EasyChair Preprint

№ 3517

A Low Overhead Logging Scheme for Fast Recovery in Distributed Shared Memory Systems

Mushtaq Ahmad, Saima Amir, Mehwish Shabbir and Muhammad Nadeem

May 30, 2020

A Low Overhead Logging Scheme for Fast Recovery in Distributed Shared Memory Systems

Authors:

Mushtaq Ahmad
Riphah International University
Webeng.mushtaq@gmail.com

Saima Amir
University of Lahore
saima.amir@pctb.punjab.gov.pk

Mehwish Shabbir
Riphah International University
mehwish.shabbir22@gmail.com

Muhammad Nadeem
Riphah International University
nadeem.ameen@yahoo.com

Abstract. This paper presents an efficient, writer-based logging scheme for recoverable distributed shared memory systems, in which logging of a data item is performed by its writer process, instead of every process that accesses the item logging it. Since the writer process maintains the log of data items, volatile storage can be used for logging. Only the readers' access information needs to be logged into the stable storage of the writer process to tolerate multiple failures. Moreover, to reduce the frequency of stable logging, only the data items accessed by multiple processes are logged with their access information when the items are invalidated, and also semantic-based optimization in logging is considered. Compared with the earlier schemes in which stable logging was performed whenever a new data item was accessed or written by a process, the size of the log and the logging frequency can be significantly reduced in the proposed scheme.

Keywords: checkpointing, distributed shared memory system, fault tolerant system, message logging, rollback-recovery and log size

1. Introduction

Distributed shared memory DSM systems 15 transform an existing network ofŽ . w x workstations into a powerful shared-memory parallel computer which could deliver superior pricerperformance ratio. However, with more workstations engaged in the system and longer execution time, the probability of failures increases, which could render the system useless. For the DSM system 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 when there is a failure 25 . An approach to providingw x fault-tolerance to the DSM systems is to use checkpointing and rollback-recovery. Checkpointing is an operation to save intermediate system states into stable storage which is not affected by system failures. With periodic checkpointing, the system can recover to one of the saved states, called a *checkpoint*, when a failure occurs in the system. The activity to resume the computation from one of the previous checkpoints is called *rollback*.

An earlier version of this work has appeared in the proceedings of the 17th International Conference on Distributed Computing Systems, 1997.

In DSM systems, the computational state of a process becomes dependent on the state of another process by reading a data item produced by that process. Because of such dependency relations, a process recovering from a failure has to force its dependent processes to roll back together, if it cannot reproduce the same sequence of

data items. While the rollback is being propagated to the dependent processes, the processes may have to roll back recursively to reach a consistent recovery line, if the checkpoints for those processes are not taken carefully. Such recursive rollback is called the domino effect w17x, and in the worst case, the consistent recovery line consists of a set of the initial points; i.e., the total loss of the computation in spite of the checkpointing efforts.

One solution to cope with the domino effect is coordinated checkpointing, in which each time a process takes a checkpoint, it coordinates the related processes to take consistent checkpoints together 3, 4, 5, 8, 10, 13 . Since each checkpointingw x coordination under this approach produces a consistent recovery line, the processes cannot be involved in the domino effect. One possible drawback of this approach is that the processes need to be blocked from their normal computation during the checkpointing coordination. The communication-induced checkpointing is another form of coordinated checkpointing, in which a process takes a checkpoint whenever it notices a new dependency relation created from another process w9, 22, 24, 25 . This checkpointing coordination approach also ensures no dominox effect since there is a checkpoint for each communication point. However, the overhead caused by too frequent checkpointing may severely degrade the system performance.

Another solution to the domino effect problem is to use message logging in addition to independent checkpointing w19x. If every data item accessed by a process is logged in the stable storage, the process can regenerate the same computation after a rollback by reprocessing the logged data items. As a result, the failure of one process does not affect other processes, which means that there is no rollback propagation and also no domino effect. The only possible drawback of this approach is the nonnegligible logging overhead.

To reduce the logging overhead, the scheme proposed in 23 avoids repeatedw x logging of the same data item accessed repeatedly. For correct recomputation, each data item is logged once when it is first accessed, and the count of repeated access is logged for the item, when the data item is invalidated. As a result, the size of the log can be reduced compared to the scheme in 19 . The scheme proposed in 11w x w x suggests that a data item should be logged when it is produced by a write operation. Hence, a data item accessed by multiple processes need not be logged at multiple sites and the size of the log can be reduced. However, for a data item written but accessed by no other processes, the logging becomes useless. Moreover, for the correct recomputation, a process accessing a data item has to log the location where the item is logged and the access count of the item. As a result, there cannot be much reduction in the frequency of logging compared to the scheme in 23 .w x

To further reduce the logging overhead, the scheme proposed in w x7 suggests volatile logging. When a process produces a new data item by a write operation, the value is logged into the volatile storage of the writer process. When the written value is requested by other processes, the writer process logs the operation number of the requesting process. Hence, when the requesting process fails, the data value

and the proper operation number can be retrieved from the writer process. Compared with the overhead of logging into stable storage, volatile logging incurs much less overhead. However, when there are concurrent failures at the requesting process and the writer process, the system cannot be fully recovered.

In this paper, we present a new logging scheme for a recoverable DSM system, which tolerates multiple failures. In the proposed scheme, a two-level log structure is used in which both the volatile and the stable storages are utilized for efficient logging. To speed up the logging and the recovery procedures, a data item and its readers' access information are logged into the volatile storage of the writer process. And, to tolerate multiple failures, only the log of access information for the data items is saved into the stable storage. For volatile logging, the limited space can be one possible problem and for stable logging, the access frequency of the stable storage can be the critical issue. To solve these problems, logging of a data item is performed only when the data becomes invalidated by a new write operation, and the writer process takes the whole responsibility for logging, instead of every process accessing the data concurrently logging it. Also, to eliminate unnecessary logging of data items, semantic-based optimization is considered for logging. As a result, the size of the log and the frequency of stable storage accesses can substantially be reduced.

The rest of this paper is organized as follows: Section 2 presents the DSM system model and the definition of the consistent recovery line is presented in Section 3. In Section 4 and Section 5, the proposed logging and rollback recovery protocols are presented, respectively, and Section 6 proves the correctness of the proposed protocols. To evaluate the performance of the proposed scheme, we have implemented the proposed logging scheme on top of CVM Coherent Virtual Machinež . w12x. The experimental results are discussed in Section 7, and Section 8 concludes the paper.

2. The system model

A DSM system considered in this paper consists of a number of nodes connected through a communication network. Each node consists of a processor, a volatile main memory and a nonvolatile secondary storage. The processors in the system do not share any physical memory or global clock, and they communicate by message passing. However, the system provides a shared memory space and the unit of the shared data is a fixed-size page.

The system can logically be viewed as a set of processes running on the nodes and communicating by accessing a shared data page. Each of the processes can be considered as a sequence of state transitions from the initial state to the final state. An event is an atomic action that causes a state transition within a process, and a sequence of events is called a *computation*. In a DSM system, the computation of a process can be characterized as a sequence of readrwrite operations to access the shared data pages. The computation performed by each process is assumed to be

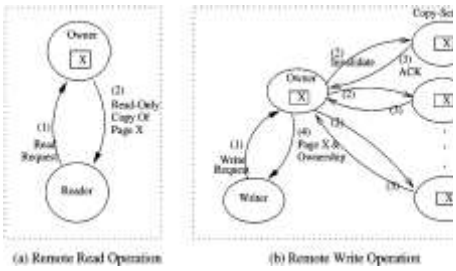


Figure 1. Remote readrwrite procedures

piece-wise deterministic; that is, the computational states generated by a process is fully determined by a sequence of data pages provided for a sequence of read operations.

For the DSM model, we assume the *read-replication* model 21 , in which thew x system maintains a single writable copy or multiple read-only copies for each data page. The memory consistency model we assume is the *sequential consistency* model, in which the version of a data page a process reads should be the latest version that was written for that data page 14 . A number of different memoryw x semantics for the DSM systems have been proposed including processor, weak, and release consistency 16 , as well as causal coherence 1 . However, in this paper, wew x w x focus on the sequential consistency model, and the *write-in¨alidation* protocol 15w x is assumed to implement the sequential consistency.

Figure 1 depicts the read and the write procedures under the write-invalidation protocol. For each data page, there is one owner process which has the writable copy in its local memory. When a process reads a data page which is not in the local memory, it has to ask for the transfer of a read-only copy from the owner. A set of processes having the read-only copies of a data page is called a *copy-set* of the page. For a process to perform a write operation on a data page, it has to be the owner of the page and the copy-set of the page must be empty. Hence, the writer process first sends the *write request* to the owner process, if it is not the owner. The owner process then sends the *in¨alidation message* to the processes in the copy-set to make them invalidate the read-only copies of the page. After collecting the invalidation acknowledgements from the processes in the copy-set, the owner transfers the data page with the ownership to the new writer process. If the writer process is the owner but the copy-set is not empty, then it performs the invalidation procedure before overwriting the page.

For each system component, we make the following failure assumptions: The processors are fail-stop 20 . When a processor fails, it simply stops and does notw x perform any malicious actions. The failures considered are transient and independent. When a node recovers from a failure and re-executes the computation, the same failure is not likely to occur again. Also, the failure of one node does not affect other nodes. We do not make any assumption on the number of simultaneous node failures. When a node falls, the register contents and the main memory contents are lost. However, the contents of the secondary storage are preserved and the secondary storage is used as a stable storage. The communication subsystem is reliable; that is, the message delivery can be handled in an error-free and virtually lossless manner by the underlying communication subsystem. However, no assumption is made on the message delivery order.

3. The consistent recovery line

A state of a process is naturally dependent on its previous states. In the DSM system, the dependency relation between the states of different processes can also be created by reading and writing the same data item. If a process p_i reads a data item written by another process p_j, then p_i's states after the read event become dependent on p_j's states before the write event. More formally, the dependency relation can be defined as follows: Let $R^a{}_i$ denote the a-th read event happening at process p_i and I_i^a denote the state interval triggered by $R^a{}_i$ and ended right before $R^a{}_i{}^{q1}$, where aG 0 and $R^0{}_i$ denotes the p_i's initial state. Let W_i^a denote the set of write events happening in I_i^a. Also, *R x*Ž s *u*. Žor *W x*Ž s *u*.. denotes the read orŽ the write event on a data item. *x* with the returning or written valueŽ . *u*.

Definition 1 An interval I_i^a is said to be *dependent on* another interval I_j^b if one of the following conditions is satisfied, and such a dependency relation is denoted by $I_{jb} \triangleq I_{ia}$:

C1. *i* s *j* and $^a s^b$q 1, or

C2. $R^a{}_i$ s *R x*Ž s *u*. and *W x*Ž s *u*. g W_j^b and there is no other *W x*Ž s u^x. in-between *W x*Ž s *u*. and *R x*Ž s *u*., or

C3. There exists an interval I_k^g, such that $I_j^b \triangleq I_k^g$ and $I_k^g \triangleq I_i^a$. I

Figure 2 shows an example of the computational dependency among the state intervals for a DSM system consisting of three processes p_i, p_j, and p_k. The horizontal arrow in Figure 2 a represents the progress of the computation at eachŽ . process and the arrow from one process to another represents the data page transfer between the processes. A data page *X* Žor *Y* . containing the data item *x* Žor *y*. is denoted by *X x*Ž . Žor *Y y*Ž ... Figure 2 b depicts the dependency relationŽ . created in Figure 2 a as a directed graph, in which each node represents a stateŽ . interval and an edge or a path from a nodeŽ . n_a to another node n_b indicates a direct Žor a transitive. dependency relation from a state interval n_a to another state interval n_b.

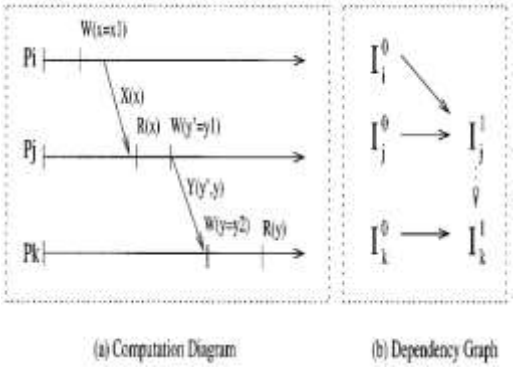
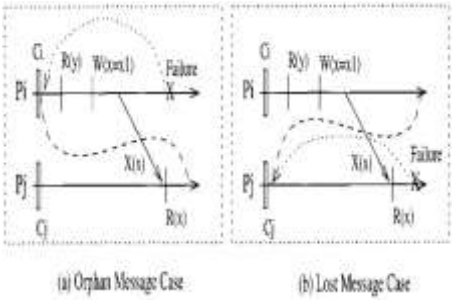


Figure 2. An example of dependency relations

Note that in Figure 2 a , there is no dependency relation fromŽ . I_j^1 to I_k^1 according to the definition given before. However, in the DSM system, it is not easy to recognize which part of a data page has been accessed by a process. Hence, the computation in Figure 2 a may not be differentiated from the one in whichŽ . p_k's read operation is *R y*Ž ^{x}.. In such a case, there must be a dependency relation, $I_j^1 \triangleq I_k^1$. The dotted arrow in Figure 2 bŽ . denotes such a possible dependency relation and the logging scheme must be

carefully designed to take care of such a possible dependency for consistent recovery.

The dependency relations between the state intervals may cause possible inconsistency problems when a process rolls back and performs the recomputation. Figure 3 shows two typical examples of inconsistent rollback recovery cases, discussed in message-passing based distributed computing systems 6 . First, sup-w x



pose the process p_i in Figure 3 a should roll back to its latest checkpointŽ . C_i due to a failure but it cannot retrieve the same data item for *R y*Ž .. Then, the result of *W x*Ž . may be different from the one computed before the failure and hence, the consistency between p_i and p_j becomes violated since p_j's computation after the event *R x*Ž . depends on the invalidated computation. Such a case is called an *orphan message* case.

Figure 3. Possible inconsistent recovery lines.

On the other hand, suppose the process p_j in Figure 3 b should roll back to itsŽ . latest checkpoint C_j due to a failure. For p_j to regenerate the exact same computation, it has to retrieve the same data item *x* from p_i. But, p_i does not roll back to resend the data page *X x*Ž .. Such a case is called a *lost message* case. However, in the DSM system, the lost message case itself does not cause any inconsistency problem. If there has been no other write operation since *W x*Ž . of p_i, then p_j can retrieve the same contents of the page from the current owner, at any time. Even though there has been another write operation and the contents of the page has been changed, p_j can still retrieve the data page *X x*Ž . Ževen with different contents. and the different recomputation of p_j does not affect other processes unless p_j had any dependent processes before the failure.

Hence, in the DSM system, the only rollback recovery case which causes an inconsistency problem is the orphan message case.

Definition 2 A process is said to recover to a *consistent reco¨ery line*, if and only if it is not involved in any orphan message case after the rollback recovery.

4. The logging protocol

For efficient logging, three principles are adopted. One is writer-based logging. Instead of multiple readers logging the same data page, one writer process takes the responsibility for logging of the page. Also, invalidation-triggered logging is used, in which logging of a data page is delayed until the page is invalidated. Finally, semantic-based logging optimization is considered. To avoid unnecessary logging activities, the access pattern of the data by related processes is considered in the logging strategy.

4.1. Writer-based, in¨alidation-triggered logging

For consistent regeneration of the computation, a process is required to log the sequence of data pages it has accessed. If the same contents of a data page have been accessed more than once, the process should log the page once and log its access duration, instead of logging the same page contents, repeatedly. The access duration is denoted by the first and the last computational points at which the page has been accessed. The logging of a data page can be performed either at the process which accessed it Žthe *reader*. or at the process which produced it Žthe *writer*.. Since a data page produced by a writer is usually accessed by multiple readers, it is more efficient for one writer to log the page rather than for multiple readers to log the same page. Moreover, the writer can utilize the volatile storage for the logging of the data page, since the logged pages are required for the reader's failure, not for the writer's own failure. Even if the writer loses the page log due to its own failure, it can regenerate the contents of the page, under the consistent recovery assumption.

To uniquely identify each version of data pages and its access duration, each process p_i in the system maintains the following data structures in its local memory:

- **pid**$_i$: A unique identifier assigned to process p_i.
- **opnum**$_i$: A variable that counts the number of read and write operations performed by process p_i. Using **opnum**$_i$, a unique sequence number is assigned to each of the read and write operations performed by p_i.

  For each version of the data page X produced by p_i, a unique version identifier is assigned.
- **version** $_x$: A unique identifier assigned to each version of data page X. **version** $_x$ s **pid :opnum**$_i$ $_i$, where **opnum**$_i$ is the **opnum** value at the time when p_i produced the current version of X.

When p_i produces a new version of X by a write operation, **version** $_x$ is assigned to the page. When the current version of X is invalidated, p_i logs the current version of X with its **version** $_x$ into p_i's volatile log space, and it also has to log the access duration for the current readers of page X. To report the access duration of a page, each reader p_j maintains the following data structure associated with page X, in its local memory.

- **duration** $_{jx}$: A record variable with four fields, which denote the access information of page X at p_j.

Ž .1 **pid: pid** $_j$.

Ž .2 **version: version** $_x$.

Ž .3 **first:** The value of **opnum**$_j$ at the time when page X is first accessed at p_j.

Ž .4 **last:** The value of **opnum**$_j$ at the time when page X is invalidated at p_j.

When the new version of page X is transferred from the current owner, p_j creates **duration** $_{jx}$ and fills out the entries **pid**, **version**, and **first**. The entry **last** is completed when p_j receives an invalidation message for X from the current owner, p_i. Process p_j then piggybacks the complete **duration** $_{jx}$

into its invalidation acknowledgement sent to p_i. The owner p_i, after collecting the **duration**$_{k\ x}$ from every reader p_k, logs the collected access information into its volatile log space. The owner p_i may also have **duration**$_{i\ x}$, if it has read the page X after writing on it. Another process which implicitly accesses the current version of X is the next owner. Since the next owner usually makes partial updates on the current version of the page, the current version has to be retrieved in the event of the next owner's failure. Hence, when a process p_k sends a write request to the current owner p_i, it should attach its **opnum**$_k$ value, and p_i, on receipt of the request, create **duration**$_{k\ x}$, in which **first** s **last** s **opnum**$_k$ q 1.

We here have to notice that the volatile logging of access information by the writer provides fast retrieval in case of a reader's failure. However, the information can totally be lost in the event of the writer's failure since unlike the data page contents, the access information cannot be reconstructed after the writer's failure. Hence, to cope with the concurrent failures which might occur at the writer and the readers, stable logging of the access information is required. When the writer p_i makes the volatile log of access information, it should also save the same information into its stable log space, so that the readers' access information can be reconstructed after the writer fails.

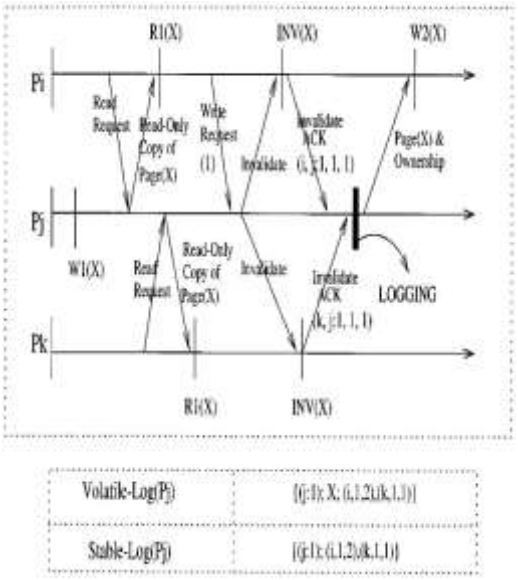
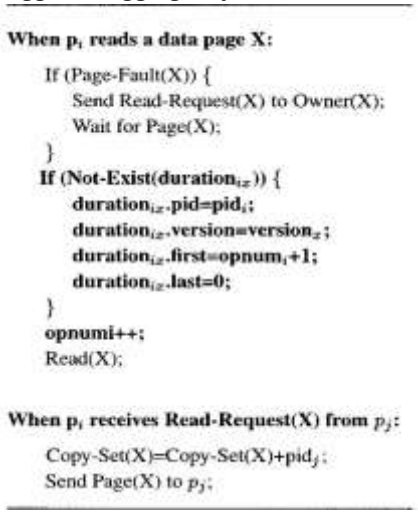


Figure 4. An example of writer-based, invalidation-triggered logging.

Figure 4 shows how the writer-based, invalidation-triggered logging protocol is executed incorporated with the sequential consistency protocol, for a system consisting of three processes p_i, p_j, and p_k. The symbol R_až *X*. žor W_až *X*.. in the figure denotes the read or the write operation to data pagež . *X* with the **opnum** value a, and *INV X*ž . denotes the invalidation of page *X*. In the figure, it is assumed that the data page *X* is initially owned by process p_j. As is evident from the figure, the proposed logging scheme requires a small amount of extra information piggybacked on the write request message and invalidation acknowledgements. Also, the volatile and the stable logging is performed only by the writer process and only at the invalidation time. Figure 4 also shows the contents of volatile and stable log storages at process p_j. Note that the stable log of p_j includes only the access information, while the volatile log includes the contents of page *X* in addition to the access information.

By delaying the page logging until the invalidation time, the readers' access information can be collected without

any extra communication. Moreover, the logging of access duration for multiple readers can be performed with one stable storage access. Though the amount of access information is small, frequent accesses to the stable storage may severely degrade the system performance. Hence, it is very important to reduce the logging frequency with the invalidationtriggered logging. However, the invalidation-triggered logging may cause some data



When p_i reads a data page X:

```
If (Page-Fault(X)) {
    Send Read-Request(X) to Owner(X);
    Wait for Page(X);
}
If (Not-Exist(duration_ix)) {
    duration_ix.pid=pid_i;
    duration_ix.version=version_x;
    duration_ix.first=opnum_i+1;
    duration_ix.last=0;
}
opnumi++;
Read(X);
```

When p_i receives Read-Request(X) from p_j:

```
Copy-Set(X)=Copy-Set(X)+pid_j;
Send Page(X) to p_j;
```

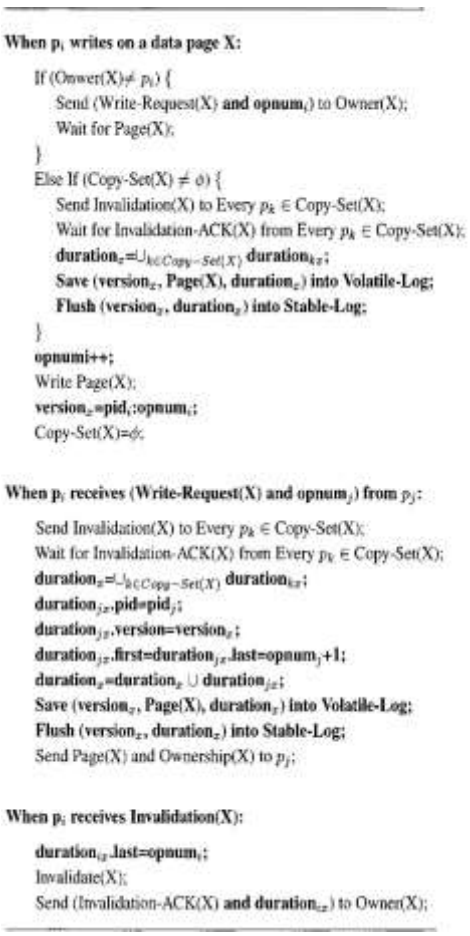
Figure 5. Writer-based, invalidation-triggered logging protocol.

pages accessed by readers but not yet invalidated to have no log entries. For those pages, a reader process cannot retrieve the log entries when it re-executes the computation due to a failure. Such a data page, however, can be safely refetched from the current owner even after the reader's failure, since a data page accessed by multiple readers cannot be invalidated unless every reader sends the invalidation acknowledgement back. That is, the data pages currently valid in the system do not necessarily have to be logged.

The sequential consistency protocol incorporated with the writer-based, invalidation-triggered logging is formally presented in Figure 5 and Figure 6, in which the bold faced codes are the ones added for the logging protocol.

4.2. Semantic-based optimization

Every invalidated data page and its access information, however, do not necessarily have to be logged, considering the semantics of the data page access. Some data pages accessed



When p_i writes on a data page X:

```
If (Owner(X)≠ p_i) {
    Send (Write-Request(X) and opnum_i) to Owner(X);
    Wait for Page(X);
}
Else If (Copy-Set(X) ≠ ϕ) {
    Send Invalidation(X) to Every p_k ∈ Copy-Set(X);
    Wait for Invalidation-ACK(X) from Every p_k ∈ Copy-Set(X);
    duration_x=∪_{k∈Copy-Set(X)} duration_kx;
    Save (version_x, Page(X), duration_x) into Volatile-Log;
    Flush (version_x, duration_x) into Stable-Log;
}
opnumi++;
Write Page(X);
version_x=pid_i:opnum_i;
Copy-Set(X)=ϕ;
```

When p_i receives (Write-Request(X) and opnum$_j$) from p_j:

```
Send Invalidation(X) to Every p_k ∈ Copy-Set(X);
Wait for Invalidation-ACK(X) from Every p_k ∈ Copy-Set(X);
duration_x=∪_{k∈Copy-Set(X)} duration_kx;
duration_jx.pid=pid_j;
duration_jx.version=version_x;
duration_jx.first=duration_jx.last=opnum_j+1;
duration_x=duration_x ∪ duration_jx;
Save (version_x, Page(X), duration_x) into Volatile-Log;
Flush (version_x, duration_x) into Stable-Log;
Send Page(X) and Ownership(X) to p_j;
```

When p_i receives Invalidation(X):

```
duration_ix.last=opnum_i;
Invalidate(X);
Send (Invalidation-ACK(X) and duration_ix) to Owner(X);
```

can be reproduced during the recovery and some access duration can implicitly be estimated from other logged access information. In the semantic-based logging strategy, some unnecessary logging points are detected based on the data page access pattern, and the logging at such points is avoided or delayed. This logging strategy can further reduce the frequency of the stable logging activity and also reduce the number of data pages logged in the volatile storage.

Figure 6. Writer-based, invalidation-triggered logging protocol continued .Ž .

First of all, the data pages with no remote access need not be logged. A data page with no remote access means that the page is read and invalidated locally, without creating any dependency relation. For example, in Figure 7, process p_i first fetches the data page X from p_j and creates a new version of X with an identifier Ži:1 . This version of the page is locally read for. R_2Ž X. and R_3Ž X., and invalidated for W_4Ž X.. However, when the version Ži:1. of X is invalidated due to the operation W_4Ž X., p_i need not log the contents of page X and the access duration Ži, i:1, 2, 4 . The reason is that during the recovery of. p_i, the version i:1 ofŽ . X can be regenerated by the operation W_1Ž X. and the access duration i, i:1, 2, 4 can beŽ . estimated as the duration between W_1Ž X. and W_4Ž X.. The next version Ži:4. of page X, however, needs to be logged when it is invalidated due to the operation W_2Ž X. of p_j, since the operation W_2Ž X. of p_j implicitly requires the remote access of the version i:4 .Ž .

By eliminating the logging of local data pages, the number of logged data pages in the volatile log space and also the access frequency to the stable log space can significantly be reduced. However, such elimination may cause some inconsistency problems as shown in Figure 8, if it is integrated with the invalidation-triggered logging. Suppose that process p_i in the figure should roll back after its failure. For consistent recovery, p_i has to perform the recomputation up to W_4Ž X.. Otherwise,

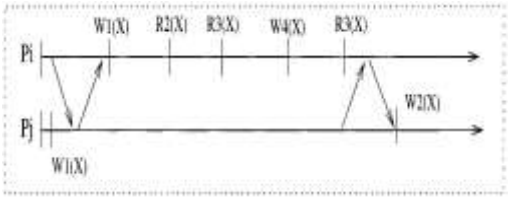


Figure 7. An example of local data accesses.

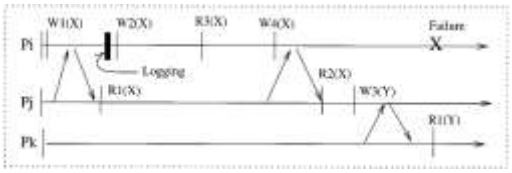


Figure 8. An example of operation counter vectors.

an orphan message case occurs between p_i and p_j. However, p_i performed its last logging operation before W_2Ž X. and there is no log entry up to W_4Ž X.. If p_i has no dependency on p_j, then it does not matter whether p_i rolls back to W_2Ž X. or to W_4Ž X.. However, due to the dependency on p_j, process p_i has to perform the recomputation at least up to the point at which the dependency has been formed.

To record the **opnum** value up to which a process has to recover, each process p_i in the system maintains an n-integer array, called an *operation counter ¨ector* ŽOCV ., where n is the

number of processes in the system ŽOCV$_i$ s ŽV$_i$w x1 , . . . , V i$_i$w x, . . . , V n$_i$w x... The *i*-th entry, V i$_i$w x, denotes the current **opnum** value of p_i, and V j$_i$w x Ži / j. denotes the last **opnum** value of p_j on which p_i's current computation is dependent. This notation is similar to the causal vector proposed in 18 . Hence,w x when a process p_j transfers a data page to another process p_i, it sends its current OCV_j value with the page. The receiver p_i then updates its OCV_i by taking the entry-wise maximum value of the received vector and its own vector, as follows:

$$OCV_i \text{ s } \check{Z}max\ V\check{Z}_i\text{w x1 , } V_j\text{w x1}$$
$$\text{.}, \ldots, max\ V\ n\check{Z}_i\text{w x, } V\ n_j\text{w x...}$$

For example, in Figure 8, when p_i sends the data page X and its version identifier i;4 toŽ . p_j, it sends its OCV_i s Ž4, 0, 0 with the page and then. p_j updates its OCV_j as 4, 2, 0 . WhenŽ . p_j sends the data page Y and its version identifier j:3Ž . to p_k, OCV_j s Ž4, 3, 0. is also sent with the page, and OCV_k is updated as OCV_k s Ž4, 3, 1 . As a result, each. V j$_i$w x in OCV_i indicates the last operation of process p_j on which process p_i's current computation is directly or transitively dependent. Hence, when p_j performs a rollback recovery, it has to complete the recomputation at least up to the point V j$_i$w x to yield consistent states between p_i and p_j.

Another data access pattern to be considered for the logging optimization is a sequence of write operations performed on a data page, as shown in Figure 9. Processes p_i, p_j, p_k, and p_l, in the figure, sequentially write on a data page X, although, the written data is read only by R_2Ž X. of p_l. This access pattern means that the only explicit dependency relation which occurred in the system is W_1Ž X. ª R_2Ž X. of p_l. Even though there is no explicit dependency between any of the write operations shown in the figure, the write precedence order between those

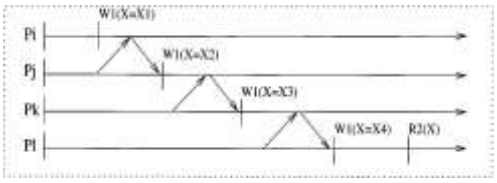


Figure 9. An example of write precedence order.

operations is very important, since the order indicates the possible dependency relation explained in Section 3 and it also indicates which process should become the current owner of the page after the recovery. To reduce the frequency of stable logging without violating the write precedence order, we suggest the delayed stable logging of some write precedence orders.

In delayed stable logging, the volatile logging of a data page and its access duration is performed as described before, but, the stable logging is not performed when a data page having no copy-set is invalidated. Instead, the information regarding the precedence order between the current owner of the page and its next owner is attached into the data page transferred to the next owner. Since the new owner maintains the unlogged precedence order information, the correct recomputation of its precedent can be performed as long as the new owner survives. Now, suppose that the new owner and its

precedent fail concurrently. If the new owner fails without making any new dependent after the write, arbitrary recomputation may not cause any inconsistency problem between the new owner and its precedent. However, if it fails making new dependents after the write, correct recovery may not be possible. Hence, a process maintaining the unlogged precedence order information should perform the stable logging before it creates any dependent process.

For example, in Figure 9, p_i does not perform stable logging when it invalidates page X. Instead, p_j maintains the precedence information, such as Ži:1.ªŽj:1 ,. and performs stable logging when it transfers page X to p_k. At this time, the precedence order between p_j and p_k, Žj:1.ªŽk:1 , can also be stably logged. together. Hence, the page X transferred from p_j to p_k need not carry the precedence relation between p_j and p_k. As a result, the computation shown in Figure 9 requires at most two stable logging activities, instead of four stable logging activities.

5. The recovery protocol

For consistent recovery, two log structures are used. The volatile log is mainly used for the recovery process to perform consistent recomputation, and the stable log is used to reconstruct the volatile log to tolerate multiple failures. In addition to the data logging, independent checkpointing is periodically performed by each process to reduce the recomputation time.

5.1. Checkpointing and garbage collection

To reduce the amount of recomputation in case of a failure, each process in the system periodically takes a checkpoint. A checkpoint of a process p_i includes the intermediate state of the process, the current value of **opnum**$_i$ and OCV_i, and the data pages which p_i currently maintains. When a process takes a new checkpoint, it can safely discard its previous checkpoint. The checkpointing activities among the related processes need not be performed in a coordinated manner.

A process, however, has to be careful in discarding the stable log contents saved before the new checkpoint, since any of those log entries may still be requested by other dependent processes. Hence, for each checkpoint, C_a, of a process p_i, p_i maintains a *logging ¨ector*, say **LV**$_{i,\,a}$. The j^{th} entry of the vector, denoted by **LV**$_{i,\,a}$**[ ]j** , indicates the largest **opnum**$_j$ value in **duration** $_{jx}$ logged before the corresponding checkpoint. When a process p_j takes a new checkpoint and the recomputation before that checkpoint is no longer required, it sends its current **opnum**$_j$ value to the other processes. Each process p_i periodically compares the received **opnum**$_j$ value with the **LV**$_{i,\,a}$**[ ]j** value of each checkpoint C_a. When for every p_j in the system, the received **opnum**$_j$ becomes larger than **LV**$_{i,\,a}$**[ ]j** , process p_i can safely discard the log information saved before the checkpoint, C_a.

5.2. *Rollback-reco¨ery*

The case of recovery of a single failure is discussed first. For a process p_i to be recovered from a failure, a recovery process for p_i, say p^x_i, is first created and it sets p_i's status as *reco¨ering*. Process p^x_i then broadcasts the *log collection* message to all the other processes in the system. On the receipt of the *log collection* message, each process p_j replies with the i-th entry of its OCV_j, $V\ i_j$w x. Also, for any data page X which is logged at p_j and accessed by p_i, the logged entry of **duration**$_{i\ x}$ and the contents of page X are attached to the reply message. When p^x_i collects the reply messages from all the processes in the system, it creates its *reco¨ery* _ *log* by arranging the received **duration**$_{i\ x}$ in the order of **duration**$_{i\ x}$**.first** and also arranging the received data pages in the corresponding order. Process p^x_i then selects the maximum value among the collected $V\ i_j$w x entries, where j s 1, . . . , n, and sets the value as p_i's *reco¨ery point*.

Since all the other processes in the system, except p_i, are in the normal computational status, p^x_i can collect the reply messages from all of them, and the selected recovery point of p_i indicates the last computational state of p_i on which any process in the system is dependent. Also, the constructed *reco¨ery* _ *log* for p_i includes every remote data page that p_i has accessed before the failure. The recovery process p^x_i then restores the computational state from the last checkpoint of p_i and from the restored state, process p_i begins the recomputation. The restored state includes the same set of active data pages which were residing in the main memory when the checkpoint was taken. The value of **opnum**$_i$ is also set as at the checkpointing time. During the recomputation, process p_i maintains a variable, called $Next_i$, which is the value of **duration**$_{i\ x}$**.first** for the first entry of the *reco¨erylog*, and $Next_i$ indicates the time to fetch the next data page from the *reco¨ery*_ *log*.

The read and write operations for p_i's recomputation are performed as follows: For each read or write operation, p_i first increments its **opnum**$_i$ value by one, and then compares **opnum**$_i$ with $Next_i$. If they match, the first entry of the *reco¨ery* _ *log* including the contents of the corresponding page and its **duration**$_{i\ x}$ is moved to the *acti¨e data page space*. Then, the operation is performed on the new page and any previous version of the page is removed from the active data page space. The new version of the page is used for the read and write operations until **opnum**$_i$ reaches the value of **duration**$_{i\ x}$**.last**. For some read and write operations, data pages created during the recomputation need to be used because of the logging optimization. Hence, if a new version of a data page X is created by a write operation and the corresponding log entry is not found in the *reco¨ery*_ *log*, the page must be kept in the active data

page space and the **duration**$_{i\,x}$**.last** is set as infinity. This version of page X can be used until the next write operation on X is performed or a new version of X is retrieved from the *reco¨ery_ log*.

Sometimes, when p_i reads a data page X, it may face the situation that a valid version of X is not found in the active data page space and it is not yet the time to fetch the next log entry Ž**opnum**$_i$ - *Next*$_i$.. This situation occurs for a data page which has been accessed by p_i before its failure, but, has not been invalidated. Note that such a page has not been logged since the current version is still valid. In this case, the current version of page X must be refetched from the current owner. Hence, when p_i reads a data page X, it has to request the page X from the current owner, if it does not have any version of X in the active data page space, or, the **duration**$_{i\,x}$**.last** value for page X in the active data page space is less than **opnum**$_i$. In both cases, **opnum**$_i$ must be less than *Next*$_i$. Any previous version of page X has to be invalidated after receiving a new version. The retrieval and the invalidation activities of data pages during the recomputation are summarized in Table 1. The active data page space is abbreviated to *ADPS* in the table.

During the recomputation, process p_i also has to reconstruct the volatile log contents which were maintained before the failure, for the recovery of other dependent processes. The access information of p_i's volatile log can be retrieved from its stable log contents while p^{X}_{i} is waiting for the reply messages after sending out the *log _ collection* requests. However, the data pages which were saved in the volatile log must be created during the recomputation. Hence, for each write operation, p_i logs the contents of the page with its version identifier if the corresponding access information entry is found in the volatile log. In any case, the write operation may cause the invalidation of the previous version of the page in the active data page space, however, it does not issue any invalidation messages to the other processes during the recomputation. When **opnum**$_i$ reaches the selected recovery point, p_i changes its status from *reco¨ering* to *normal* and resumes the normal computation.

Table 1. Retrieval of data pages during the recomputation

Condition	
opnum$_i$ - *Next*$_i$	X g *ADPS* **and** **opnum**$_i$ F **duration**$_{i\,x}$**.last**
	X g *ADPS* **and** **opnum**$_i$) **duration**$_{i\,x}$**.last**
	X f *ADPS*
opnum$_i$ G *Next*$_i$	

Now, we extend the protocol to handle concurrent recoveries from multiple failures. While a process p_i Žor p^{X}_{i}. performs the recovery procedure, another process p_j in the system can be

in the failed state or can also be in the *reco¨ering* status. If p_j is in the failed state, it cannot reply back to the *log_collection* message of p^x_i, and hence p^x_i has to wait until p_j wakes up. However, if p_j is in the *reco¨ering* status, it should not make p_i wait for its reply since, in such a case, both p_i and p_j must end up with a deadlocked situation. Hence, any message sent out during the *reco¨ering* status must carry the *reco¨ery* mark to be differentiated from the normal ones, and such a *reco¨ery* message must be taken care of without blocking, whether the message is for its own recovery or related to the recovery of another process. However, any normal message, such as a readrwrite request or an invalidation message, need not be delivered to a process in the *reco¨ering* status, since the processing of such a message during the recovery may violate the integrity of the system.

When p_i Žor p^x_i. in the *reco¨ering* status receives a *log_collection* message from another process p^x_j, it reconstructs the access information part of p_i's volatile log from the stable log contents, if it has not yet done so. It then replies to p_j with the **duration** $_{jx}$ entries logged at p_i. Even though the access information can be restored from the stable log contents, the data pages which were contained in the volatile log may have not yet been reproduced. Hence, for each **duration** $_{jx}$ sent to p_j, p_i Žor p^x_i. records the value of **duration** $_{jx}$**.version** and the corresponding data page should be sent to p_j later as p_i creates the page during the recomputation. Process p_j begins the recomputation as the access information is collected from every process in the system.

As a result, for every data page logged before the failure, the corresponding log entry, **duration**$_{i\ x}$, can be retrieved from the recovery log, however, the corresponding data page X may not exist in the recovery log when the process p_i begins the recomputation. Note that in this case, the writer of the corresponding page may also be in the recovery procedure. Hence, p_i has to wait until the writer process sends the page X during the recomputation or it may send the request for the page X using the **duration**$_{i\ x}$**.version**. In the worst case, if two processes p_i and p_j concurrently execute the recomputation, the data pages must be retransferred between two processes as they have done before the failure. However, there cannot occur any deadlocked situation, since the data transfer exactly follows the scenario described by the access information in the recovery log and the scenario must follow a sequentially consistent memory model.

Before the recovery process p_i begins its normal computation, it has to reconstruct two more items of information: One is the current operation counter vector and the other is the data page directory. The operation counter vector can be reconstructed from the vector values received from other processes in the system. For each $V\,j_i$w x value, p_i can use the value $V\,i_j$w x retrieved from process p_j, and for the $V\,i_i$w x value, it can use its current **opnum**$_i$ value. The directory includes the ownership and the copy-set information for each data page it owns. The checkpoint of p_i contains the ownership information of the data pages it has owned at the time of checkpointing.

Hence, during the recomputation, p_i can reconstruct its current ownership information as follows: When p_i performs a write operation on a data page, it records the ownership of the page on the directory. When p_i reads a new data page from the log, it invalidates the ownership of that page since the logging means invalidation. However, the copy-set of the data pages the process owns can not be obtained. Since the copy-set information is for future invalidation of the page, the process can put all the processes into the copy-set.

6. The correctness

Now, we prove the correctness of the proposed logging and recovery protocols.

Lemma 1: *The reco¨ery point selected under the proposed reco¨ery protocol is consistent.*

Proof: We prove the lemma by a contradiction. Suppose that a process p_i recovering from a failure selects an inconsistent recovery point, say R_i. Then, p_i must have produced a data page X with **version** $_x$ s **i:k**, where k) R_i, and there must be another process p_j alive in the system, which has read that page. This means that $V\ i_j$w x of p_j must be larger than or equal to k. Since R_i is selected as the maximum value among the V_kw xi values collected, R_i G $V\ i_j$w x and $V\ i_j$w x G k. A contradiction occurs. I

Lemma 2: *Under the proposed logging protocol, a log exists for e¨ery data access point prior to the selected reco¨ery point.*

Proof: For any data access point, if the page used has been transferred from another process, either it was logged before it was transferred Žthe remote write case or it is logged when the page is invalidated the remote read case . If a data. Ž . page locally generated is used for a data access point, either a log is created for the page when the page is invalidated Žthe remote invalidation case. or the log contents can be calculated from the next write point the local invalidation case .Ž . In any case, the page which has not been invalidated before the failure can be retrieved from the current owner. Therefore, for any data access point, the log of the data page can either be found in the recovery log or calculated from other log contents. I

Theorem 1: *A process reco¨ers to a consistent reco¨ery line under the proposed logging and reco¨ery protocols.*

Proof: Under the proposed recovery protocol, a recovering process selects a consistent recovery point ŽLemma 1 , and the logging protocol ensures that for. every data access point prior to the selected recovery point, a data log exists ŽLemma 2 . Therefore, the process recovers to a consistent recovery line.. I

7. The performance study

To evaluate the performance of the proposed scheme, two sets of experiments have been performed. A simple trace-driven simulator has been built to examine the logging behavior of various parallel programs running on the DSM system, then the logging protocols implemented on top of the CVM system to measure the effects of logging under the actual system environments.

7.1. Simulation results

A trace-driven simulator has been built and the following logging protocols have been simulated:

Shared-access tracking (SAT) [23 :] Each process logs the data pages transferred for read and write operations, and also logs the access information of the pages.

Read-write logging (RWL) [11 :] Each process logs the data pages produced by itself, and also, for the data pages accessed, it logs the access information of the pages. In both of the SAT and the RWL schemes, the data pages and the related information are first saved in the volatile storage and then logged into the stable storage when a process creates new dependency by transferring a data page. **Write-triggered logging WTL :( )** This is what we propose in this paper.

The simulation has been run with two different sets of traces: One is the traces synthetically generated using random numbers and the other is the execution traces of some parallel programs.

First, for the simulation, a model with 10 processes is used and the workload is randomly generated by using three random numbers for the process number, the readrwrite ratio, and the page number. One simulation run consists of 100,000 workload records and the simulation was repeated with various readrwrite ratios and locality values. The readrwrite ratio indicates the proportion of read operations to the total number of operations. A readrwrite ratio of 0.9 means that 90% of operations are reads and 10% are writes. The locality is the ratio of memory accesses which are satisfied locally. A locality of 0.9 means that 90% of the data accesses are for the local pages.

The simulation results with the synthetic traces are the ones which best show the effects of logging for the various application program types. Figure 10 and Figure 11 show the effects of the readrwrite ratio and the locality of the application program on the number of logged data pages and the frequency of stable logging, respectively. The number in the parenthesis of the legend indicates the locality. In the SAT scheme, after each data page miss, logging of the newly transferred page is required. Hence, as the write ratio increases, a large number of data pages

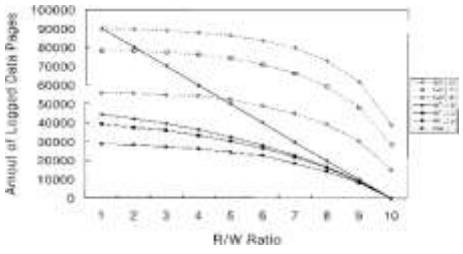


Figure 10. Comparison of the amount of logging synthetic traces .Ž .

become invalidated and a large number of page misses can occur. As a result, the number of logged pages and also the logging frequency are increased. However, as the locality increases, a higher proportion of the page accesses can be satisfied locally, and hence the number of data pages to be logged and the logging frequency can be decreased.

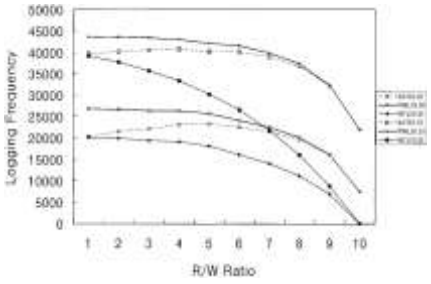


Figure 11. Comparison of the logging frequency synthetic traces .Ž .

In the RWL scheme, the number of logged data pages is directly proportional to the write ratio and this number is not affected by the locality, since each write operation requires logging. However, the stable logging under this scheme is performed when the process creates a new dependent as in the SAT scheme, and hence, the logging frequency of the RWL scheme shows a performance which is similar to that one of the SAT scheme. Comparing the SAT scheme with the RWL scheme, the performance of the SAT scheme is better when both the write ratio and the locality are high, since in such environments, there has to be a lot of logging for the local writes in the RWL scheme.

As for the WTL scheme, only the pages being updated are logged and the logging is performed only at the owners of the data pages. Compared with the SAT scheme in which every process in the copy-set performs the logging, the number of logged data pages is much smaller and the logging frequency is much lower in the WTL scheme. Also, in the WTL scheme, there is no logging for data pages with no remote access and some logging of the write-write precedence order can be delayed. Hence, the WTL scheme shows a much smaller number of logged data pages and a much lower logging frequency compared with the RWL scheme, in which the logging is performed for every write operation. Furthermore, the logging of data pages for the SAT scheme and the RWL scheme require stable storage, while for the WTL scheme, volatile storage can be used for the logging.

To further validate our claim, we have also used real multiprocessor traces for the simulation. The traces contain references produced by a 64-processor MP, running the following four programs: FFT, SPEECH, SIMPLE and WEATHER. Figure 12 and Figure 13 show the simulation results using the parallel program traces. In Figure 12, for the programs, FFT, SIMPLE, and WEATHER, the SAT scheme shows the worst performance, because those programs may contain a large

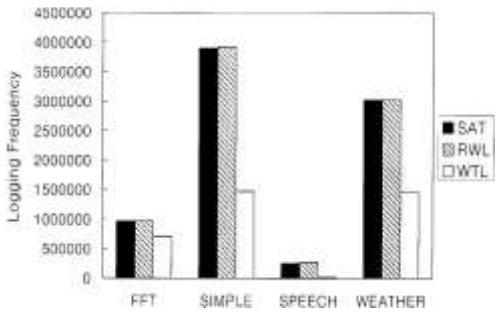


Figure 12. Comparison of the logging amount parallel program traces .Ž .

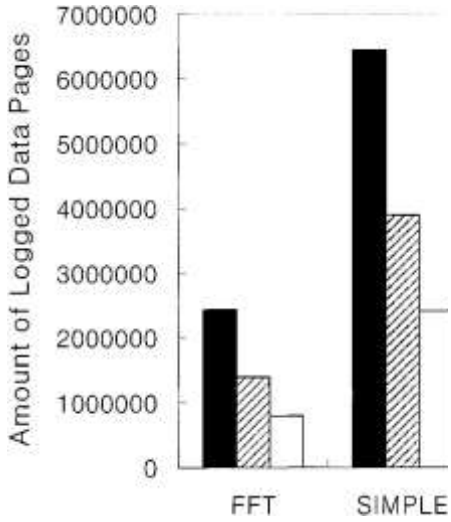


Figure 13. Comparison of the logging frequency parallel program traces .Ž .

number of read operations and the locality of those reads must be low. However, for the program, SPEECH, the RWL scheme shows the worst performance, because the program contains a lot of local write operations. In all cases, the WTL scheme consistently shows the best performance for the log size. Also, considering the logging frequency shown in Figure 13, for all programs, the WTL scheme shows the lowest frequency.

From the simulation results, we can conclude that our new scheme ŽWTL. consistently reduces the number of data pages that have to be logged and also the frequency of the stable storage accesses, compared with the other schemes ŽSAT,RWL . The reduction is more than 50% in most of the cases and it is shown.

in both synthetic and parallel program traces.

7.2. Experimental results

To examine the performance of the proposed logging protocol under the actual system environments, the proposed logging protocol ŽWTL. and the protocol proposed in 23w x ŽSAT have been implemented on top of a DSM system. In order. to implement a sequentially consistent DSM system, we use the CVM CoherentŽ Virtual Machine package 12 , which supports the sequential consistency memory. w x model, as well as the lazy release consistency memory models. CVM is written using Cqq and well modularized and it was pretty straightforward to add the logging scheme. The basic high level classes are the CommManager class and the Msg class which handle the network operation, the MemoryManager class which handles the memory management, and the Page class and the DiffDesc class handling the page management. The protocol classes such as LMW, LSW, and SEQ inherit the high level classes and support operations according to each protocol. We have modified the subclasses in SEQ to implement the logging protocols. We ran our experiments using four SPARCsystem-5 workstations connected through a 10 Mbps ethernet. For the experiments, four application programs, FFT, SOR, TSP, and WATER have been run. Table 2 summarizes the experimental results.

The amount of logged information in Table 2 denotes the number of data pages and the amount of access information which should be logged in the stable storage. For the SAT scheme, data pages with a size of 4K bytes and the access information should be logged, whereas for the WTL scheme, only the access information is logged.

Hence, the amount of information logged in the WTL scheme is only 0.01%]0.5% of that logged in the SAT scheme. The amount of stable logging in the table indicates the frequency of disk access for logging. The experimental results show that the logging frequency in the WTL scheme is only 57%]66% of that in the SAT scheme. In addition to the amount of logged information and the logging frequency, we have also measured the total execution times of the parallel programs under each logging scheme and without logging to compare the logging overhead.

The logging overhead in Table 2 indicates the increases in the execution time under each protocol compared to the execution time under no logging environment, and the comparison of the logging overhead is also depicted in Figure 14. As shown in the table, the SAT scheme requires 20%]189% logging overhead, whereas the WTL scheme requires 5%]85% logging overhead. Comparing these two schemes, the WTL scheme achieves 55%]75% reduction in the logging overhead compared to the SAT scheme. One reason for such a reduction is the low logging frequency imposed by the WTL scheme; the small amount of log information written under the WTL scheme is another possible reason. However, considering the fact that the increases in the number of data pages written per disk access do not cause much increase in the disk access time, the 75% reduction in the logging overhead may require another explanation. One possible explanation is the cascading delay due to the disk access time; that is, the stable logging delays the progress of not only the process which performs the logging, but also the one waiting for the data transfer from the process.

Table 2. Eperimental results

Application program	Logging schemes	Execution time sec.Ž.	Logging overhead %Ž .
TSP	SAT	645.92	117
	WTL	407.29	37
SOR	SAT	419.11	152
	WTL	259.01	56
FFT	SAT	424.47	189
	WTL	272.64	85
WATER	SAT	247.52	20
	WTL	215.15	5

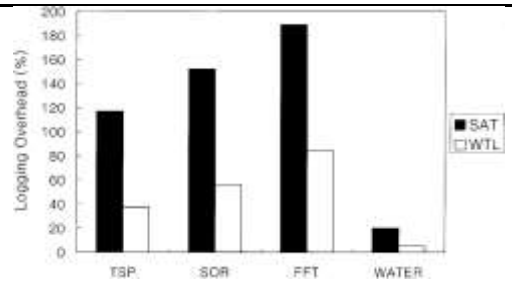


Figure 14. Comparison of the logging overhead.

Overall, the experimental results show that the WTL scheme reduces the amount of logged information and the logging frequency compared to the SAT scheme, and they also show that, in the actual system environment, more reductions in the total execution time can be achieved.

8. Conclusions

In this paper, we have presented a new message logging scheme for DSM systems. The message logging has usually been performed when a data

page is transferred for a read operation so that the process does not have to affect other processes in case of failure recovery. However, the logging to the stable storage always incurs some overhead. To reduce such overhead, the logging protocol proposed in this paper utilizes a two-level log structure; the data pages and their access information are logged into the volatile storage of the writer process and only the access information is duplicated into the stable storage to tolerate multiple failures. The usage of a two-level log structure can speed up the logging and also the recovery procedures with higher reliability.

The proposed logging protocol also utilizes two characteristics of the DSM system. One is that not all the data pages read and written have to be logged. A data page needs to be logged only when it is invalidated by overwriting. The other is that a data page accessed by multiple processes need not be logged at every process site. By one responsible process logging the data page and the related information, the amount of the logging overhead can be substantially reduced.

Through extensive experiments, we have compared the proposed scheme with other existing schemes and concluded that the proposed scheme always enforces much lower logging overhead and the reduction in the logging overhead is more profound when the processes have more reads than writes. Since disk logging slows down the normal operation of the processes, we believe that parallel applications would greatly benefit from our new logging scheme.

References

1. M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed sharedmemory. In *Proc. of the 10th Int'l Conf on Distributed Computing Systems*, pp. 274]281, Jun. 1990.
2. M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger. Causal memory. In *Proc. of the 11th Int'l Conf on Distributed Computing Systems*, pp. 274]281, May 1991.
3. R. E. Ahmed, R. C. Frazier, and P. N. Marinos. Cache-aided rollback error recovery Žcarer. algorithms for shared-memory multiprocessor systems. In *Proc. of the 20th Symp. on Fault-Tolerant Computing*, pp. 82]88, Jun. 1990.
4. G. Cabillic, G. Muller, and I. Puaut. The performance of consistent checkpointing in distributedshared memory systems. In *Proc. of the l4th Symp. on Reliable Distributed Systems*, Sep. 1995.
5. J. B. Carter, A. L. Cox, S. Dwarkadas, E. N. Elnozahy, D. B. Johnson, P. Keleher, S. Rodrigues, W.Yu, and W. Zwaenepoel. Network multicomputing using recoverable distributed shared memory. In *Proc. of the IEEE Int'l Conf. CompCon'93*, Feb. 1993.
6. M. Chandy and L. Lamport. Distributed snapshot: Determining global states of distributed systems.*ACM Trans. on Computer Systems*, 3 1 :63Ž .]75, Feb. 1985.
7. M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight logging for lazy releaseconsistent distributed shared memory. In *Proc. of the USENIX 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.
8. G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributedshared memory multicomputers. In *Proc. of the 13th Symp. on Reliable Distributed Systems*, pp. 42]51, Oct. 1994.
9. B. Janssens and W. K. Fuchs. Relaxing consistency in recoverable distributed shared memory. In*Proc. of the 23rd Annual Int'l Symp. on Fault-Tolerant Computing*, pp. 155]163, Jun. 1993.
10. B. Janassens and W. K. Fuchs. Reducing interprocessor dependence in recoverable shared memory.In *Proc. of the 13rd Symp. on Reliable Distributed Systems*, pp. 34]41, Oct. 1994.

11. S. Kanthadai and J. L. Welch. Implementation of recoverable distributed shared memory by loggingwrites. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems*, pp. 116]123, May 1996.
12. P. Keleher. CVM: The coherent virtual machine. http:www.cs.umd.eduprojectscvm.
13. A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed sharedmemory integrating coherence and recoverability. In *Proc. of the 25th Int'l Symp. on Fault-Tolerant Computing Systems*, pp. 289]298, Jun. 1995.
14. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28 9 :690Ž .]691, Sep. 1979.
15. K. Li. Shared virtual memory on loosely coupled multiprocessors. Ph.D. thesis, Department ofComputer Science, Yale University, Sep. 1986.
16. B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, Aug. 1991.
17. B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computing Sur¨eys*, 10 2 :123Ž .]165, Jun. 1978.
18. M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way toimplement it. *Information Processing Letters*, 39 6 :343Ž .]350, 1991.
19. G. G. Richard III and M. Singhal. Using logging and asynchronous checkpointing to implementrecoverable distributed shared memory. In *Proc. of the 12th Symp. on Reliable Distributed Systems*, pp. 58]67, Oct. 1993.
20. R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerantcomputing systems. *ACM Trans. on Computer Systems*, 1 3 :222Ž .]238, Aug. 1983.
21. M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 54]64, May 1990.
22. M. Stumm and S. Zhou. Fault tolerant distributed shared memory. In *Proc. of the 2nd IEEE Symp. on Parallel and Distributed Processing*, pp. 719]724, Dec. 1990.
23. G. Suri, B. Janssens, and W. K. Fuchs. Reduced overhead logging for rollback recovery indistributed shared memory. In *Proc. of the 25th Annual Int'l Symp. on Fault-Tolerant Computing*, Jun. 1995.
24. V. O. Tam and M. Hsu. Fast recovery in distributed shared virtual memory systems. In *Proc. of the 10th Int'l Conf on Distributed Computing Systems*, pp. 38]45, May 1990.
25. K. L. Wu and W. K. Fuchs. Recoverable distributed shared memory. *IEEE Trans. on Computers*, 39 4 :460Ž .]469, Apr. 1990.