Fault-Tolerance Using Cache-Coherent Distributed Shared Memory Systems

D. L. Hecht, K. M. Kavi, R. K. Gaede
The University of Alabama in Huntsville
{hecht, kavi, gaede}@ece.uah.edu

C. Katsinis
Drexel University, Philadelphia, PA.
ckatsini@ece.drexel.edu

Abstract

In this paper, we describe new protocols augmenting traditional cache coherency mechanisms to implement fault-tolerance based on Recovery Blocks and checkpointing. Concurrent processes compound rollback recovery since the rollback can potentially lead to a "domino effect" whereby the process is rolled back to the beginning. Several approaches have been proposed to limit the domino effect. One set of such techniques requires communicating processes to periodically synchronize in order to checkpoint a globally consistent state. These schemes can be implemented more naturally on distributed shared memory systems using synchronization on shared memory. We have developed extensions to well known cache-coherency methods (e.g., directory-based) for the implementation of checkpointing consistent states.[1]

1. Introduction

While hardware fault-tolerance has long enjoyed considerable attention, software fault-tolerance received interest only during the 1980's, giving rise to techniques such as N-version programming, Recovery Blocks and rollback recovery. Backward rollback recovery involves the periodic checkpointing of state, which is restored when an error is detected. The backward recovery schemes can be implemented more naturally on distributed shared memory (DSM) systems using synchronization on shared memory. With suitable support for data coherency, the granularity of synchronization can range from very fine-grained (per variable) to coarse-grained (per page).

We have developed extensions to well known directory-based cache-coherency methods ([11]) for the implementation of backward recovery. No new language constructs are needed for the specification of conversations among concurrent processes ([12], [10]) because they are implied by the use of the shared memory programming model. Our technique combines directory-based protocols for maintaining cache-coherency with version numbers to track checkpoint/recovery boundaries. The granularity of synchronization for rollback purposes is a cache block, the unit for which consistency is implemented in DSM systems. On recovery, only the processes that have shared the affected block since a previous globally consistent state will rollback their computations to the previous checkpoint.

In this paper, we describe our approach using the most basic directory protocol (p+1 protocol) and illustrate the changes needed in the directory and local cache memories. We tested our method using a simulator designed specifically for this purpose. Our research should be contrasted with the research that aims to make DSMs fault tolerant ([1], [8], [14]). Our goal is to utilize DSMs to develop fault-tolerant software. There have been a few attempts to utilize DSM systems for transparent rollback and checkpointing ([2], [15]). The research that comes close to our work can be found in [6], in which nodes periodically checkpoint their shared pages. Each checkpoint is identified by a number, which is communicated to other nodes that request the pages. Receiving nodes use this checkpoint number to create "dependency tables" that track the sharing history. On an error, a globally consistent checkpoint is constructed using the dependency tables of all nodes and the rollback is then propagated to other nodes as appropriate.

These techniques do not address Recovery Block implementations, particularly when processes are structured with multiple nested Recovery Blocks. A recursive cache designed for a single node system was proposed in [12] for storing checkpoints from nested recovery layers but does not take into account recovery layers that span multiple machines.

Our approach is similar in spirit to the techniques that rely on DSMs' ability to maintain coherency for shared memory. Unlike other techniques, however, we utilize directory-based cache coherency protocols and version numbers to keep track of different Recovery Block boundaries and sharing history, for checkpointing and rollback propagation to affected processes.

[1] *Acknowledgments.* This work is supported in part by the following NSF grants: MIPS 9796310, EIA 9729889 and EIA 9895216. Partial travel support for D. Hecht was provided by the Graduate School of UAH.

2. Extending DSMs for Recovery Blocks

In this section, we describe how directory-based cache coherency techniques can be extended for use with Recovery Block based fault-tolerance. In our algorithm, each cache block is assigned a version number to reflect the Recovery Block level in which the cache block was modified. A Rollback on failure involves the rolling back of shared variables to a value with a prior version number. In general, all processors that have accessed a shared variable with a version number that corresponds to a failed Recovery Block must rollback their cache blocks to previously checkpointed data from a prior version number. No rollback is necessary for a processor that did not access variables with a failed version number.

Each Recovery Block is associated with a globally unique, non-decreasing layer number (Global Layer number) maintained by the directory. When a processor enters a new Recovery Block, it requests a new Global Layer number from the directory controller. Every processor maintains a local copy of the Global Layer number called the Local Layer number. The Local Layer number is used to mark modified variables. Each processor keeps track of the layer numbers that correspond to its individual recovery block structure.

2.1 New Cache and Directory Structures.

In directory-based coherence schemes, local caches maintain state information for each entry as SHARED (read-only) access or EXCLUSIVE (write) access. In our case, local caches must also include the Local Layer number, version numbers for each local copy of a shared cache block, and a stack of local checkpoints.

Figure 1 shows the revised cache structures used by the cache controllers. Each time the directory controller communicates with a cache controller, the current Global Layer number is included with the message and used by the cache controller to update its Local Layer number. In this presentation, we assume that each cache block corresponds to a single variable (although, in general, a cache block contains 32-128 bytes). The version number of a variable is the number of the Local Layer in which the variable was last modified. Additionally, the processor that modified the variable is stored as part of the Version/Processor to facilitate dependency tracking during rollback. If processors enter recovery blocks between modifications to a variable, the version number of the variable may change multiple times during a single period of ownership. If the version number of the variable changes, the variable is locally checkpointed before it is modified. A separate Checkpoint Stack exists for each variable in the processor's cache.

A layer stack is included in each cache to track the Recovery Layers entered by that processor. On entering a Recovery Block, a processor receives a new Global Layer number from the directory controller. The cache controller pushes the new layer number onto the layer stack and sets the Local Layer number to the Global Layer number. Exiting a Recovery Block causes the processor to pop the top layer from the stack but does not result in messages to the directory controller or in any changes to either the Local Layer or Global Layer numbers. When a processor must rollback, the number at the top of the stack identifies the number of the layer to which it must rollback (the Rollback Layer).

```
PROCESSOR: 0
LOCAL_LAYER: 13
layer stack: 12 9 6 3 0    (Val/Ver/Proc)
VAR STATE VALUE VER/PROC  CHECKPOINT_STACK
 A    S     1     5/0
 B    S     1     8/0
 C    S     0     5/2
 D    E     8    11/0       4/11/1
 E    E     8    13/0       4/10/0
 F    I    10    13/0
 G    I     0     0/-1
 H    I     0     0/-1
PROCESSOR: 2
LOCAL_LAYER: 13
layer stack: 13 10 4 1 0   (Val/Ver/Proc)
VAR STATE VALUE VER/PROC  CHECKPOINT_STACK
 A    I     8     1/2
 B    S     1     8/0
 C    S     0     5/2
 D    I     0     0/-1
 E    I     0    -1/-1
 F    I     0    -1/-1
 G    E     7     8/2       0/-1/-1
 H    I     0     0/-1
PROCESSOR: 1
LOCAL_LAYER: 13
layer stack: 7 5 2 0       (Val/Ver/Proc)
VAR STATE VALUE VER/PROC  CHECKPOINT_STACK
 A    S     1     5/0
 B    S     1     8/0
 C    S     0     5/2
 D    I     4    11/1
 E    I     4    10/0
 F    E     1    13/1        10/13/0
 G    I     0     0/-1
 H    E    31     6/1        0/-1/-1
```

Figure 1: Modified Local Cache Structures

In order for the directory controller to direct the global backward error recovery process, additional information must be added to the basic p+1 directory structure [4].

Figure 2 shows the new directory structure with values taken from a simulated environment with three processors, P0, P1, P2, and several shared variables, A through H. Items added to the directory structure include the Global Layer number, Version/Processor numbers associated with individual variables, Access Set, an Access Stack, and a Checkpoint Stack. The Version/Processor information is initialized to -1 in order to distinguish it from the values assigned at runtime. The Global Layer number identifies the current Global Recovery Layer of the program. As noted before, the version number in Version/Proc is the Recovery Block layer in which a variable is modified and the Proc is the processor that modified the variable. This information is used for rollback purposes

The directory controller uses the version number to determine which variables are affected by a failed Recovery Block. The value and version of a variable is updated when a WriteBack message is received from the owner of the variable. The WriteBack message contains the value and version assigned to the variable by the owner. The previous value and version of the variable are checkpointed before the directory entry is updated. Checkpoints are kept in the Checkpoint Stack for each variable and are shown in the figure as value/version/proc, where proc is the number of the processor that modified that version of the variable.

```
  GLOBAL_LAYER: 13
  VAR STATE VALUE VER/PROC  P0  P1  P2  ACCESS
   A    S     1     5/0      1   1   0   000
   B    S     1     8/0      1   1   1   010
   C    S     0     5/2      1   1   1   110
   D    E     4    11/1      1   0   0   000
   E    E     4    10/0      1   0   0   100
   F    E    10    13/0      0   1   0   110
   G    E     0    -1/-1     0   0   1   000
   H    E     0    -1/-1     0   1   0   000

  ACCESS STACK:
  Layer  A   B   C   D   E   F   G   H
  12     000 000 000 000 000 000 000 000
0 11     010 011 000 110 010 01  000 000
  10     000 000 000 100 100 000 000 000
  9      000 000 000 000 000 000 000 000
  8      110 100 110 000 010 100 001 000
  7      010 010 000 000 000 000 000 000
  6      000 000 000 000 000 000 000 010
  5      100 100 001 000 001 001 000 000
  4      000 000 000 010 000 000 000 000
  3      000 000 000 000 000 000 000 000
  2      000 001 011 000 000 000 000 000
  1      001 000 001 000 000 000 000 000
  0      000 000 000 000 000 000 000 000

  CHECKPOINT STACK  (Val/Ver/Proc)
  Var  Checkpoints
  A    8/1/2 0/-1/-1
  B    2/5/0 11/2/2 0/-1/-1
  C    3/2/1 13/1/2 0/-1/-1
  D    3/7/1 4/4/1 0/-1/-1
  E    1/8/1 0/-1/-1
  F    5/11/1 0/-1/-1
  G
  H
```

Figure 2: Modified Directory Structure

Note that the Copy Set only indicates which processors currently contain a copy of the cache block and it does not include processors that may have had a copy of a data item before it was either invalidated or replaced. The Access Set associated with a variable tracks all processors that access the variable in the current Recovery Block Layer (even if a processor does not currently have a copy of the variable), allowing the correct rollback to occur. Since a processor may rollback to any previous Recovery Layer, Access Sets of variables in previous Recovery Layers are saved on the Access Stack.

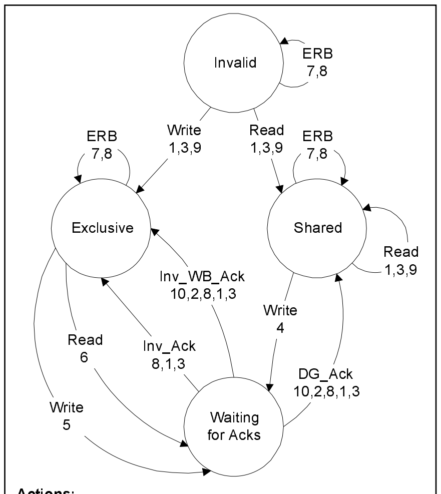
2.2 New Coherency Algorithms

State transition diagrams for the cache block entries in the directory and the local caches are shown in Figure 3 and Figure 4, respectively. The following description of the state transition diagrams emphasizes the modifications made to the basic p+1 directory protocols [4].

In each of the state diagrams, the nodes represent the states of a cache block and the edges are labeled with the events and actions associated with the state transitions. Actions are represented by numerical labels corresponding to the type of action performed. Messages received by the directory and cache controllers are shown in Table 1.

Table 1: Input Messages

Cache Controller	Directory Controller
Read	Read
Write	Write
Enter Recovery Block (ERB)	Enter Recovery Block (ERB)
Downgrade (DG)	Downgrade Acknowledge (DG_Ack)
Invalidate (Inv)	Invalidate Acknowledge (Inv_Ack)
	Invalidate-Writeback Acknowledge (Inv_WB_Ack)



Actions:
1) Update Copy Set
2) Update Variable Info (value, state, version)
3) Send reply
4) Send Invalidate
5) Send Invalidate-Writeback
6) Send Downgrade
7) Increment Global Layer
8) Save Access Set
9) Update Access Set
10) Take Checkpoint
11) Restore checkpoint

Figure 3: State Diagram for Directory Entries

Changes to the basic p+1 directory controller actions for Read and Write operations include the updating of the Access Set and the Take Checkpoint action that happens on the WriteBack message caused by an invalidation or downgrade. In addition to the traditional actions performed by the cache controller, the new actions involved in the write operation include updating the version number and taking a checkpoint of the variable (or cache block) when it is modified.

When a processor enters a new recovery block, it sends an Enter Recovery Block (ERB) request to the directory controller and waits to receive the Global Layer number to which the new recovery layer is assigned. The directory controller receives the ERB request from the processor, increments the current Global Layer number and sends the new number back to the processor. The directory controller also pushes the current Access Set onto the Access Stack and updates the current Access Set by clearing it. The last action is performed because the Access-Set indicates which processors have accessed the variable during the current recovery layer and a new recovery layer was just entered.

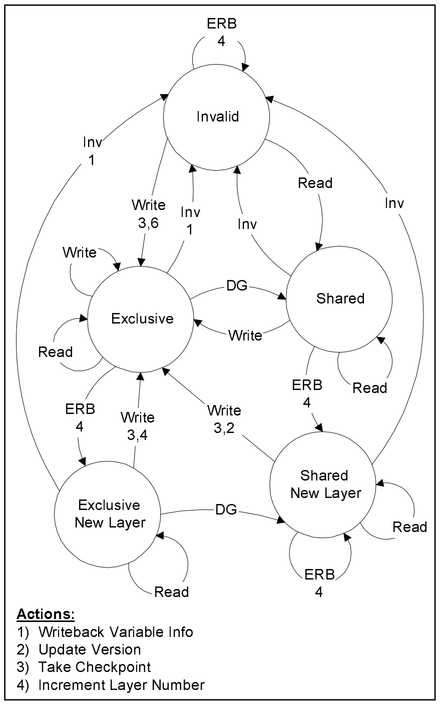


Figure 4: State Diagram for Cache Entries

After the waiting processor receives the new Global Layer number, it pushes that number onto the Layer Stack and updates the Local Layer number. When a processor exits a recovery block, it pops that block's layer number from the Layer Stack. No other actions are taken by either the processor or the directory controller when a recovery block is exited.

The major modification in the state transition diagram for cache entries is the addition of the ERB message. Changes in the version numbers of the variables and the resulting checkpoints are caused by the change in the Local Layer number that results from the ERB message. The Shared New Layer and Exclusive New Layer states indicate that the next write to the variable will result in a checkpoint and a new version for the variable.

The modifications to the state transition diagram for entries in the directory are concerned with the saving of the current Access Set and the change in the Global Layer number when an ERB message is received. The ERB message is not serviced while a shared variable is waiting for Invalidation Acknowledgments, lest an inconsistent state result.

When an acceptance test fails, the processor sends a rollback request to the directory controller with the Layer number of the failed Recovery Block (Failed Layer). The Global Backward Error recovery actions take place in two phases. In the first phase, the directory determines which processors will be required to rollback (using the Access Sets associated with the cache blocks) and notifies them. At this point, the affected processors will stop executing their programs until the Backward Error Recovery is completed. The processors send acknowledgments back to the directory controller with the layer to which they will roll back.

During the second phase, the directory controller creates a table of all Rollback Layers, which is used by the directory and cache controllers to choose the appropriate checkpoints to restore. After this phase is completed, the directory controller informs the processors that they may resume execution of their programs. It should be noted that, in our approach, there is no domino effect (or cascading of rollbacks). The checkpoint (or layer) to which a processor must rollback is determined before any rollback is effected.

For the following example, the system state before rollback (Figure 1, Figure 2) and after rollback (Figure 5) are needed. If P0 failed, it would send a Rollback Request to the directory controller with Failed Layer number = 12. When the directory controller receives the Rollback Request, it searches the directory entries for each variable with a version number greater than or equal to the Failed Layer number. In this case, F is the only variable affected by the rollback since it has a version of 13.

In order to determine which processors have accessed the affected variable, the Access Sets for all layers from the Failed Layer through the current layer (12-13) are examined. The directory controller creates a list of processors that must roll back, sends each of them a Rollback message with the Failed Layer number and then waits for their acknowledgments with the layer to which each processor will roll back (Rollback Layer). In the example, processors 0 and 1 have accessed the variable and must be notified.

After receiving a Rollback message, each processor pops all layers from its Layer Stack that are higher than the Failed Layer, leaving the Rollback Layer at the top of the stack. P0 rolls back through layer 12 and P1 rolls back through layer 7. The processor sends the directory controller its Rollback Layer and waits for the directory controller to send the list of the other processor's Rollback Layers. The directory controller creates a table of the Rollback Layers for each of the processors and sends it to the processors that must roll back. The processors use the rollback table along with version information to choose the appropriate checkpoints to restore.

```
 DIRECTORY
 GLOBAL_LAYER: 15
 VAR STATE VALUE VER/PROC  P0  P1  P2 ACCESS
  A    S     1      5/0     1   1   0   000
  B    S     1      8/0     1   1   1   000
  C    S     0      5/2     1   1   1   000
  D    E     4     11/1     1   0   0   000
  E    E     4     10/0     1   0   0   000
  F    S     0     -1/-1    0   0   0   000
  G    E     0     -1/-1    0   0   1   000
  H    E     0     -1/-1    0   1   0   000

 ACCESS STACK:
 Layer   A   B   C   D   E   F   G   H
 14     000 000 000 000 000 000 000 000
 13     000 010 110 000 100 110 000 000
 12     000 000 000 000 000 000 000 000

 CHECKPOINT STACK  (Val/Ver/Proc)
 Var  Checkpoints
 A    8/1/2 0/-1/-1
 B    2/5/0 11/2/2 0/-1/-1
 C    3/2/1 13/1/2 0/-1/-1
 D    3/7/1 4/4/1 0/-1/-1
 E    1/8/1 0/-1/-1
 F    5/11/1 0/-1/-1
 G
 H
```

```
PROCESSOR: 0
LOCAL_LAYER: 14
layer stack:14 9 6 3 0        (Val/Ver/Proc)
VAR STATE VALUE VER/PROC    CHECKPOINT_STACK
 A    S    1       5/0
 B    S    1       8/0
 C    S    0       5/2
 D    E    8       11/0         4/11/1
 E    E    4       10/0
 F    I    10      13/0
 G    I    0       0/-1
 H    I    0       0/-1
```

```
PROCESSOR: 1
LOCAL_LAYER: 15
layer stack:15 5 2 0          (Val/Ver/Proc)
VAR STATE VALUE VER/PROC    CHECKPOINT_STACK
 A    S    1       5/0
 B    S    1       8/0
 C    S    0       5/2
 D    I    4      11/1
 E    I    4      10/0
 F    I    10     13/0
 G    I    0       0/-1
 H    E    31      6/1          0/-1/-1
```

```
PROCESSOR: 2
LOCAL_LAYER: 13
layer stack: 13 10 4 1 0       (Val/Ver/Proc)
VAR STATE VALUE VER/PROC     CHECKPOINT_STACK
 A    I     8      1/2
 B    S     1      8/0
 C    S     0      5/2
 D    I     0      0/-1
 E    I     0     -1/-1
 F    I     0     -1/-1
 G    E     7      8/2          0/-1/-1
 H    I     0      0/-1
```

Figure 5: Directory and Caches after Rollback

The directory controller is responsible for providing correct checkpoint values for SHARED variables. If the processor contains a checkpoint for the EXCLUSIVE variables in its local memory, the checkpoint is restored. Otherwise, the processor sends a request to the directory controller to restore the checkpoint for the variable.

Choosing the appropriate checkpoint is tricky since each processor rolls back to a different layer. The correct checkpoint is one that reflects all of the changes that will not be repeated, but reverses the modifications from layers that will be rolled back. This explains the need for storing the modifying processor number with the version number.

For each checkpoint, the version number is compared to the Rollback Layer for the processor that modified the variable. If the version number is greater than or equal to the Rollback Layer, that particular modification is repeated after rollback and the checkpoint is not the one to use. Each checkpoint is evaluated and discarded until an acceptable one (version < rollback_layer[proc]) is found and used to update the variable .

P1 does not have the correct checkpoint for F since the checkpoint version is 13 and therefore not acceptable. P1 is forced to invalidate F and ask the directory controller to restore it. The directory controller discards the 5/11/1 checkpoint since the rollback layer for P1 was 7 (which is less than version 11). In this case, the initialization value will be used as the restore value for F.

When all required checkpoints are restored, the directory controller sends Restart messages containing a new Global Layer number to each of the processors that have rolled back. Subsequent to a recovery, new, higher, Global Layer numbers are used for the layers that must be repeated due to rollback, since Global Layer numbers are non-decreasing. From the time the directory controller receives a Rollback Request message until it sends the Restart messages, no new requests (Read, Write, Enter Recovery Block, or Rollback Request) are serviced, so that inconsistent directory states can be prevented.

3. Performance Evaluation

In this section we describe the overhead incurred by our algorithms, in terms of additional memory required, messages exchanged between the directory and cache controllers, and computational overhead.

The Access Stacks and the Checkpoint Stacks account for the majority of the memory overhead in our scheme. The number of levels in the Access Stack corresponds to the total number of Recovery Blocks entered by all of the processors. The number of Checkpoints in the Checkpoint Stack for each cache block depends on the memory access pattern of the programs. The worst case would occur when every processor modified every cache block in every layer. We feel that for a typical program, the average memory requirements would be substantially less than the worst case. Moreover, it is possible to reduce the memory required for the Access Stack and Checkpoint Stack by defining barrier synchronizations with each conversation. The synchronization points can be used to obtain a globally consistent state. We are working to develop a memory consistency model based on such barrier synchronizations.

During normal operation, the number of messages passed between the directory and processors is increased due to the Enter Recovery Block messages that are required to establish the Global Layer Number for the recovery block. Rollback messages add to the traffic during backward error recovery.

Additional ownership requests may occur after rollback since processors may have invalidated EXCLUSIVE cache blocks when local checkpoints were not available for the variables. The directory controller could include some of the variable checkpoints when servicing a write request. This increases the chance that the required checkpoint will exist on the processor when a rollback occurs; avoiding unnecessary ownership requests after the rollback completes. This approach also increases the write-reply message size and the memory required to store the checkpoints in the processor.

The computational overhead is determined by the extra work required to maintain the new components of the directory and processor cache structures and to perform the Backward Error Recovery operations in the case of a failure, namely the searches of the directory, Access Stack and Checkpoint Stacks.[2]

4. Summary And Future Research

In this paper we presented a new approach for the implementation of distributed Recovery Blocks on DSMs which provides a reduction in programming complexity over current conversation schemes. Our approach combines directory-based protocols for maintaining cache coherency with version numbers to keep track of Recovery Block levels, and judicious checkpointing of cache blocks to effectively implement backward recovery. Although we have used a centralized directory in this paper, our technique is also applicable for distributed directories. Since our approach extends data coherency techniques, they are applicable to both hardware and software DSM systems.

We are continuing to investigate ways to reduce the overhead of our approach. One possibility is to extend our approach to relaxed memory consistency models ([3][5][7]), in which case coherency is maintained only on locks. In addition, it is also possible to limit the amount of rollback caused by errors. We propose barriers (Global and Partial) with communicating processes such that all processes synchronizing on a barrier maintain consistent checkpoints when they depart from a barrier. Global barriers assure that all processes synchronize and thus receive a globally consistent state. In partial barriers, only those processors involved in a partial barrier will achieve a consistent state. When using barriers, a computation is not rolled back beyond the previous barrier. In the near future, we hope to modify an existing DSM system (either ThreadMarks [9] or Brazos [12]) so that the Recovery Block support based on our algorithms can be implemented and the efficacy of our methods can be empirically tested.

[2] Detailed analysis along with results obtained from our simulations can be found at http://crash1.eb.uah.edu/~kavi/Research/dsm.html.

5. References

[1] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. “The performance of consistent checkpointing”, Proc of the 11th Symposioum on Reliable Distributed Systems, Houston, Texas, 1992, pp39-47.

[2] M.J. Feeley, J.S. Chase, V.R. Narasayya and H.M. Levy. "Integrating coherency and recoverability in distributed systems", Proc of the first Symposium on Operating Systems Design and Implementation (OSDI), Nov. 1994, Monterey, CA, pp 215-227.

[3] K. Gharachorloo, A. Gupta and J. Hennessy. "Performance evaluation of memory consistency models for shared memory multiprocessors", Proc of Int'l. conference on architectural support for programming languages and operating systems (ASPLOS), 1991, pp 245-257.

[4] J.L. Hennessy and D.A. Patterson. "Computer Architecture: A quantitative approach", 2nd Ed. Morgan Kaufman, 1996.

[5] L. Iftode, et. al. "Scope consistency: A bridge between Release consistency and Entry consistency", Proc of the 8th Annl ACM Symposium on Parallel Algorithms and Architectures, 1996.

[6] B. Janssens and W. K. Fuchs, "Ensuring Correct Rollback Recovery in Distributed Shared Memory Systems," Journal of Parallel and Distributed Computing, vol. 29, no. 2, Sept. 1995, pp. 211-218.

[7] B, Janssens and W.K. Fuchs. "Relaxing consistency in recoverable distributed shared memory", Proc. 23rd Int'l. Symp. on Fault-Tolerant Computing, June 1993, pp 155-163.

[8] N. C. Juul and B. C. Fleish. A Memory Approach to Consistent, Reliable Distributed Shared Memory. Proc of the 5th Symp. on Hot Topics in Operation Systems, May 1995. To be published.

[9] P. Keleher, et. al. "ThreadMarks: Distributed Shared Memory on standard workstations", Proc of 1994 Winter Usenix Conference, 1994, pp 115-131.

[10] K.H. Kim. "Approaches to mechanizations of the conversation schemes baed on monitors", IEEE Trans. on Software Engr., May 1982, pp 189-197.

[11] D. Lilja. "Cache coherence in large scale shared memory mulitprocessors: Issues and comparisons", ACM Computing Surveys, Sept. 1993, pp 303-338.

[12] B. Randell. "Systems structure for software fault tolerance", IEEE Trans. on Software Engr., June 1975, pp 220-232.

[13] E. Speight. "Efficient runtime support for cluster based distributed shared memory multiprocessors", PhD Thesis, Rice University, Houston, TX, 1997.

[14] M. Stumm and S. Zhou. "Fault-tolerant distributed shared memory algorithms", Proc. Of 2nd Symp. on Parallel and Dist. Processing, Dec. 1990, Dallas, TX pp 719-724.

[15] K.L. Wu and W.K. Fuchs. "Recoverable distributed shared virtual memory", IEEE Trans. on Computers, April 1990, pp 460-469.