An Optimistic-Based Partition-Processing Approach for Distributed Shared Memory Systems

JENN-WEI LIN AND SY-YEN KUO[*]
Department of Computer Science and Information Engineering
Fu Jen Catholic University
Taipei, 242 Taiwan
E-mail: jwlin@csie.fju.edu.tw
[*]*Department of Electrical Engineering*
National Taiwan University
Taipei, 107 Taiwan
E-mail: sykuo@ee.ntu.edu.tw

This paper investigates the problem of network partitioning in Distributed Shared Memory (DSM) systems. We propose an optimistic-based partition-processing approach, which can make shared pages available when network partitioning occurs. However, this approach does not guarantee that the same page in different partitions can maintain a consistent value. To eliminate this problem, a memory-based coordinated checkpointing scheme is presented to save consistent states at low cost. If there are inconsistencies between two partitions, one saved consistent state is chosen to perform backward error recovery. Extensive trace-driven simulations have been performed to evaluate the effects of the proposed approach on system performance.

Keywords: network partitioning, distributed shared memory, checkpointing, backward error recovery, trace-driven simulations

1. INTRODUCTION

A Distributed Shared Memory (DSM) system provides a shared memory abstraction on a network of computers [1]. In such a system model, programmers do not need to be concerned about data movement between processing nodes. The chore of interprocesor communication is left to the DSM system. Applications in such a system model can be easily programmed as if they are executing on a real shared memory machine.

As the number of components in a DSM system and the running time of applications increase, the probability that a failure will occur during the execution of an application also increases. Fault tolerance in DSM systems has been studied extensively [2-12]. Most of the existing approaches only tolerate node failures and make the impractical assumption that the network environment is immune to partitioning. However, the processing nodes in a DSM system are interconnected through a network. Besides nodes, the communication links in the network may also fail. In this case, all the processing nodes may be divided into several disconnected groups (partitions). Nodes in one partition can not communicate with the nodes in other partitions. This phenomenon is known as network partitioning.

Received August 31, 2001; accepted April 15, 2002.
Communicated by Jang-Ping Sheu, Makoto Takizawa and Myongsoon Park.

The network partitioning problem has been extensively discussed with respect to distributed database systems. The existing partition-processing approaches are classified into two categories: pessimistic and optimistic [13]. Pessimistic-based approaches prevent inconsistencies between partitions by restricting the availability of data objects. A data object is allowed to be accessed in one partition only if the consistency of this data object between partitions can be ensured. In contrast, optimistic-based approaches allow data objects to be accessed in each partition without considering the consistency between partitions. When partitions are to be reconnected, optimistic-based approaches need to eliminate the inconsistency between the partitions. Therefore, consistency is sacrificed to achieve high data availability. However, the partition-processing approaches for distributed database systems are not suitable for the DSM system environment. In distributed database systems, the number of copies of a data object is fixed and is the same for all data objects. The number of copies for a shared page in DSM systems changes dynamically as the program is being executed .

Up to now, only the approach in [9] has attacked the network partitioning problem in DSM systems. This approach has a dynamic coherence protocol to guarantee a chosen degree of data availability based on the paid cost. During the period of network partitioning, the dynamic coherence protocol still provides a degraded degree of data availability according to the partitioning status. When a data item is allowed to be accessed in a partition, its consistency between partitions is also ensured. However, this approach only claims that it is capable of handling the network partitioning problem, but does not clearly describe how to handle the problem. In addition, each shared page needs to dynamically maintain the minimum number of up-to-date copies in order to guarantee a given degree of data availability. The trade-off between the degree of fault tolerance and the up-to-date cost has been further studied in [10].

The main goal of this paper is to design an efficient approach to tolerating network partitioning in DSM systems. Unlike the approach in [9], the proposed approach is based on the optimistic viewpoint. It consists of a one-copy read and one-copy write access scheme and a memory-based coordinated checkpointing scheme. The one-copy read and one-copy write access scheme is used to guide access to the shared pages during the network partitioning period. The memory-based coordinated checkpointing scheme is used to resolve inconsistency between partitions. Basically, our approach and the approach in [9] work from two different viewpoints (the optimistic viewpoint and the pessimistic viewpoint) to handle the network partitioning problem. It is hard to determine which approach is better. Compared to the approach in [9], our approach offers higher data availability but complicates the task of reconnecting partitions since inconsistencies between partitions may occur. To evaluate the effect of our approach on system performance, trace-driven simulations were performed to quantify the data availability and the inconsistency effect.

The remainder of this paper is organized as follows. Section 2 describes the system model and terminology. Section 3 presents an optimistic partition-processing approach to allowing data to be available during network partitioning as well as to resolving inconsistencies between partitions. Section 4 evaluates the effects of the proposed approach on system performance. Finally, we give concluding remarks in section 5.

2. BACKGROUND

This section provides background material on the research on network partitioning for DSM systems. First, a brief introduction of a DSM system model is given and possible failures are described. Next, the consistency protocol used in the DSM system is discussed.

2.1 System Model

The system model is shown in Fig. 1. There is a virtually memory shared by all the processing nodes. Each processing node communicates with others via the virtually shared memory. The address space of the virtually shared memory is organized into a set of shared pages. A shared page is the basic unit for data transfer and consistency maintenance between processing nodes. An application program is divided into multiple processes, and each process is executed in a processing node.

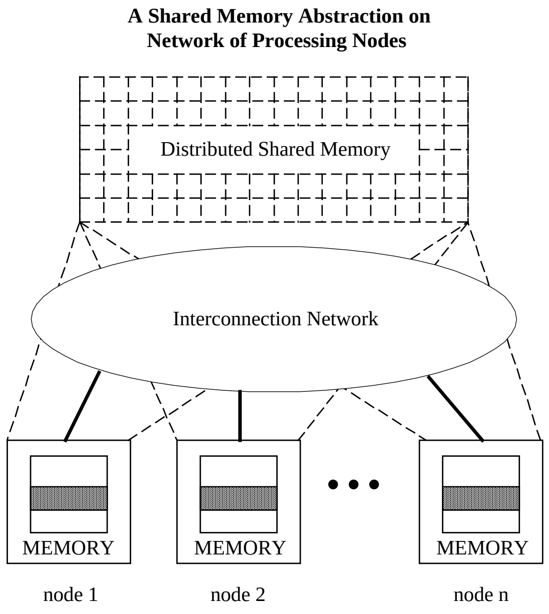

A Shared Memory Abstraction on
Network of Processing Nodes
Distributed Shared Memory
Interconnection Network
MEMORY
MEMORY
MEMORY
node 1
node 2
node n

Fig. 1. System model.

Unlike the previous research on fault tolerance in DSM systems [2-8, 11, 12], the reliable network assumption is not made in this paper. If a node or a communication link fails, the whole communication network may be partitioned into several disconnected groups. The communication network is not immune to partitioning.

2.2 Consistency

Consistency protocols used in DSM systems include the release consistency, weak consistency, processor consistency, and sequential consistency protocols [9]. The release consistency protocol is the least restrictive one of the above protocols, and it allows a high degree of parallelism by utilizing application program semantics. On the other hand, the sequential consistency restricts parallelism and sacrifices some degree of performance. However, in this consistency protocol, no additional program semantic knowledge is needed. This is the reason why the sequential consistency protocol has been adopted in some recent studies on of recoverable DSM systems [9, 11, 12].

The consistency among shared pages is assumed to be managed by a fixed distributed manager (FDM) protocol [14], which is a sequential consistency protocol. The task of consistency maintenance in the FDM protocol is distributed among all the nodes. The association between a shared page and its consistency manager is determined statically by a mapping function. Two pieces of information, the owner and the copyset are recorded in a manager. The owner indicates which node owns the shared page. The copyset lists the identifiers of the nodes which have a copy of the shared page. If node *x* attempts to access shared-page *p* on which it does not have the access right, it will send a *read-fault* or a *write-fault* message to the manager of shared-page *p*. For a read fault on shared-page *p*, the *read-fault generating node* (the node issuing the read fault) first sends a *read-fault message* to the default manager of shared-page *p*. The manager then asks the owner of shared-page *p* to send a copy to the read-fault generating node and includes the identifier of the generating node in the copyset of shared-page *p*. For a write fault on shared-page *p*, the *write-fault generating node* (the node issuing the write fault) sends a *write-fault message* to the default manager of shared-page *p*. The manager asks the owner of shared-page *p* to send a copy to the write-fault generating node, sends an invalidation message to each node that has an old copy of shared-page *p*, sets the copyset of shared-page *p* to null, and makes the generating node the new owner of shared-page *p*.

3. NETWORK PARTITIONING

This section proposes a new partition-processing approach for DSM systems. To achieve high data availability, a one-copy read and one-copy write protocol is used to guide access to shared pages as much as possible during network partitioning. However, since the proposed partition-processing approach is an optimistic based approach, consistency between partitions can not be guaranteed. A memory-based coordinated checkpointing scheme is also proposed to preserve consistent states at appropriate access points in advance. If two partitions are to be reconnected, one preserved consistent state is chosen to eliminate inconsistency between the two partitions.

3.1 Availability

After network partitioning, not all accesses to shared pages can be executed successfully. For example, as shown in Fig. 2, due to network partitioning, the system with 5 processing nodes and 2 shared pages is divided into two partitions. Node 1, node 2, and node 3 are located in Partition 1. Node 4 and node 5 are located in Partition 2. The copies of shared-page 1 and shared-page 2 are distributed in these two partitions. Partition 1 has two read-only copies of shared-page 1 and the owner copy of shared-page 2. Partition 2 has the owner copy of shared-page 1 and one read-only copy of shared-page 2. Based on

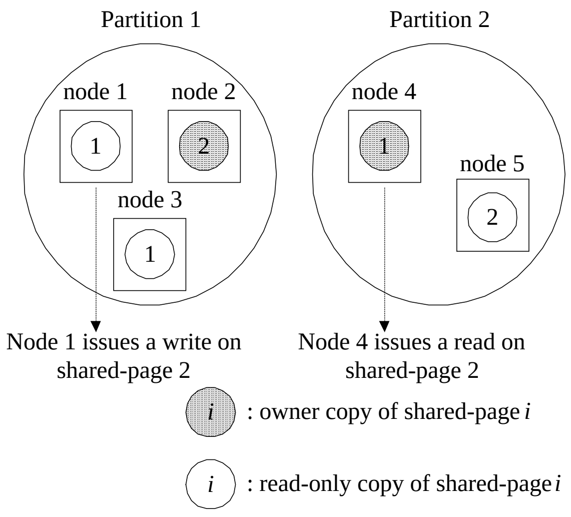

Partition 1
Partition 2
node 1
node 2
node 3
node 4
node 5
1
2
1
1
2
Node 1 issues a write on shared-page 2
Node 4 issues a read on shared-page 2
i : owner copy of shared-page i
i : read-only copy of shared-page i

Fig. 2. Partition scenario.

the fixed distributed manager protocol (see Subsection 2.2), if a node in Partition 1 (e.g., node 1) has a write-fault on shared-page 2, this write access can not be completed since one read-only copy of shared-page 2 in Partition 2 can not be invalidated. Similarly, if a node in Partition 2 (e.g., node 4) has a read-fault on shared-page 2, this read access also can not be completed since the owner copy of shared page 2 is not in Partition 2. To ensure that shared pages can still be accessed after network partitioning, we propose a shared-page access protocol during network partitioning, called the one-copy read and one-copy write protocol. This access protocol works as follows.

- A read on shared-page *p* (r_p) in partition *i* can be executed as long as there is at least one (owner or read-only) copy of shared-page *p* in partition *i*. Otherwise, r_p will be blocked.
- A write on shared-page *p* (w_p) in partition *i* can be executed as long as there is at least one copy of shared-page *p* in partition *i*. Otherwise, w_p will be blocked.

In the protocol, if no copy of shared-page *p* is in partition *i*, then none of the nodes in partition *i* can supply the data of shared-page *p*. Based on the access protocol, the previous two non-executable shared accesses (see Fig. 2) can now be executed.

The accessibility of shared pages during the network partitioning period is based on an optimistic assumption. The assumption is that two or more partitions seldom access the same shared page during the partitioning period. Even if a shared page is accessed by different partitions during the partitioning period, the probability that there will be conflicts between accesses to the page is very small. If a program has a small number of shared-write references, the above assumption is quite realistic since two accesses to the same page will conflict only if at least one of them is a write. The reference characteristics of programs studied by Eggers and Katz [15] indicate that the ratio of the number of write accesses to shared data over the total number of memory accesses is very small and ranges from 0.00005 to 0.001. Therefore, the optimistic assumption made in the proposed access scheme is quite reasonable.

In addition, after network partitioning, the consistency of shared pages may not be maintained. For example, in Fig. 2, the consistency of shared-page 2 is first assumed to be managed by a node in Partition 2. Now, if a write on shared-page 2 is allowed to be executed in Partition 1, the new consistency information of shared-page 2 can not be kept in its default manager. However, the maintenance range of a shared page's consistency can be shrunk within a partition. If an access to a shared page is allowed to be executed in a partition, it is also required that the consistency of this page within the partition be maintained. There are two different algorithms for maintaining consistency information: the centralized manager and distributed manager algorithms [14]. The method for maintaining consistency within a partition using the centralized manager algorithm is similar to that using the distributed manager algorithm. To conveniently demonstrate how to maintain consistency within a partition, the centralized manager algorithm is used here. After network partitioning, each partition specifies one node in itself as the default consistency manager. The manager keeps two pieces of consistency information, the owner and the copyset, in a table called *Info*, which has an entry for each shared page. In partition *i*, the owner of shared-page *p* in the *Info* table (*Info[p].owner*) indicates which node in partition *i* currently owns shared-page *p*. The copyset of shared-page *p* in *Info* (*Info[p].copyset*) indicates which nodes in partition *i* have a copy of shared-page *p*. The detailed algorithm for handling both shared accesses and consistency maintenance within a partition is shown in Fig. 3.

As shown in Fig. 3, the read (write) fault handler is invoked whenever a node in partition *i* attempts to access a shared page on which it does not have the access privilege. The read (write) fault handler contacts the manager of partition *i* to determine whether the current read (write) fault access can be serviced or not, as follows. The entry of the requested page in *Info* is first located. If this entry has not been yet established, a special message, *Locate_Copy*, is broadcast in partition *i* to collect the consistency information of the requested page within partition *i*. When a node receives a *Locate_Copy* message, if it has a copy of the requested page, it sends the attribute of the copy (owner or copyset) to the manager. After reconstructing the consistency information of the requested page, if the fault access is a read fault, the manager can forward this read-fault access to an arbitrary node that has a copy of the requested page. If the fault access is a write fault, the manager can send an invalidation message to every node which has an old copy of the requested page.

Read Fault Handler

```
send read-fault request to the manager;
if (this read access can not be completed)
    /*a non-service message is received*/
    put this read-fault request in pending queues;
else
    receive a copy of shared-page p;
    set the permission of the copy of shared-page p to read-only;
    execute the read-fault access to shared-page p;
end if;
```

Fig. 3. Algorithm for shared accesses and consistency maintenance within a partition.

```
Read Request Handler
if (I am a member of the copyset of shared-page p)
    send a copy of shared-page p to the requesting node;
end if;
if (I am the consistency manager)
    if (Info[p] has not been established)
        /*there is a first access to shared-page p in partition i*/
          broadcast a special message $Locate\_Page_p$ to each node in partition i;
          if (none of the nodes in partition i has a copy of shared-page p)
            return a non-service message;
        else
          establish the entry Info[p];
          fill Info[p].owner and Info[p].copyset according to the acknowledgement of
          $Locate\_Page_P$ message;
        end if;
    end if;
    $Info[p].copyset \leftarrow Info[p].copyset \cup \{requesting\ node\}$;
    ask one member of shared-page p's copyset to send a copy;
end if;
Write Fault Handler
send a write-fault request to the manager;
if (this write access can not be completed)
    /*a non-service message is received*/
put this write-fault request in pending queues;
else
receive a copy of shared-page p;
set the permission of the copy of shared-page p to read-write;
execute the write-fault access to shared-page p;
end if;
Write Request Handler
if (I have a copy of shared-page p)
invalidate the copy of shared-page p;
end if;
if (I am the consistency manager)
    if (Info[p] has not been established)
        /*there is a first access to shared-page p in partition i*/
          broadcast a special $Locate\_Page_P$ message to each node in partition i;
          if (none of the nodes in partition i has a copy of shared-page p)
            return a non-service message;
        else
          establish the entry Info[p];
          fill Info[p].owner and Info[p].copyset according to the acknowledgement of
          $Locate\_Page_P$ message;
        end if;
    end if;
    send invalidation messages to all members of Info[p].copyset;
    $Info[p].owner \leftarrow \{requesting\ node\}$;
    $Info[p].copyset \leftarrow \{\}$;
end if;
```

Fig. 3. (Cont'd) Algorithm for shared accesses and consistency maintenance within a partition.

3.2 Inconsistency Between Partitions

Consistency between partitions can not be guaranteed since an optimistic assumption is made in the one-copy read and one-copy write protocol. To demonstrate this phenomenon, Fig. 2 is used again to show an inconsistency example. If a node in Partition 1 has a write on shared-page 2, this write is allowed to be executed. The read-only copy of shared-page 2 in Partition 2 can not be invalidated. Thereafter, if a node in Partition 2 has a read on shared-page 2, it will read an old value of shared-page *p*. Hence, there is a write-read conflict between the accesses to shared-page 2. This also implies that there is inconsistency with respect to shared-page 2 between Partition 1 and Partition 2.

From the above description, we observe that if all the shared accesses executed in Partition 1 and Partition 2 are traced, the inconsistency between Partition 1 and Partition 2 can be revealed by detecting whether there are conflicts between the two partitions' shared accesses. The shared-access tracing of Partition 1 and Partition 2 can be integrated with the consistency management of Partition 1 and Partition 2, respectively. As mentioned earlier, each partition uses the *Info* table to manage consistency within itself. Here, the *Info* table is extended to add two fields: *read* and *write*. During the network partitioning period, when shared-page *p* is first read by a node in Partition 1 (Partition 2), a read-tracing message for shared-page *p* is sent to the manager of Partition 1 (Partition 2). The manager then puts a "*read*" mark in shared-page *p*'s entry in the *Info* table ($Info[p].read \leftarrow$ "1"). Similarly, when shared-page *p* is first written by a node in Partition 1 (Partition 2), a "*write*" mark is also entered in shared-page *p*'s entry in the *Info* table ($Info[p].write \leftarrow$ "1").

In the following two subsections, we will further discuss how to detect and resolve inconsistency between partitions using tracing information.

3.2.1 Inconsistency detection

If there is inconsistency with respect to shared-page *p* between partition i_1 and partition i_2, the following two properties must hold:

- Shared-page *p* has been accessed by both partition i_1 and partition i_2.
- There is a conflict between the accesses to shared-page *p*.

The inconsistency between partition i_1 and partition i_2 is detected as follows:

- Find all the multi-accessed pages between partition i_1 and partition i_2. (A shared page is a multi-accessed page if it has been accessed by two or more partitions.)
- Check each multi-accessed page to determine whether there is a conflict between the accesses to it.

All the multi-accessed pages between partition i_1 and partition i_2 are found as follows. *Info* is used to manage consistency within a partition. If partition i_1 and partition i_2 are to be reconnected, both *Info* tables ($Info_{i_1}$ and $Info_{i_2}$) first need to be merged in order to form a new *Info* table to manage consistency within the new reconnected partition.

During the merging the two *Info* tables of partition i_1 and partition i_2 if shared-page p has an entry in both $Info_{i_1}$ and $Info_{i_2}$, this means that shared-page p has been accessed by both partition i_1 and partition i_2. Therefore, shared-page p is a multi-accessed page between partition i_1 and partition i_2.

The next step is to determine the multi-accessed pages that have introduced inconsistency between partition i_1 and partition i_2. This task can be done by applying the following conflict-detection function f to each multi-accessed page:

$$\begin{aligned} f(Info_{i_1}[p], Info_{i_2}[p]) = {} & (Info_{i_1}[p].read \wedge Info_{i_2}[p].write) \rightarrow \text{read-write conflict} \\ & \text{or } (Info_{i_1}[p].write \text{ and } Info_{i_2}[p].read) \rightarrow \text{write-read conflict} \\ & \text{or } (Info_{i_1}[p].write \text{ and } Info_{i_2}[p].write) \rightarrow \text{write-write conflict,} \end{aligned}$$

where p is the identifier of a multi-accessed paged, $Info_{i_1}[p]$ represents the entry of shared-page p in the *Info* table of partition i_1, and $Info_{i_2}[p]$ represents the entry of shared-page p in the *Info* of partition i_2.

If function f returns the value "0", this means that all the accesses to the multi-accessed page p in both partition i_1 and partition i_2 are "*read*" operations. Hence, there is no conflict between the accesses to the multi-accessed page p. In this case, the entry of shared-page p in the new *Info* table can be formed by directly combining two entries: $Info_{i_1}[p]$ and $Info_{i_2}[p]$. On the other hand, if function f returns the value "1", then there is at least one conflict (read-write conflict, write-read conflict, or write-write conflict) between the accesses to shared-page p. In this case, shared-page p is an inconsistent page between partition i_1 and partition i_2 (the value of shared-page p in partition i_1 may not be the same as that in partition i_2). The entry of shared-page p in the new *Info* table can not be formed.

3.2.2 Inconsistency resolution

To resolve inconsistency between partition i_1 and partition i_2, an intuitive idea is to reconstruct the consistency of each inconsistent page between partition i_1 and partition i_2. However, an inconsistent page may further propagate its own inconsistency. For example, in Fig. 4, shared-page 1 propagates its inconsistency to shared-page 2 and shared-page 3 since the data written on both shared-page 2 and shared-page 3 is dependent on the data of shared-page 1. In this situation, although shared-page 2 and shared-page 3 are not detected as inconsistent pages, the data written in them is incorrect. The inconsistency propagation problem needs to be considered in inconsistency resolution. The idea for resolving the inconsistency between partition i_1 and partition i_2 is thus modified as follows: roll back all nodes in partition i_1 and partition i_2 to a state without any inconsistency dependency with partition i_1 and partition i_2, respectively.

To achieve the above goal, a coordinated checkpointing scheme is employed to save consistent states at appropriate access points. Whenever shared-page p is first accessed in partition i, the current consistent state with respect to shared-page p in partition i is also saved. The saved consistent state can form a recovery line with respect to shared-page p in partition i. When partition i and other partitions are to be reconnected, if shared-page p

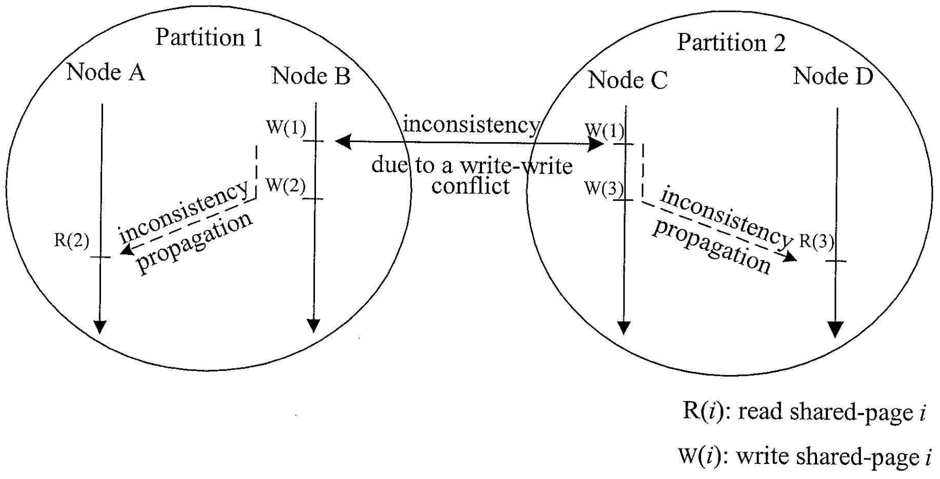

Partition 1
Node A
Node B
W(1)
W(2)
R(2)
inconsistency propagation
inconsistency due to a write-write conflict
Partition 2
Node C
Node D
W(1)
W(3)
R(3)
inconsistency propagation
R(i): read shared-page i
W(i): write shared-page i

Fig. 4. Example inconsistency propagation.

is detected as an inconsistent page between partition *i* and other partitions, then all its introduced accesses (first access, second access etc.) and inconsistency propagation in partition *i* can be nullified by rolling back all the nodes in partition *i* to the recovery line with respect to shared-page *p*. After rolling back, the original accesses to shared-page *p* in partition *i* can be re-executed in the new reconnected partition under a safe execution sequence.

There are two techniques for saving a consistent state: asynchronously coordinated checkpointing and synchronously coordinated checkpointing [16]. With the first technique, the consistent state of a partition is incrementally maintained in each node. When shared-page *p* is first accessed in a partition, only the accessing node generates a checkpoint. Other nodes in the same partition do not generate checkpoints until they have a dependency on the accessing node (they read a shared page from the accessing node or send an invalidation message to the accessing node). With the second technique, when shared-page *p* is first accessed in a partition, all the nodes in the partition simultaneously generate checkpoints to maintain the consistent state with respect to shared-page *p*. To avoid taking unnecessary checkpoints in some nodes, we adopt the asynchronously coordinated checkpointing technique. Fig. 5 illustrates how the inconsistency shown in Fig. 4 can be resolved using the asynchronously coordinated checkpointing technique. In Fig. 5, when shared-page 1 is first accessed in Partition 1 at time t_1, only node B generates a checkpoint to maintain the recovery line with respect to shared-page 1. Node A does not generate the consistent checkpoint until it reads shared-page 2 from node B at time t_3. The recovery line with respect to shared-page 1 in Partition 2 can also be obtained similarly. Finally, when Partition 1 and Partition 2 are to be reconnected at time t_4, shared-page 1 is detected as an inconsistent page between both. All the nodes in Partition 1 and Partition 2 are rolled back to the respective recovery line with respect to shared-page 1.

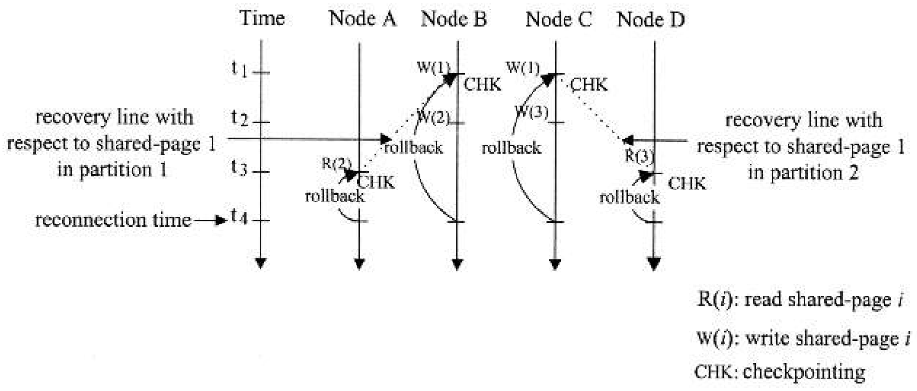

Time
Node A
Node B
Node C
Node D
t1
t2
t3
t4
recovery line with respect to shared-page 1 in partition 1
reconnection time
recovery line with respect to shared-page 1 in partition 2
W(1)
CHK
W(2)
W(3)
R(2)
R(3)
rollback
R(i): read shared-page i
W(i): write shared-page i
CHK: checkpointing

Fig. 5. Inconsistency elimination example.

3.2.3 Implementation issues

The consistent states saved to resolve inconsistency can be put in the memory space since they will not be used to handle node failures. If a node accesses an inconsistent page or suffers form inconsistency propagation, the consistent states saved in its main memory are still valid. However, the amount of memory space required to save all the states of nodes may be large. To reduce the size of the required memory space, the copy-on-write technique [17] is applied here. The copy-on-write technique can reduce the amount of information saved in the consistent state and overlap execution of the target program with generation of the consistent state. The main idea is to break up the consistent state into small pieces, and to then work on one piece at a time only when necessary. The detailed procedure is described as in the following.

- When a node is required to generate a checkpoint to maintain a consistent state with respect to a shared page, the current state of the node is first divided into the CPU state and the memory state. The CPU state indicates the current values of the node's registers.
- A fixed size pool of pages is allocated from the memory space. Execution of the node is frozen, and the CPU state is entirely copied to the page pool. At the same time, the protection bits of all the pages in main memory are set to be "read-only."
- Then, execution of the node is restarted, and a copy thread is initiated. The copy thread copies a page to the page poll only if the protection bit of the page is violated (one process in the node writes on the page). If a page is not written by any process, it is not required that it be copied to the page pool. Once the page pool is full, a writer thread is started to write the page pool to stable storage. The page pool is then emptied. The memory state is saved incrementally.

4. EVALUATION

This section evaluates the effects of the proposed partition-processing approach on system performance. We first introduce the simulation environment and then present the simulation results.

4.1 Trace-Driven Simulation

Trace-driven simulations are used in studies of system performance to avoid building expensive prototypes [18-20]. These simulations are driven by using address traces. The address traces of a program are the memory references while it is being executed. The three programs used in the trace-driven simulations were: FFT, Simple and Weather. The FFT program is a radix-2 fast Fourier transform. The Simple program models the behavior of fluids and employs the finite difference method to solve equations describing hydrodynamic behaviors. The Weather program partitions the atmosphere into a three-dimensional grid and employs the finite difference method to solve equations describing the states of the system. The three programs show a wide range of difference in load balancing. The various patterns of communication between nodes can be extensively derived by the three programs. The characteristics of the three traces are summarized in Table 1. The total number of memory references for each of the three programs is the total sum of the number of data reads, the number of data writes, and the number of code reads.

Table 1. Trace characteristics.

Application Program	Total number of references	Number of data reads		Number of data writes		Number of code reads
		total	shared	total	shared	
FFT	7,440,001	2,936,935	528,604	1,390,178	516,961	3,112,888
Simple	27,030,092	11,541,252	5,048,671	3,894,668	451,207	11,594,172
Weather	31,764,036	15,579,599	4,531,576	2,546,230	484,724	13,638,207

The effects of the proposed partition-processing approach on system performance were evaluated in terms of the following metrics:

- How many operations are executed correctly in a node during network partitioning? An operation is said to be executed correctly in a node if it neither accesses any inconsistent page nor suffers from inconsistency propagation.
- How many checkpoints are taken in a node to resolve inconsistency?
- What percentage of operations is rolled back to resolve inconsistency?

The first metric was treated as the advantage of the proposed approach. The second metric was regarded as the failure-free overhead incurred by the proposed approach. The third metric was regarded as the effect of the inconsistency introduced by the proposed approach.

The workload parameters used in the simulations were specified as follows:

- The numbers of partitions varied from 2 to 16 to illustrate the effects of various partition sizes on the above three performance metrics.
- The average number of cycles required for a local operation was set to 4 cycles.

- The average number of cycles required for a page fault operation was set to 6 cycles.
- The page size used in the simulations was 1 kilobytes.
- Each of the following simulation results is the average of 100 simulation runs. The reference patterns (address traces) used in each simulation run were different from those used in others. Therefore, the simulation results were not dependent on any specific reference pattern. The network partitioning interval (the time period between the occurrence of network partitioning and the moment when the network was fully reconnected) in each simulation run was generated randomly.

4.2 Simulation Results

Fig. 6 shows the advantage achieved by the proposed partition-processing approach, with the number of partitions varying from 2 to 16. As shown in Fig. 6, the number of operations executed correctly ranged from 782 to 1670. The average network partitioning interval among 100 simulation runs was 10,000 cycles. Based on the second workload parameter, we can further infer that the average maximum number of operations executed during the partitioning period in 100 simulation runs was 2500 (10000/4). The percentage ratio of the number of correct operations over the maximum number of operations executed during the partitioning period ranged from 31% to 67%. The percentage advantage is dependent on the access point of the node's first inaccessible page since if a node can not access a shared page from its partition, its execution will be blocked. In addition, if a node has accessed an inconsistent page or suffered from inconsistency propagation, it is also required that it be rolled back. Therefore, the percentage advantage is also dependent on the rollback point for resolving inconsistency. In conclusion, if the first inaccessible point or rollback point of a node is closer to the point of partitioning, the node will have a smaller percentage advantage. Conversely, if most of a node's shared accesses during partitioning period can be executed correctly, the node will have a larger percentage advantage.

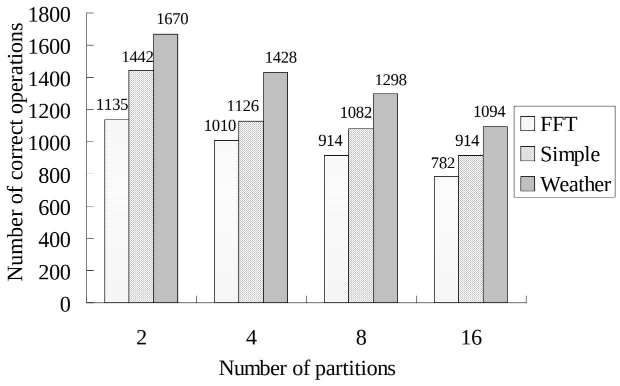


Fig. 6. Advantage of the proposed partition-processing approach with the partitioning period = 2500 time units (references).

Fig. 7 shows the failure-free overhead incurred by the proposed partition-processing approach, with the number of partitions varying from 2 to 16. The number of checkpoints taken for resolving inconsistency was rather small. The percentage of the number of checkpoints over the maximum number of operations (2,500 operations) allowed during the partitioning period ranged from 0.4% to 3.2%. The number of checkpoints taken to resolve inconsistency depends on how many shared pages are accessed in a partition. In a program, references to shared pages usually exhibit temporal locality (shared pages currently being accessed will likely be accessed again in the near future). This property implies that few shared pages are accessed during the partitioning period. Therefore, the number of checkpoints taken to resolve inconsistencies is small. Here, the checkpoints taken are incrementally saved in memory (see subsection 3.3) based on the copy-on-write technique. The size of a checkpoint is dependent on the CPU state plus the number of write operations issued during network partitioning, not the size of a full node's state. If few write operations are executed during network partitioning, the size of a taken checkpoint is small.

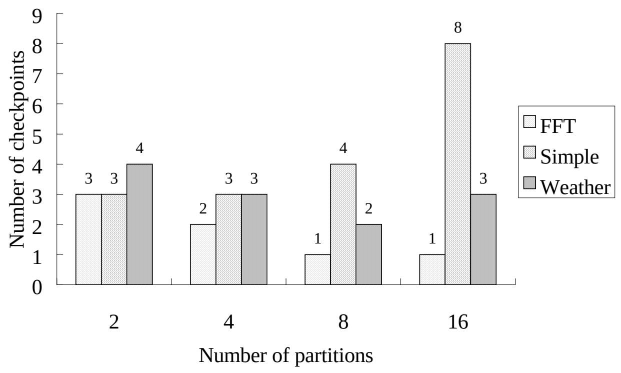


Fig. 7. Overhead of the proposed partition-processing approach with the partitioning period = 2,500 time units (references).

Fig. 8 shows the effect of the inconsistency introduced by the proposed partition-processing protocol, with the number of partitions varying from 2 to 16. The inconsistency effect (the percentage of the number of operations required to be rolled back over the number of operations allowed to be executed) ranged from 7.7% to 24.7%. In other words, up to 92.3% (7.7%) of the operations executed during network partitioning did not introduce any inconsistency. In the worst case, 75.3% of the operations still did not introduce any inconsistency during network partitioning.

5. CONCLUSIONS

The basic idea of the proposed approach is derived from one the DSM's own properties: each shared page in the DSM system usually has several copies in distinct nodes. Therefore, after network partitioning, if a node wants to access a shared page, one or

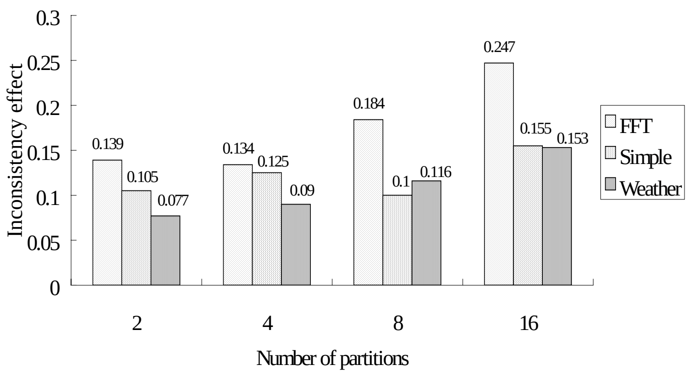


Fig. 8. Inconsistency of the proposed partition-processing approach with the partitioning period = 2500 time units (references).

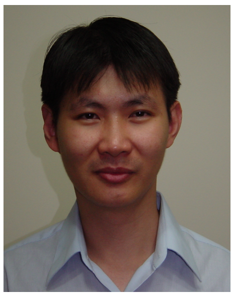
more copies of the page may exist in the node's located partition. The one-copy read and one-copy write protocol has been proposed and used to show how the accessibility requirement of shared pages can be well satisfied. However, the proposed approach is an optimistic-based partition-processing approach. If two partitions are to be reconnected, there may be inconsistencies between them. To resolve inconsistencies, a memory-based coordinated checkpointing scheme has also been presented. Finally, simulation results were been obtained and used to show that the level of data availability provided ranges from 31% to 67%, and that the level of the introduced inconsistency ranges from 7.7% to 24.7%.

REFERENCES

1. B. Nitzberg and V. Lo, "Distributed shared memory: a survey of issues and algorithms," *IEEE Computer*, Vol. 24, 1991, pp. 52-60.
2. K. L. Wu and W. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Transactions on Computers*, Vol. 39, 1990, pp. 460-469.
3. G. G. Richard III and M. Singhal, "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory," in *Proceedings of 12th International Symposium on Reliable Distributed Systems*, 1993, pp. 58-67.
4. B. Janssens and W. K. Fuchs, "Relaxing consistency in recoverable distributed shared memory," in *Proceedings of 23rd International Symposium on Fault Tolerant Computing Systems*, 1993, pp. 155-163.
5. G. Janakiraman and Y. Tamir, "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers," in *Proceedings of 13th International Symposium on Reliable Distributed Systems*, 1994, pp. 42-51.
6. G. Suri, B. Janssens, and W. K. Fuchs, "Reduced overhead logging for rollback recovery in distributed shared memory," in *Proceedings of 25th International Symposium on Fault-Tolerant Computing Systems*, 1995, pp. 279-288.
7. A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A recoverable dis-

tributed shared memory integrating coherence and recoverability," in *Proceedings of 25th International Symposium on Fault-Tolerant Computing Systems*, 1995, pp. 289-298.

8. B. Janssens and W. K. Fuchs, "Eusuring correct rollback recovery in distributed shared memory systems," *Journal of Parallel and Distributed Computing*, Vol. 29, 1995, pp. 211-218.
9. O. E. Theel and B. D. Fleisch, "A dynamic coherence protocol for distributed shared memory enforcing high data availability at low costs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, 1996, pp. 915-930.
10. B. D. Fleisch, H. Michel, S. K. Shah, and O. E. Theel, "Fault tolerance and configurability in DSM coherence protocols," *IEEE Concurrency*, Vol. 8, 2000, pp. 10-21.
11. C. Morin, A. M. Kermarrec, M. Banatre, and A. Gefflaut, "An efficient and scalable approach for implementing fault-tolerant DSM architectures," *IEEE Transactions on Computers*, Vol. 49, 2000, pp. 414-430.
12. J. W. Lin and S. Y. Kuo, "A new log-based approach to independent recovery in distributed shared memory systems," *Journal of Information Science and Engineering*, Vol. 16, 2000, pp. 271-290.
13. S. B. Davidson, H. G. Molina, and D. Skeen, "Consistency in partitioned networks," *ACM Computing Surveys*, Vol. 17, 1985, pp. 341-370.
14. K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, Vol. 7, 1989, pp. 321-359.
15. S. J. Eggers and R. H. Katz, "A characterization of sharing in parallel programs and its application to coherency protocol evaluation," in *Proceedings of 24th International Symposium on Computer*, 1988, pp. 373-382.
16. J. L. Kim and T. Park, "An efficient protocol for checkpointing recovery in distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, 1993, pp. 955-960.
17. K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, 1994, pp. 874-879.
18. R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: a survey," *ACM Computing Surveys*, Vol. 29, 1997, pp. 128-170.
19. R. Giorgi and C. A. Prete, "PSCR: a coherence protocol for eliminating passive sharing in shared-bus shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, 1999, pp. 742-763.
20. A. S. Vaidya, A. Sivasubramaniam, and C. R. Das, "Impact of virtual channels and adaptive routing on application performance," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, 2001, pp. 223-237.



Jenn-Wei Lin (林振緯) received the M.S. degree in computer and information science from National Chiao Tung University, Hsinchu, Taiwan, in 1993, and the Ph.D. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1999. He is currently an Assistant Professor in the Department of Computer Science and Information Engineering, Fu Jen Catholic University, Taiwan. He was a researcher at Chunghwa Telecom Co., Ltd., Taoyuan, Taiwan from 1993 to 2001. His current research interests are fault-tolerant computing, mobile computing and networks, distributed systems, and broadband networks.

Sy-Yen Kuo (郭斯彥) received the BS (1979) in Electrical Engineering from National Taiwan University, the MS (1982) in Electrical & Computer Engineering from the University of California at Santa Barbara, and the PhD (1987) in Computer Science from the University of Illinois at Urbana-Champaign. Since 1991 he has been with National Taiwan University, where he is currently a professor and the Chairman of Department of Electrical Engineering. He spent his sabbatical year as a visiting researcher at AT&T Labs-Research, New Jersey from 1999 to 2000. He was the Chairman of the Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan from 1995 to 1998, a faculty member in the Department of Electrical and Computer Engineering at the University of Arizona from 1988 to 1991, and an engineer at Fairchild Semiconductor and Silvar-Lisco, both in California, from 1982 to 1984. In 1989, he also worked as a summer faculty fellow at Jet Propulsion Laboratory of California Institute of Technology. His current research interests include mobile computing and networks, dependable distributed systems, software reliability, and optical WDM networks.

Professor Kuo is an IEEE Fellow. He has published more than 170 papers in journals and conferences. He received the distinguished research award (1997-2001) from the National Science Council, Taiwan. He was also a recipient of the Best Paper Award in the 1996 International Symposium on Software Reliability Engineering, the Best Paper Award in the simulation and test category at the 1986 IEEE/ACM Design Automation Conference (DAC), the National Science Foundation's Research Initiation Award in 1989, and the IEEE/ACM Design Automation Scholarship in 1990 and 1991.