Recoverable Distributed Shared Memory Using the Competitive Update Protocol [*]

Jai-Hoon Kim Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX, 77843-3112
E-mail: {jhkim,vaidya}@cs.tamu.edu
Web: http://www.cs.tamu.edu/faculty/vaidya/

Abstract

In this paper, we propose a recoverable DSM that uses a competitive *update protocol. In this update protocol, multiple copies of each page may be maintained at different nodes. However, it is also possible for a page to exist in only one node, as some copies of the page may be invalidated. We propose an implementation that makes the* competitive *update protocol recoverable from a single node failure, by guaranteeing that at least two copies of each page exist.*

The paper presents preliminary evaluation of the recoverable DSM (using simulation). It is shown that the message overhead of making the DSM recoverable is small.

1 Introduction

Distributed shared memory (DSM) systems have many advantages over message passing systems [19, 17]. Since DSM provides a user a simple shared memory abstraction, the user does not have to be concerned with data movement between hosts. Many approaches have been proposed to implement distributed shared memory (e.g., [15, 5]). The DSM implementations are based on *write-invalidation* and/or *write-update*. A simple implementation of a *write-update* protocol is likely to be inefficient, as many copies of a page may be updated, even if some of them are not going to be accessed in the future. The *competitive* update protocol [11, 7] invalidates a copy of a page at some node A, if it is updated by other nodes "too many" times before node A accesses it. The proposed recoverable DSM is based on the *competitive* update protocol, and it can tolerate a single node failure without significant overhead. The proposed approach can also be used with other update protocols that selectively invalidate some copies of a page.

For future reference, note that we use the terms *node* and *processor* interchangeably. A node may execute one or more processes, however, failure of a node results in the failure of all such processes.

2 Related Work

This paper presents a recoverable DSM based on the *competitive* update protocol [11, 7]. The *competitive* update protocol defines a *limit* for each page at each node. If the number of update messages received for a page P at some node A – without an intervening access by node A – exceeds the *limit* for page P at node A, then the local copy of the page at node A is invalidated (other copies of the page are not affected).

Many recoverable DSM schemes have been presented in the literature. Some of them use stable storage (disk) to save recovery data [22, 8, 18, 9], and others use main memory for checkpointing, replicating shared memory or logging the shared memory accesses [20, 2, 4, 16, 6, 12, 21]. Proposed recoverable DSM belongs to the second category (uses main memory). [20, 21] are based on update (full-replication) protocol, while [2, 4, 16, 6, 12] are based on invalidate (read-replication) protocol.

Stumm and Zhou extended four DSM algorithms to tolerate single node failures [20]. One of their algorithms is for an update protocol. But, implementations of our algorithm is different because their algorithm is based on update protocol where all copies of a page are updated, whereas our scheme is based on the *competitive* update protocol (some copies are invalidated to reduce overhead). Additionally, our scheme supports *release* consistency.

Theel and Fleisch recently presented a coherence protocol [21] that is highly available. Their scheme has an upper bound (to reduce overhead) as well as a lower bound (for availability) on the number of copies of each shared memory page. Unlike [21], our scheme is based on the *competitive* update protocol.

Janssens and Fuchs [9] present a recoverable DSM that exploits release consistency to reduce the number of checkpoints, as compared to communication-induced checkpointing schemes for sequential consistency. Their scheme requires a process to take a checkpoint either when performing a write on a synchronization variable, or when another process performs a read on the synchronization variable. The checkpoints are stored on a storage not subject to failures. Our single fault tolerance scheme handles the non-shared data similar to [9]; it "checkpoints" non-shared data in the *volatile* memory of another processor. However, the shared data is not explicitly checkpointed – instead the shared data is duplicated as a part of the update protocol (if multiple copies already exist, no additional overhead is incurred). Janssens and Fuchs [10] also present an approach to reduce interprocessor dependencies in recoverable DSM.

Brown and Wu presented recoverable DSM, based on an *invalidate* protocol, that can tolerate single point failure [4]. A dynamic *snooper* keeps a backup copy of each page and takes over if the page owner fails. The snooper keeps

[*] This work is supported in part by the National Science Foundation under grant MIP-9502563.

track of the page contents, location of page replicas, and the identity of the page owner. The snooper can respond on behalf of a failed owner. Our scheme also maintains at least two copies of a page, however, the scheme is based on an *update* protocol, unlike [4].

Neves et al. presented a checkpoint protocol for a multithreaded distributed shared memory system based on the entry consistency memory model [16]. Their algorithm needs to maintain log of shared data accesses in the volatile memory. Fuchi and Tokoro proposed a mechanism for recoverable shared virtual memory [6]. Their scheme maintains backup process for every primary process. When the primary process sends/receives a message to/from another process (or writes/reads a shared memory), the primary process sends this information to backup process so that the backup process can log the events of the primary process.

Backward error recovery on a Cache Only Memory Architecture is implemented using invalidate protocol by Banatre et al. [2]. (A similar scheme was implemented on an Intel Paragon by Kermarrec et al. [12].) This scheme periodically takes system-wide *consistent* checkpoints. After a node fails, all nodes need to rollback to the last checkpoint.

3 *Competitive* Update Protocol [11, 7]

The basic idea of the *competitive* update protocol [11, 7] is to update those copies of a page that are expected to be used in the near future, while selectively invalidating other copies. We assume an implementation that is similar to Munin [5], with a few modifications to facilitate *competitive updates*. Each node maintains an information structure for each page resident in its memory. The information structure contains many pieces of information, as summarized below.

- *update-counter*: Counts how many times this page has been updated by other nodes, since the last local access to this page.
- *version*: Counts how many times this page has been updated since the beginning of the execution.
- *limit* L: Either set by user or transparently by the DSM protocol. The *limit* for each page determines the performance of the *competitive* update protocol.
- *last-updater*: Identity of the node that updated this page most recently. The *last-updater* is identical for all copies of a page.
- *copyset*: Set of nodes that are assumed to have a copy of this page.
- *probOwner*: Points towards the “owner" of the page [5]. On a page fault, a node requests the page from the *probOwner*. If the *probOwner* does not have a copy of the page, it *forwards* the request to its *probOwner*. The request is thus *forwarded* until a node having a copy of the page is reached.
- *back-up*: used for recoverable DSM (to be explained later).

During the execution, *update-counter* of each copy of a page is incremented on receiving an update message for the page. A copy of a page is *invalidated* when its *update-counter* becomes equal to the *limit*. Thus, the competitive update protocol invalidates those pages that are accessed “less frequently” – the protocol can be tuned to a given application by a proper choice of limit L. It is possible to choose a different limit for each copy of each page, and the limits can be changed dynamically [13, 14].

4 Recoverable Competitive Update Protocol

Recoverable scheme for a DSM, based on the *competitive* update protocol [11, 7], is relatively simple. The basic idea behind the proposed scheme is to maintain, at all times, *at least two* copies of each page (at two different nodes) in stead of checkpointing. This will allow the DSM to recover from a *single* node failure without significant overhead (provided the non-shared data is also recoverable, as discussed later).

When the competitive update protocol is used, it is possible that a page may be resident in *only one* node. Therefore, to tolerate a single node failure, it is necessary to modify the *competitive* update protocol, to ensure that *at least two* nodes have a copy of each page. Thus, there are two issues that must be dealt with to make the DSM fault tolerant (for single node failures).

1. Modification of the competitive update protocol to guarantee two copies of each shared memory page.
2. Some mechanism needs to be incorporated to make the non-shared data recoverable.

To simplify the discussion, we assume that each page has the same fixed limit L. To make the DSM recoverable, we must modify the competitive update protocol, such that some copy of the page is *not* invalidated, *even if* its update counter is equal to the *limit* L. This is achieved by designating, for each *update*, one of the nodes as the “back-up”. The copy of a page at the *back-up* node cannot be invalidated, irrespective of the value of its *update-counter*.

Maintaining the *back-up*

When a node A obtains a copy of a page P from some other node B, node B also sends identifier of the *last-updater* of page P. Node A, on receiving the page, sets its *last-updater* as well as *back-up* equal to the *last-updater* received from node B.

Contents of a *back-up* field can change in two different ways. Let us consider the copy of a page P at a node A.

1. Node A receives an update message for page P from some other node, say C: In this case, the *back-up* field at node A is set equal to C. The node C is used as *back-up* when node A updates other nodes.
2. Node A performs a *release* and sends update messages, for page P, to other nodes: When the other nodes receive these update messages, they acknowledge the update message, and send their update-counters along with the acknowledgement. Node A finds the node, say D, whose update-counter is the smallest (ties broken arbitrarily), and sets *back-up* equal to D.

Note that, for a given page, the *back-up* at different nodes may be different.

The motivation behind the above procedure is to identify a node as the *back-up* only if it has accessed the page recently (this, in turn, is motivated by the principle of locality). However, it is possible that, if the most recent access to a page is a read, the node that performed the *read* may not be identified as the *back-up* in the other nodes. This can

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Memory Access		2L	2R	2U	0L	0W	0W	0U	1L	1R	1W	1U	0L	0W	0U	0L	0W	0U	0L	0W	0U	2L	2R	2U	2L	2W	2U
Update-counter: 0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Update-counter: 1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	2	2	2	3	3	3	3	3	3	~~4~~
Update-counter: 2			0	0	0	0	0	1	1	1	1	2	2	2	~~3~~								0	0	0	0	0
Last-updater	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	2
Back-up: 0	1	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
Back-up: 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Back-up: 2			0	0	0	0	0	0	0	0	0	1	1	1									0	0	0	0	0

Figure 1: Update Counter for Recoverable DSM

happen because a read can be performed locally without other nodes knowing about it.

The modified (recoverable) competitive update protocol

The proposed scheme assumes that programs are *data-race-free*[1]. The modified protocol is essentially identical to the original competitive update protocol with one difference: A node that is designated as the *back-up* for an update does *not* invalidate the local copy of the page *even if* the *update-counter* becomes equal to limit L or exceeds L. (Update message sent to the *back-up* node is tagged by a special *marker*.) Any other node, whose update-counter is $\geq L$ invalidates its local copy of the page. This procedure ensures that, at any time, at least two copies of a page are in existence.

The *back-up* for an update is always a node that has accessed the page in the recent past. Therefore, from the locality principle, this node is likely to access the page in the near future as well. The modified update protocol forces this node to retain a copy of the page. This protocol may be viewed as incorporating a "pre-fetch" mechanism. As the page copy is likely to be used in the near future, the overhead of updating the copy is often compensated by a reduction in the number of page faults.

Note that "cost" (e.g., number of messages) of the recoverable protocol can be larger than that of the non-recoverable protocol, only when the non-recoverable protocol would result in a page having only one copy. Whenever, the non-recoverable protocol results in multiple copies of a page, the recoverable protocol does not result in any additional cost. Thus, the *difference* between the costs of the recoverable and non-recoverable protocols is greatest when limit is 1, and reduces as *limit* becomes larger.

Figure 1 illustrates how the *back-up* is maintained. For this example, assume that the *limit* L is 3. The system is assumed to contain three nodes, 0, 1 and 2. In the figure, iL and iU denote *acquire* and *release* operations.[1] Also, iR and iW denote *read* and *write* operations performed on this page by node i. Initially, the page is loaded in the local memory of two nodes (0 and 1 in our example), and one of them (node 0) is considered to be the *last-updater*. *Back-up* at nodes 0 and 1 is initialized to 1 and 0, respectively. The *memory access* row in Figure 1 presents a total ordering on the accesses to the page under consideration. The next three rows present values of the update-counters at the three nodes at various times, e.g., the *update-counter:0* row corresponds to node 0. (The values in column i correspond to the update-counters *after* the memory access in column i is performed.) The next row of the table lists the *last-updater* variable at each node (it is identical at all nodes). The last three rows list the value of the *back-up* variable for the page at each node. Note that *last-updater* and *back-up* change only when a *release* is performed, whereas, *update-counter* at a node A changes when either (i) node A performs a local access to the node, or (ii) another node performs an update to the page. A "blank" in the table implies that the corresponding node does not have a copy of the page at that time, and the X in the figure denotes an invalidation.

4.1 Duplicate Copy for Non-Shared Data

When it is necessary to ensure that the faulty node can be recovered, the non-shared data [2] must also be duplicated. When a node writes shared data and updates other copies of the data, the non-shared data at the node can be sent, along with the update message, to any one node. Although no additional messages are required, the size of one of the messages will be larger. The amount of non-shared data transferred can be reduced by sending only the *modifications* to the local data since the most recent update performed by the node. This *incremental* approach makes recovery more complicated, as the non-shared data of a node can be scattered at various nodes in the system.

An alternative is to specify for each node (say A), another node (say B) to which the incremental changes in the *non-shared* data are sent, when node A performs an update to some *shared* data. While this will simplify recovery, it may increase the number of messages.

4.2 Recovery

The proposed DSM system is recoverable from all single node failures (fail-stop) because all shared memory pages and non-shared memory pages (if necessary) have at least two copies. The recovery is straightforward. After a single node failure, the shared memory remains available. If the faulty node is to be recovered, then its non-shared data is obtained from other nodes (the non-shared data is duplicated, as described above). Two issues need further elaboration.

- Since failure can occurs at any time, contents of the copies of the same page may be different (if the failure occurs while an update is in progress). In this case, some copies are out-of-date. This problem can be resolved by searching the most up-to-date copy – to facilitate this, a *version* number is attached to each page to count the number of updates performed to the page from the beginning of execution. The copy with the largest version number is the most up-to-date copy (this is similar to [20]). If a node fails after it has written to a page, but before it has performed a *release* then the modifications made by the node are lost when the node fails. This is acceptable, as the system state will still be consistent after the failure. However, if a node fails after the node sent update messages only for the part of pages to be updated on a *release*, the node may not restart from the previous consistent state because old *version* of the updated pages may not exist. This problem can be solved by new mechanisms.

[1] Although we obtained the notation *iL* and *iU* by abbreviating *i-Lock* and *i-Unlock*, it should be noted that *acquire* and *release* operations in release consistency are not necessarily equivalent to *lock* and *unlock*.

[2] The *non-shared* data includes process status, e.g., contents of stack and registers.

One possibility is for the updating node to send all updates in one message to another node which will send update messages to other nodes. By this *indirect* update mechanism, all pages can be updated atomically in spite of a single node failure. Another mechanism, *lazy-decoding*, postpones decoding of update message for a page until the first access of the data to be modified by the decoding. *Lazy-decoding* allows every page being updated on a *release* to have at least one copy of old *version* of the page until all updates finish on the *release*. (The updates are sent immediately on a release, but not decoded immediately.)

- It is necessary to ensure that, after recovery, each shared memory page has at least two copies. Therefore, after failure, if only one node has a copy of a page, then another copy is created on any other node. Now we assume that two copies of each page exist. The recovery algorithm must also ensure that all the *last-updater* and *back-up* fields are correct. We now illustrate how this can be achieved. Consider a page P. Two cases are possible.

  (a) If the *last-updater* for page P fails, then any other node having the page is designated as the *last-updater*, and its update-counter is cleared to 0. All relevant nodes are informed of the new *last-updater*. These nodes set their *last-updater* as well as the *back-up* fields to point to the new *last-updater*. The new *last-updater* sets its *back-up* field to point to any other node that has a copy of the page.

  (b) If some node other then the *last-updater* is faulty, then it is possible that the *back-up* field at the last-updater may be pointing to the faulty node. It is only necessary to set the back-up to point to any other node that has a copy of the page.

5 Performance Evaluation

5.1 Methodology

We measured overhead for maintaining recoverable *shared* data [3] by comparing the "cost" for non-recoverable protocol and recoverable protocol. The "cost" metrics used here are (i) number of messages, (ii) amount of information transferred between the nodes, and (iii) number of page faults. These results are very preliminary and a more complete evaluation of the proposed scheme will be performed by means of an implementation.

As a preliminary test, we generated synthetic trace data by using an event generator. (Possible events are read, write, acquire, and release.) The event generator can produce synthetic trace data according to the memory access behavior which we can define as input. We also modified the Proteus [3], execution-driven multiprocessor system simulator, to produce trace data. The modified Proteus produces trace data for shared memory operations, read, write, acquire, and release. The trace data are used as input for our simulator which computed the cost (the number of page faults, the number of messages, and the amount of data transferred). We assume that the DSM system consists of 16 nodes, and that the page size is 1024 bytes.

For the simulation, we assume an implementation similar to Munin, i.e., based on the dynamic distributed ownership mechanism [5].

5.2 Cost Measurement

On a page fault, the number of messages required varies because of the dynamic distributed ownership mechanism. The page request is forwarded along the *probOwner* link until a node that has a copy of the page is reached (when $L > 1$), or till the page "owner" is reached (when $L = 1$). We assume that an *acquire* and *release* are implemented as special procedures using a message passing library – an *acquire* is assumed to require three messages. On a *release*, two messages are required per copy of the modified pages – one for sending a request and the other for acknowledgment. Some application programs traced using Proteus use semaphores to achieve synchronization – we appropriately interpreted these as *acquire*s and *release*s.

Message size for an update at a *release* is proportional to on the number of *write*s performed since the recent *acquire*. Other short messages (e.g., acknowledgement) are assumed to be 8 bytes.

5.3 Results

Figures 2, 3, and 4 show the result of our experiments with synthetic trace data. We assumed distributed shared memory system of 16 nodes. 10,000 memory accesses (each access reads or writes 8 bytes) for a single page of 1024 bytes are simulated for three different read ratios (70%, 80%, and 90% uniform distribution) to compute overhead for the recoverable scheme. The results of the simulation converge by 10,000 accesses. In the figures, "non-recoverable:x%" means the cost for the competitive update protocol *without* recoverable scheme at x% read ratio and "recoverable:x%" means the cost for the *recoverable* competitive update protocol at x% read ratio.

Figure 2 shows that the number of page faults decreases as the update limit (L) increases because the number of page copies increases by allowing more updates. Observe that the number of page faults of recoverable scheme is less than that of non-recoverable scheme for low limit (L). Maintaining at least two copies for the recoverable scheme causes page "pre-fetch" effect which reduces the number of page faults.

Figure 3 shows that the number of messages increases, especially at high limit (L), as read ratio decreases. At low limit (L), the number of messages for the recoverable scheme is greater than that of non-recoverable scheme, because recoverable scheme needs extra messages to maintain at least two copies. However, note that the increase in the number of messages is not too large.

Figure 4 shows that small amount of data is transferred between the nodes, per memory access, at high limit (L) and/or high read ratio. (A page fault needs to copy the whole page, whereas, the amount of data transfer needed due to updates is small for the synthetic traces.) For small L, the recoverable scheme requires less data transfer, as it reduces the page fault rate.

[3] We did not measure the overhead (the number of messages and the amount of data) for non-shared data. However, no additional messages are required for non-shared data if the non-shared data is sent along with an update message for shared data. As for the amount of data, we believe that the amount of data transferred due to non-shared data is relatively small as compared to shared data, in many scientific applications.

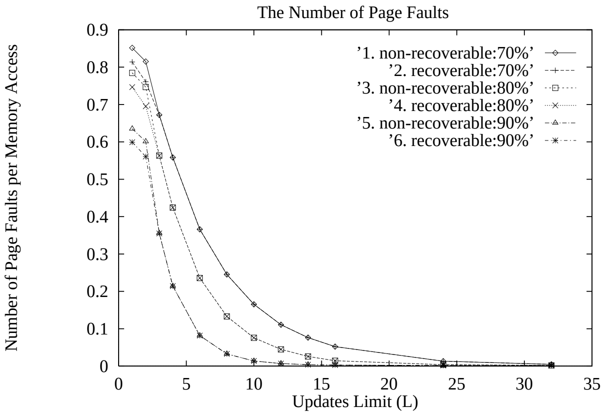

Figure 2: The number of page faults (synthetic trace)

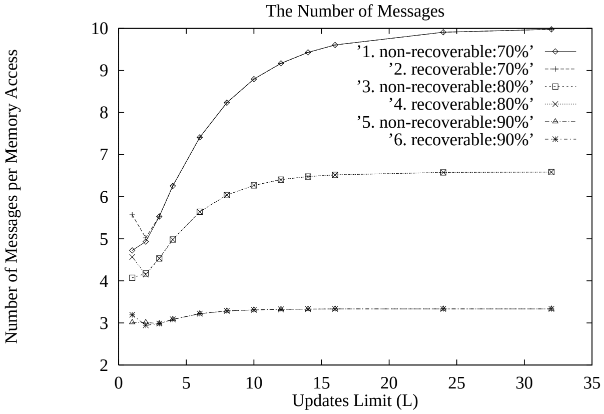

Figure 3: The number of messages (synthetic trace)

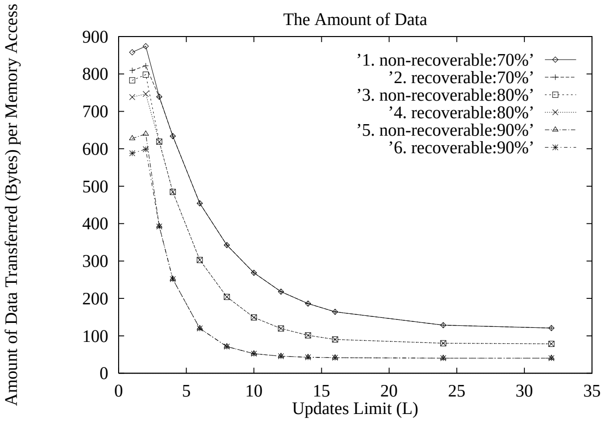

Figure 4: The amount of data (synthetic trace)

Four application programs (MP3D, Floyd-Warshall, FFT, and Gauss-Jacobi) were used to evaluate the overhead of recoverable *competitive* update protocol. These applications are simulated on the modified Proteus to produce trace data, and our protocol simulator is executed with the trace data to evaluate the overhead. Due to lack of space, we present the results for only one application in Figure 5 (Note that Figure refmp3d plots total cost over the entire application) – the other applications yield similar results [13]. Observe that, in most cases, the recoverable scheme has a comparable or smaller "cost" than the non-recoverable protocol.

As noted previously, the difference in the "cost" of the recoverable protocol and the non-recoverable protocol is likely to be the greatest when the *limit*, is small. The simulation results suggest that the cost of the recoverable scheme is comparable or smaller than that of the non-recoverable scheme, for all values of the *limit*.

6 Conclusion and Future Work

This paper presented a scheme to implement a software DSM that is recoverable in the presence of a single node failure. Our scheme differs from the previous work in that the proposed scheme is based on the *competitive update* protocol, which combines the advantages of invalidate as well as traditional update protocols. In addition, our approach is integrated with the release consistency model for maintaining memory consistency. In the basic competitive update protocol, the number of copies of a page varies dynamically – in the extreme, only one node may have a copy of the page or all nodes may have a copy of the page. Our approach is based on the simple observation that, to make the DSM recoverable from a single failure, it is adequate to ensure that each page has at least two copies at all times. To achieve this we suggest a modification to the basic *competitive update* protocol. Recovery is simple because an active back-up copy exists for each page. The proposed scheme is applicable to other updated-based protocols that incorporate mechanisms to selectively invalidate some pages. It is also applicable to generalizations of the *competitive* update protocols where the *limit* may be different for each page, and vary with time [13, 14].

Preliminary performance evaluation results indicate that the proposed scheme does not significantly increase the number or size of messages required by an application. Further analysis is necessary to fully evaluate the proposed scheme. The proposed recoverable DSM scheme is presently being implemented on a network of workstations.

Acknowledgements: We thank the referees for their detailed comments.

References

[1] S. V. Adve, *Designing Memory Consistency Models for Shared-Memory Multiprocessors*, PhD thesis, University of Wisconsin-Madison, Dec. 1993.

[2] M. Banatre, A. Gefflaut, and C. Morin, "Tolerating node failures in cache only memory architectures," Tech. Rep. 853, INRIA, 1994.

[3] E. Brewer and C. Dellarocas, *Proteus User Doc.*, 1992.

[4] L. Brown and J. Wu, "Dynamic snooping in a fault-tolerant distributed shared memory," in *Symposium on Distributed Computing Systems*, pp. 218–226, 1994.

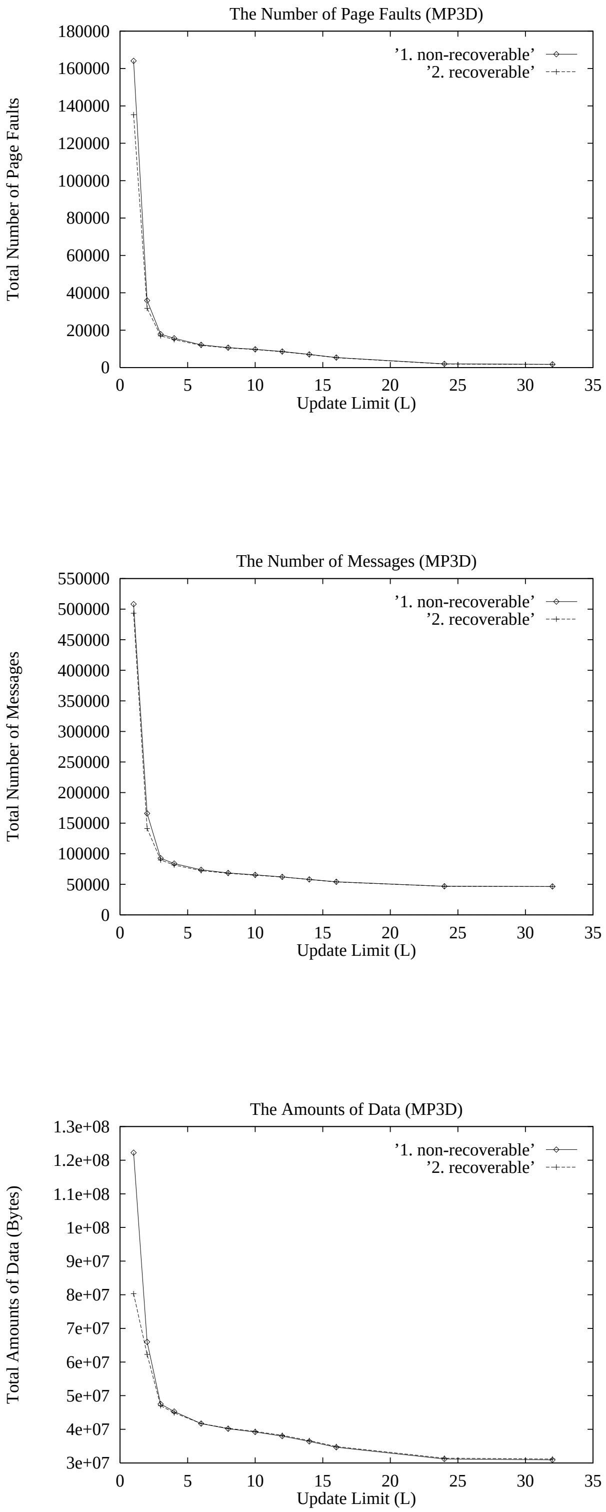

Figure 5: Overhead for Recoverable Scheme (MP3D)

[5] J. B. Carter, *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*. PhD thesis, Rice University, Sept. 1993.

[6] T. Fuchi and M. Tokoro, "A mechanism for recoverable shared virtual memory," 1994.

[7] H. Grahn, P. Stenstrom, and M. Dubois, "Implementation and evaluation of update-based cache protocols under relaxed memory consistency models," *Future Generation Computer Systems*, vol. 11, pp. 247–271, June 1995.

[8] G. Janakiraman and Y. Tamir, "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputer," in *13th Symp. on Rel. Distr. Syst.*, 1994.

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

[10] B. Janssens and W. K. Fuchs, "Reducing interprocessor dependence in recoverable distributed shared memory," in *13th Symposium on reliable Distributed Systems*, Oct. 1994.

[11] A. Karlin et al., "Competitive snoopy caching," in *Proc. of the 27'th Annual Symposium on Foundations of Computer Science*, pp. 244–254, 1986.

[12] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A recoverable distributed shared memory integrating coherence and recoverability," in *Proc. 25th Int. Symp. on Fault-Tolerant Computing*, pp. 289–298, 1995.

[13] J.-H. Kim and N. H. Vaidya, "Distributed shared memory: Recoverable and non-recoverable limited update protocols," Tech. Rep. 95-025, Texas A&M Univ., College Stn., 1995.

[14] J.-H. Kim and N. H. Vaidya, "Towards an adaptive distributed shared memory," Tech. Rep. 95-037, Texas A&M University, College Station, 1995.

[15] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, Nov. 1989.

[16] N. Neves, M. Castro, and P. Guedes, "A checkpoint protocol for an entry consistent shared memory system," in *Symp. on Principles of Distr. Comp.*, pp. 121–129, Aug. 1994.

[17] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *IEEE Computer*, vol. 24, pp. 52–60, Aug. 1991.

[18] G. Richard and M. Singhal, "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory," in *12th Symposium on Reliable Distributed Systems*, 1993.

[19] M. Stumm and S. Zhou, "Algorithms implementing distributed shared memory," *IEEE Computer*, pp. 54–64, May 1990.

[20] M. Stumm and S. Zhou, "Fault tolerant distributed shared memory algorithms," in *Int. Conf. on Parallel and Distr. Processing*, pp. 719–724, 1990.

[21] O. Theel and B. Fleisch, "Design and analysis of highly available and scalable coherence protocols for distributed shared memory systems using stochastic modeling," in *Int. Conf. on Parallel Procesing*, vol. I, Aug. 1995.

[22] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory: Memory coherence and storage structures," in *Int. Symp. on Fault-Tolerant Comp.*, pp. 520–527, 1989.