As a result, the rollback-recovery of $p_i$ is safe (by Lemma 1). □
For the performance evaluation, the proposed causal logging scheme has been implemented on top of the CVM [11] distributed shared memory system. The CVM system is written in C++ and well modularized. The basic high level classes of CVM are CommManager and Msg classes to handle the network operation, MemoryManager class for the memory management, and Page and DiffDesc classes for the page management. The protocol classes such as LMW, LSW and SEQ inherit the high level classes and support operations to achieve the proper memory consistency. LMW and LSW support the lazy release memory consistency with multiple writers and with a single writer, respectively; and SEQ supports the sequential memory consistency model.

For our experiments, the LMW protocol was selected. The classes related to the LMW protocol are LmwInterval, LmwNotice, DiffHeap, DiffManager, IntervalManager, LmwProtocol and LmwPage. Among these, LmwInterval, IntervalManager and LmwProtocol classes were modified for the causal logging of remote accesses. For the volatile logging, some memory spaces are initially allocated to each process.

For the experiments, the IBM RS/6000 SP2 running AIX 4.1.2 was used. The machine is equipped with forty 66MHz POWER2 nodes carrying 128KB D-cache and 32KB I-cache. Eight of these nodes are wide nodes with 512MB memory; and the rest of them are thin nodes with 128MB of main memory. The thin nodes have been used for our experiments. The machine supports 5.8GB of total memory and 134GB of disk space. All the nodes are connected through the IBM SP2 high-performance switch(HPS) which is a two-level cross-bar switch and provides a point-to-point bandwidth of 40MB/sec [21].

To measure the performance of causal logging, two versions of the logging scheme have been implemented; one using the vector clock for dependency tracking and the other using the optimized dependency tracking. Some parallel application programs from the SPLASH suite [22] have run with each of the causal logging schemes and also without any logging. The applications used for the experiments are as follows:

- **Barnes-Hut (BARNs)** calculates gravitational forces for a system of particles. The algorithm is based on a hierarchical oct-tree whose internal nodes are space cells and leaves are individual particles; and 4096 particles were used for the experiment. The master processor first calculates the center of mass for each cell and then the processors calculate forces on the particles. The calculation is performed in five iterations and barriers are used to synchronize each iteration.
- **Traveling Salesman Problem (TSP)** finds the shortest path that visits every node exactly once and returns to a starting node. The algorithm is based on a branch-and-bound approach and 40 nodes were used for the experiment. The processors evaluate the promising tours from the priority queue and compares with the global minimum tour which has been selected so far. To access the priority queue and the global minimum tour, locks are used.

- **3-D FFT (FFT)** solves partial differential equations numerically. The algorithm is based on three dimensional forward and inverse FFT on a three dimensional array; and the 128X64X64 array of double precision complex numbers was used for the experiment. The calculation is performed in a number of iterations and for each iteration, barriers are used between the update and the transpose of the array.

- **Successive Over-Relaxation (SOR)** solves partial differential equations by iterating over a two dimensional array. The algorithm is based on the red-black approach and the 512X2048 matrix was used for the experiment. For each of one hundred iterations, the red elements are first updated by the processors and then the black elements are updated. Barriers are used to synchronize each update.

The detailed description of the application programs can be found in [3, 13]. The execution time of each program run and the amount of logged data for each application have been measured. Also, to examine the effects of the system size on the performance, each application has been run with the different number of nodes.

Figures 7, 8, and 9 show the execution time of the parallel programs, FFT, SOR and TSP, respectively; and the execution time is measured for the run with the optimized causal logging (denoted by *Logging*) or for the run without any logging (denoted by *No logging*). The number within the parentheses indicates the number of nodes executing the application program. The performance graphs show that in all cases, the execution time under the optimized causal logging scheme is only slightly longer than the one under no logging environment. Approximately, 0.4% to 4.4% increases in the execution time are enforced by the proposed causal logging scheme. Analyzing the overhead of the proposed scheme, each process should spend some CPU time to make an interval entry for each acquire and barrier operation, and to insert the interval entry into the acquire list field of the write notice after a write operation is performed. Since such CPU time spent for the synchronization operation is usually considered to be very small, the overhead caused by those operations are not that high.

Another type of overhead is related to the message processing. By adding the extra information to the write notices, the size of the lock grant message to the new lock acquirer must be increased and hence,
Figure 7: The Execution Time of FFT Application

Figure 8: The Execution Time of SOR Application
the longer processing time is required to pack and unpack each message. Such processing time must be closely related to the amount of data added to each message and also closely related to the number of lock requests and number of barrier operations of each application. Since in the proposed logging scheme, an interval entry contains two integer values and the size is not dependent on the number of processes participating in the parallel application, the increases in the message size is only dependent on the number of acquire and barrier operations in the application. However, in the LRC scheme, when the write notices are sent to a process, only the entries which have not been seen by the new acquire are transferred by comparing the time of the last releaser and the new acquire. Hence, the size of the additional interval entry must be proportional to the number of acquire operations which have happened after a lock release and before acquiring back the same lock. Table 1 shows the increases in the execution time enforced by the causal logging scheme, which are not dependent on the number of processes.

Figure 10, 11 and 12 compares the performance of the optimized causal logging scheme (denote by CL-Scheme1) with the one of causal logging using the vector clock itself for the dependency tracking (denoted by CL-Scheme2). Figure 10 first shows the amount of the data added to the write notice structure managed by each process for the purpose of logging. The performance graph shows that the optimized dependency tracking achieves 35%-69% of reduction in the amount of log, and the reduction becomes increased as the number of processes increases. Figure 11 shows the amount of the data added
<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>0.49%</td>
<td>1.02%</td>
<td>1.21%</td>
</tr>
<tr>
<td>SOR</td>
<td>1.89%</td>
<td>4.42%</td>
<td>3.64%</td>
</tr>
<tr>
<td>TSP</td>
<td>0.45%</td>
<td>0.39%</td>
<td>0.62%</td>
</tr>
</tbody>
</table>

Table 1: The Overhead of Causal Logging in Execution Time

Figure 10: The Amount of Log Managed for a Process

to the write notices carried in each message and the optimized logging scheme also shows 37.5% - 66% of reduction in the data size. From both figures, it can be concluded that the proposed causal logging scheme enforces much less overhead in the size of the log, and the reduction is achieved by employing the optimized dependency tracking. Also, the size of the log shows slow increase as the number of processes increases, which indicates that the proposed scheme can be a good logging choice for the systems with a large number of processes.

Figure 12 compares the execution time of two logging schemes. Comparing CL-Scheme1 with CL-Scheme2, there is a slight reduction in the execution time. However, considering the facts that processing of the message and the log can be the major overhead as the system size increases and also
Figure 11: The Amount of Log Carried in a Message

Figure 12: The Execution Time
that the performance of CL-Scheme2 is very sensitive to the number of processes, it is expected that the reduction can be more as the number of processes increases.

5 Conclusions

In this paper, we have proposed a causal logging scheme for the LRC based DSM system, which tolerates multiple failures with the low overhead. In the proposed scheme, the writer process performs the volatile logging of data contents, using the diff structure supported by the memory model. To trace the inter-process dependency, the vector clock supported by the system is also utilized and the causal logging of the vector clocks is associated with the maintenance operation of the write notices. As a result, causal logging can be achieved utilizing the data structures and operations inherited from the LRC memory model. Also, to reduce the overhead of the vector clock, causal logging in the proposed scheme is implemented by inserting the sufficient information to recreate the vector clock for each interval instead of using the vector clock itself. To evaluate the performance of the proposed scheme, the logging protocol has been implemented on top of CVM distributed shared memory system. The experimental results show that the logging operation requires only 1% - 4.4% of the execution time increases.

References


