Distributed Shared Memory: Recoverable and Non-recoverable *Limited* Update Protocols

Jai-Hoon Kim   Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
Phone: (409) 847-8609
FAX: (409) 847-8578
E-mail: jhkim@cs.tamu.edu

Technical Report 95-025

May 1995

Abstract

In the recent years, many protocols for implementing Distributed shared Memory (DSM) have been proposed. The implementations can be broadly divided into two classes: invalidation-based schemes and update-based schemes. Many approaches have also been proposed to make the DSM system recoverable from a failure. However, much of this work is restricted to reliable DSM based on invalidation.

In this report, we propose a reliable DSM that uses a *limited* update protocol and the *release* consistency model. 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 *limited* update protocol recoverable from a single node failure, by guaranteeing that at least two copies of each page exist. The proposed recoverable DSM incorporates the *release* consistency model.

The report presents preliminary evaluation of the recoverable DSM (using trace-driven simulation). It is shown that the overhead of making the DSM recoverable is small.

Index Terms: **distributed shared memory, fault-tolerance, update protocol, release consistency**.

1 Introduction

Distributed shared memory (DSM) systems have many advantages over message passing systems [19, 16]. Since DSM provides a user a simple shared memory abstraction, the user does not have to be concerned with data movement between hosts. Many applications programmed for a multiprocessor system with shared memory can be executed in DSM without significant modifications.

Many approaches have been proposed to implement distributed shared memory [6, 8, 14, 19]. The DSM implementations are based on *write-invalidation* and/or *write-update*. In the past, *write-update* protocols were often inefficient due to the overhead of updating multiple copies of a page. However, the recent implementations of DSM use relaxed memory consistency models such as *release* consistency [6]. For such implementations, *write-update* protocols are often superior than *write-invalidate* protocols.

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. In this report, we present a reliable DSM based on an approach that 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. This protocol is called the *limited* update protocol, and is based on a similar protocol for cache-coherence in hardware shared memory systems [1, 7]. The proposed reliable DSM can tolerate a single node failure without significant recovery overhead. Also, the proposed scheme incorporates the *release* consistency model.

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.

This report is organized as follows. Section 2 discusses *release* consistency and summarizes the *limited* update protocol. The proposed recoverable DSM scheme is presented in Section 3. Performance evaluation of the proposed scheme is presented in Section 4. Related work is discussed in Section 5. Section 6 concludes the report.

2 *Limited* Update Protocol

In a protocol that performs *write-update*, when a node accesses a page for the first time, a copy of the page is brought into the local memory of the node. This copy of the page is updated whenever another node modifies the page. (In contrast, in protocols based on *write-invalidate*, whenever a remote node modifies a page, the local copy is invalidated.)

A disadvantage of the update protocol is that, over the course of the execution, many nodes may obtain a copy of the page in their local memory. Whenever any node modifies the page, an update message must be sent to all these nodes, incurring significant overhead. Two approaches have been used to mitigate this overhead. First, a relaxed consistency model such as *release* consistency is used in recent implementations. Second, some copies of a page are invalidated if they are not likely to be used in the future (some heuristic may be used to determine which copies can be invalidated). Now, we summarize each of these approaches.

Release Consistency

Here we present only a brief overview of *release* consistency, as necessary to describe the proposed reliable DSM scheme. The *release* consistency protocol is based on the observation that, in a typical program, accesses to shared variables are separated by *synchronization* operations – in release consistency [6], these operations are termed *acquire* and *release.* If an access by a process to some shared data is likely to cause a race condition, then the process first performs an *acquire* operation. When the process has completed its accesses to the shared data, it performs a *release* operation. If one process has already performed an *acquire*, another process' *acquire* will block until the first process performs a *release.* This ensures that while one process is modifying some shared data, another process will not attempt to access the data. Implementations of *release* consistency can take advantage of this observation to improve performance, as follows. Consider a process on node A that has performed an *acquire*, subsequently performed multiple writes to shared data, and is now performing a *release* operation. Because of release consistency, it is adequate if node A sends a *single* update message (to all nodes that have a copy of the modified pages) corresponding to *all* the writes performed by the process since its most recent *acquire.* In implementations that use sequential consistency (instead of release consistency), it is necessary to send one update message for every *write* performed by node A. Due to release consistency, it is necessary to perform at most one update for every *release* performed by a process. This implementation of release consistency reduces the number of messages, thereby improving performance. Note that the *release* operation blocks until the updates are propagated to all relevant nodes and acknowledgment are received from them.

Invalidation

In an update protocol, when a node accesses a page, a copy of the page is created in the local memory. This copy of the page must be updated (by means of an update message) whenever another node modifies the page. It is conceivable that some node A may access a page infrequently – in this case, it is advantageous to invalidate the page's copy at node A (as compared to updating it whenever some other node modifies it). Carter et al. [6] suggest a time-out protocol to determine when a page copy should be invalidated – essentially, the local copy of a page at node A is invalidated if it is not accessed by node A for a significant duration of time. As described below, we consider a different approach, called *limited* update, to determine when a page should be invalidated. The reliable DSM proposed in this report is based on the limited update protocol.

Limited Update Protocol

The *limited* update protocol presented here is intended for a software implementation of DSM that uses *release* consistency (summarized above). This protocol is similar to two protocols previously proposed for maintaining sequential consistency in hardware caches [1, 7]. The advantage of this protocol is that it facilitates a simple implementation of a recoverable DSM.

The basic idea of the *limited* protocol is to update those copies of a page that are expected to be used in the near future, while selectively invalidating other copies. Now we summarize the limited update protocol.

Information Structure

We assume an implementation that is similar to Munin [6], with a few modifications to facilitate *limited 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. When a page is brought into the local memory, the counter is initialized to 0. Also, when a local process accesses this page, the counter is cleared to 0. The counter is incremented at every remote update of the 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 *limited* 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. (*last-updater* is the originator of the most recent update message for the page).

- *copyset*: Set of nodes that are assumed to have a copy of this page. The *copyset* at different nodes, that have a copy of the same page, may be different. In general, a node may not know exactly which other nodes have a copy of the page [6]. However, when a node performs an update (when it does a *release*), at the end of the update protocol, that node knows precisely the set of nodes, that hold the copies of the page, that were updated. The "updater" node collects this information by means of acknowledgment messages sent as a part of the update protocol. It is a simple matter to modify the update protocol to obtain the value of the *update-counter* for the modified page, from each node that received the update. This observation will be used in the recoverable DSM.

- *probOwner*: Points towards the "owner" of the page [6].

- *back-up*: To be explained later.

Note that each node maintains the above structure for *each* page in its local memory. In the following, when we use a phrase such as "the update-counter at node A", we are referring to "the update-counter for the page under consideration at node A". It is implicit that the "update-counter" (or some other information) pertains to a specific page.

Use of the Update-counter

During the execution, *update-counter* of each copy of a page changes dynamically, as explained below. A copy of a page is *invalidated* whenever its *update-counter* becomes equal to the *limit*.

When a node, say A, receives from another node a copy of a page, say P, its update-counter is initialized to 0. The update-counter is incremented whenever node A receives an update message, for page P, from any other node. If a process on node A accesses page P, then the update-counter for page P at node A is cleared to 0. If, at any time, the update counter for page P at node A becomes equal to the *limit* L, then node A invalidates its copy of page P. Thus, page P on node A is invalidated *only when* L updates to the page, by other nodes, occur without an intervening access to the page by a process on node A. Thus, the limited 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. As discussed later, the limit can also be changed dynamically.

Update-counter is analogous to *timeout* mechanism in Munin [6]. Our scheme bounds the update overhead by using a *limit* L on the number of updates without an intervening local access, while *timeout* approach uses the *freeze time* mechanism. Two mechanisms use similar concept: copy of a page is invalidated if it is expected *not* to be used in the near future (by evaluating the cost for updates). It is reasonable that the cost is evaluated by the number of updates, rather than time, between two sequential updates without an intervening local access. Consider two scenarios as shown in Figure 1: (i) a copy is updated by other nodes many times but each update occurs after an interval greater than *freeze time*; (ii) a copy is updated by other nodes only twice within a *freeze time*. With the timeout mechanism, the copy will be invalidated in case (ii), while updated in case (i). With the limited update protocol, with *limit* $L = 3$, the copy will be invalidated in (i), and updated in (ii). Therefore, although the two approaches (*limit* and *timeout*) have similar goals, they do not behave identically.

Example

Figure 2 illustrates how the limited update protocol works, by focusing on a single page in the DSM. For this example, let us assume that the *limit* L associated with this page is fixed at 3. In the figure, iL and iU denote *acquire* and *release* operations.[1] Also, iR and iW denote *read* and *write* operations

[1] Although we obtained the notation *iL* and *iW* by abbreviating *i-Lock* and *i-Unlock*, it should be noted that *acquire*

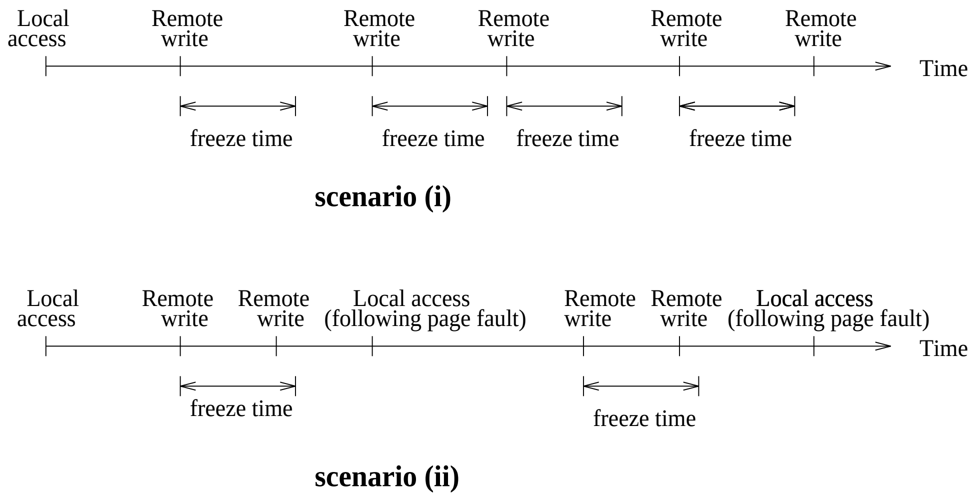


Figure 1: Timeout Mechanism

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

Figure 2: Update-counter

performed on this page by node i. The second row of the table presents a total ordering on the memory accesses performed by nodes 0, 1 and 2 on the page under consideration. The last three rows list the values of the update-counters at the three nodes at various times, e.g., the *update-counter:0* row corresponds to node 0. A "blank" in the table implies that the corresponding node does not have a copy of the page at that time.

Initially, only node 0 has a copy of the page whose update-counter is 0 (column 0 in the table). Next, node 1 performs an *acquire* and tries to *read* the page (columns 1-2 in the figure). As it does not have a copy of the page yet, it receives a copy from node 0, and the update-counter for this copy is set to 0 (column 2 in the figure). Node 1 then performs a *release* (column 3). As node 1 did not perform any *write*s, no updates are needed at the *release*. Next, node 2 also performs *acquire-read-release* (columns 4-6). Again, a copy of the page is brought to the local memory of node 2, and its update-counter is set to 0 (column 5).

Next, node 0 performs an *acquire* and performs two *write* accesses to the page, and then performs a *release*. As node 0 has performed a local access to the page, its update-counter is cleared[2] to 0 (in this case, of course, it was already 0). In the software implementation of *release* consistency protocol, the updates are not propagated to other nodes until node 0 performs a *release*. Therefore, the update-counters at nodes 1 and 2 are unchanged in columns 7, 8 and 9. When node 0 performs a *release*, the updates by node 0 are propagated to nodes 1 and 2. When the update message is received, nodes 1 and 2 incorporate the updates, and increment the respective update-counters by 1 (column 10). Next, node 1 performs an *acquire-read-write-release* sequence (columns 11-14). As node 1 has performed a local access, its update-counter is cleared to 0 (column 12). When node 1 performs the *release*, the updates are propagated to nodes 0 and 2, and their update-counters are incremented to 1 and 2, respectively (column 14). Next, node 0 performs a *acquire-write-release* sequence (columns 15-17). The update-counter of node 0 is cleared because of the local access (column 16). The update-counters at nodes 1 and 2 are incremented to 1 and 3, respectively, when the update is received from node 0 (column 17). Now, because the update-counter at node 2 has become equal to 3, the *limit* for the page, the local copy of the page at node 2 is *invalidated*. (The X in the figure denotes an invalidation.) Subsequently, node 0 performs two *acquire-write-release* sequences, at the end of which, the update-counter at node 1 becomes 3 (column 23). Therefore, the page copy at node 1 is also invalidated. At the end, only node 0 has a copy of the page, with update-counter 0 (column 23). □

Observe that, in the above protocol, *at least one* node must have a copy of the page, namely the

and *release* operations in release consistency are not necessarily equivalent to *lock* and *unlock*.

[2] Actually, in practice, only if the update-counter is non-zero, any action need be taken. This can be implemented as follows. When the update-counter for a page becomes non-zero, the page is read/write-protected, so that any local access the page will result in a page fault. On such a page fault, the fault handler will set the update-counter to 0, and remove the read/write-protection. Thus, any further local access to the page can proceed without any performance penalty.

node that wrote to the page most recently. The update-counter at this node will be equal to 0, and therefore this copy of the page will not be invalidated. However, it is possible that a page may exist at *only one* node, as illustrated in the above example (column 23). This can happen when a single node performs consecutive L updates to the page (where L is the *limit*), without an intervening access by some other node. If no node performs consecutive L updates to the page, then at least two nodes must have a copy of the page (namely, the two most recent "updaters" of the page).

Generalization of the Limited Update Protocol

Unlike other similar schemes [1, 7], implemented in hardware caches (with sequential consistency), the software implementation can be more flexible. Note that the limited update protocol is designed for the *release* consistency model, unlike [1, 7]. The above protocol can be generalized in four ways, as summarized below:

- **Multiple Consistency Protocol:** Multiple consistency protocol was presented by Carter [6], which can perform efficiently by using the appropriate protocol for each data object of different access pattern, The generalized limited update approach can provide a different protocol for each page by adjusting the *limit* parameter independently for each page. By setting the *limit* to 1, the limited update protocol becomes equivalent to invalidate protocol, while the protocol is equivalent to the traditional update protocol when "limit" is infinity (or large). Intermediate values of "limit" will interpolate between invalidate and update protocols. Thus, the generalized limited update protocol can effectively be used as a "multiple" consistency protocol, simply by using a different *limit* for each page.

- **Hybrid Protocol:** Hybrid protocol is more appropriate than a "pure" protocol for a DSM, if the memory access pattern to the same page is different in each node. TOP-1 [17], a tightly coupled snoop-cache-based multiprocessor, has a hybrid coherence protocol which allows an update protocol and an invalidate protocol, which can be dynamically changed, to coexist simultaneously. However, TOP-1 needs additional hardware design, cache mode register (to specify a cache mode: update mode and invalidate mode) and CH (Cache Hit) bus line (to indicate a snoop hit). Hybrid protocol is also implemented in Munin [6] by using the *timeout* mechanism. When a node writes and its update is propagated to other nodes, default copies are updated, while some copies are self-invalidated if the copies are not locally accessed for more than the *freeze time* since the last update. However, the limit update protocol can be extended to hybrid protocol without any additional hardware or timeout mechanism. By allowing each node to use a *different* limit for the local copy of the *same* page, the limit update protocol can become a hybrid protocol in which an update protocol, an invalidate protocol, and limited update protocol coexist for the *same* page.

- **Dynamic Protocol:** It is possible to change the *limit* associated with each page dynamically. This feature is useful when the pattern of accesses to a page changes with time. This feature is also useful when, initially, the "optimal" value of the *limit* is unknown. Some heuristic can be used to adapt the protocol to dynamically determine the appropriate limit, so as to minimize the overhead.

- **System-Dependent Protocol:** Instead of incrementing the limit by 1 each time an update occurs, we allow the limit to be incremented by a different amount. When an update message is received by a node, the amount of increment in the update-counter can be made a function of the node from which the update message is received. When the "cost" of an update is dependent on the identity of the sender node, this feature is useful to tune the limited update protocol to the underlying hardware architecture.

3 Recoverable Limited Update Protocol

In this section, we present a recoverable DSM system based on the *limited* update protocol described above. Section 5 compares the proposed scheme with other recoverable DSM schemes.

Recoverable scheme for a DSM, based on the limited update based protocol, 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). This will allow the DSM to recover from a *single* node failure without a rollback (provided the non-shared data is also recoverable, as discussed later).

As discussed previously, when the limited 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 *limited* 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 limited 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.

We first focus on the first of the above two issues.

Maintaining at least two copies of each page

To simplify the discussion, we assume that each page has the same fixed limit L. The proposed scheme can be readily extended to allow all the generalizations of the limited update protocol.

The limited update protocol needs to be modified to ensure that at least one additional copy of the page exists, in addition to that present at the most recent updater of the page. As pointed out earlier, in the original limited update protocol, if a single node, say A, performs consecutive L updates to a page, without an intervening access by some other node, all copies of the page, except that at node A, will be invalidated. Thus, to make the DSM recoverable, we must modify the limited 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. Note that the *back-up* is specified for each *update*, and may change from one update to the next update of the *same* page. The performance of the recoverable DSM depends on the choice of the back-up – in our approach, as described below, the node chosen as the back-up is the one that is expected to access the page in the near future.

Each node maintains an information structure, described earlier, for each page in its local memory. One of the fields in this information structure is *back-up*. Consider a node A that performs a *release* and sends update messages for a page P as a part of the *release*. Let the identifier stored in the *back-up* field in the information structure corresponding to page P at node A be B. Then, when the update message from node A is sent to node B, the message is tagged by a special "marker" flag – the marker flag informs node B that it cannot invalidate its copy of the page (even if the update-counter becomes large).

Maintaining the *back-up*

Initially, a page is loaded in any two nodes, say X and Y, and one of them, say X, is considered to be the last-updater. The update-counter at both nodes is initialized to 0, the *back-up* field at node X is set to Y, and vice-versa.

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.

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 happen because a read can be performed locally without other nodes knowing about it. There are two aspects to this issue: (i) First, it is adequate if the node identified as the *back-up* has accessed the page *recently*, not necessarily *most* recently. (ii) Secondly, reads that cause a page fault can often be used to modify the *back-up*.

Note that, the above procedure for maintaining the *back-up* works correctly with all the generalizations of the *limited* update protocol, described in Section 2.

The modified (recoverable) limited update protocol

The modified protocol is essentially identical to the original limited 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. (As explained before, 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 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.

Example

Figure 3 illustrates how the *back-up* is maintained. We would like to caution that the example is somewhat long. (The format for the table is similar to Figure 2.) For this example, assume that the *limit* L is 3. The system is assumed to contain three nodes, 0, 1 and 2. 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 3 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. (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

	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	$\cancel{4}$
Update-counter: 2			0	0	0	0	0	1	1	1	1	2	2	2	$\cancel{3}$								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 3: Update Counter for Recoverable DSM

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. (Again, the values in column i denote the back-ups *after* the memory access in column i is performed.) For future reference, 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.

The initial state is illustrated in column 0 of the table. The first *acquire* is performed by node 2, followed by a *read* and a *release* (columns 1-3). As shown in column 2, a copy of the page is brought to node 2 when it reads the page, its update-counter is set to 0, and the back-up is set to 0 (the last-updater for the page).

Next, node 0 performs a *acquire-write-write-release* sequence (columns 4-7). When node 0 performs a release (column 7), it sends update messages to other nodes. As the back-up at node 0, immediately before the release is performed, is node 1, the update message sent to node 1 is tagged by a marker to inform node 1 that it is the back-up. When the acknowledgements for the update messages are received, node 0 determines its new back-up by finding the minimum of the update-counters received with the acknowledgement. As both nodes 1 and 2 return update-counter 1, node 0 arbitrarily chooses node 2 to be the back-up for its next update. The new back-up is shown in column 7 of the "back-up:0" row in the table. Nodes 1 and 2 set their *back-up* variable to 0, because they received an update from node 0 (column 7).

Next, node 1 performs *acquire-read-write-release* sequence (columns 8-11). When node 1 performs a release (column 11), the update message sent to node 0 is tagged with a marker, as node 0 is the back-up for this access (as shown in column 10, row "back-up:1"). When all the acknowledgements and update-counters are received, node 1 determines the new back-up as the node whose update-counter is the smallest, namely node 0. (The new back-up is shown in column 11, row "back-up:1"). Nodes 0 and

2 change their back-ups to 1, as they received an update from node 1 (column 11).

At this point (column 11), the update-counters for nodes 0, 1 and 2 are 1, 0 and 2, respectively. Next, node 0 performs an *acquire-write-release* sequence (columns 12-14). At the release by node 0 (column 14), the update message sent to node 1 is tagged by a marker, whereas that sent to node 2 is not tagged, as node 1 is the back-up for this update (see column 13, row back-up:0). When the update message is received by node 2, it performs the update and increments its update-counter to 3. Now, node 2 invalidates the local copy of the page because, (a) its update-counter has become equal to *limit* 3, and (b) the update message sent to node 2 was not tagged by a marker (which means that node 2 is not the back-up for the update). When node 0 receives the acknowledgements, it determines that node 1 is its new *back-up*. Also, node 1 sets its *back-up* to 0, when it receives the update-message.

Now, node 0 again performs *acquire-write-release* (columns 15-17) followed by another *acquire-write-release* (columns 18-20). At the second release (column 20), update-counter for node 1 becomes equal to 3. At each of the release, node 0 sends an update message to node 1 tagged with the marker. Therefore, node 1 cannot invalidate its copy of the page. Note that the update-counter at node 1 has become 3 (column 20), but the page is not invalidated.

Node 2 now performs *acquire-read-release* (column 21-23), therefore, it receives a copy of the page. Along with the page, it also receives identifier of the last-updater for the page. On receiving the page, its update-counter is set to 0, and back-up set equal to the last-updater. As node 2 did not write to the page, no update is necessary at the *release* (column 23).

Subsequently, node 2 performs *acquire-write-release* (columns 24-26). At the release, node 2 sends update messages to nodes 0 and 1, the message sent to node 0 being tagged with a marker. When node 1 receives the update, its update-counter becomes 4. Node 1 invalidates the page, as the update message was not tagged with a marker, and the update-counter is larger than the *limit*. □

Multiple Copy for Non-Shared Data

As discussed above, our protocol can guarantee that at least two copies of each page of shared data are maintained. The protocol described above ensures that the shared data remains available in spite of a single node failure. It does not (as yet) ensure that the faulty node can be recovered to its recent state. In some cases, it is adequate to ensure availability of the shared data.

When it is necessary to ensure that the faulty node can be recovered, the non-shared data must also be duplicated. We suggest a modification to the protocol described above. When a node writes shared data and updates other copies of the data, the non-shared data at the node is 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.

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 state is obtained from other nodes (the non-shared state 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 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.

- 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.

4 Performance Evaluation

In this section, we present preliminary results on performance of the proposed approach.

Methodology

We measured overhead for maintaining recoverable shared data 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. We used trace-driven simulation method for the experiments.

As a preliminary test, we generated synthetic trace data by using a trace generator. The trace generator can produce trace data according to the memory access behavior which we can define as input. We also modified the Proteus [3] to produce trace data for shared memory operations, acquire, release, read, and write. The trace data, produced by our synthetic trace generator or modified Proteus, 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 exchanged). 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 [6].

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, similar to [12]. 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.

Results

Figures 4, 5, and 6 show the result of our experiments with synthetic trace data. We assumed distributed shared memory system of 16 nodes. 10,000 memory accesses for a single page are simulated for three different read ratios (70%, 80%, and 90%) 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

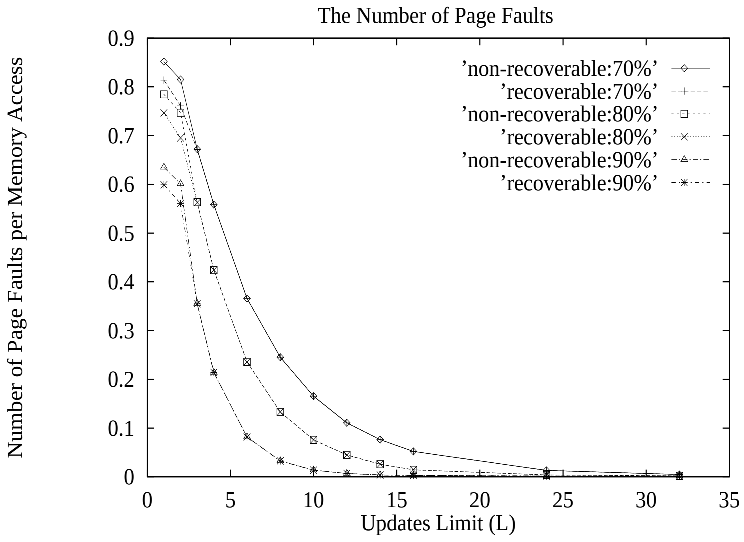


Figure 4: The number of page faults (synthetic trace)

cost for the limited update protocol *without* recoverable scheme at x% read ratio and "recoverable:x%" means the cost for the *recoverable* limited update protocol at x% read ratio.

Figure 4 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 5 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 6 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.

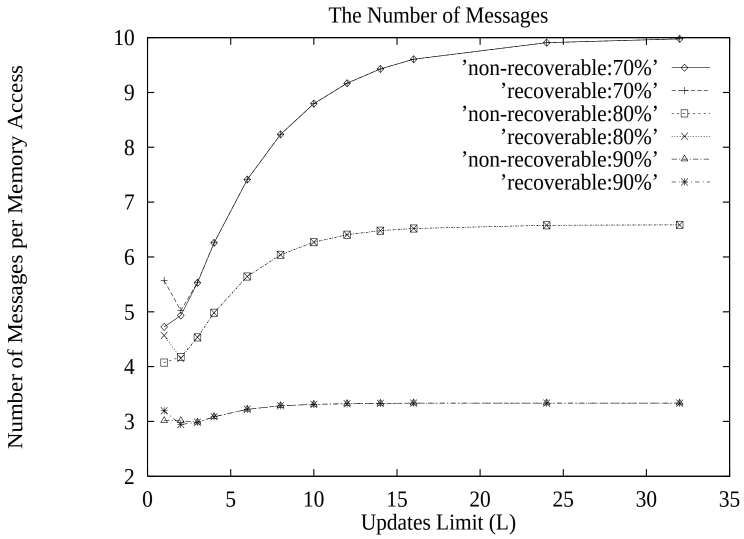


Figure 5: The number of messages (synthetic trace)

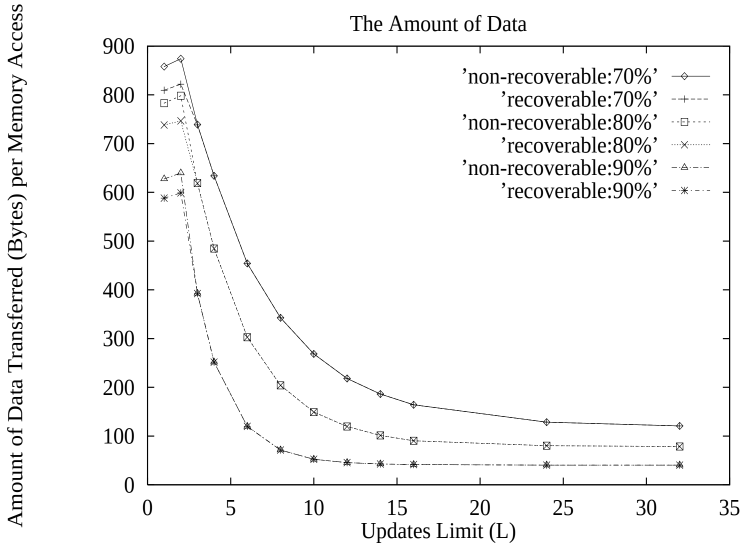


Figure 6: The amount of data (synthetic trace)

Application	*Shared Memory Access*	*Read Ratio*	*Input*
MP3D	822,639	65.7	1,600 molecules, 10 time steps
Floyd-Warshall	1,450,630	98.0	64 nodes
FFT	787,614	66.6	2^{13} points
Gauss-Jacobi	184,877	84.9	16 by 16

Table 1. Applications.

Four application programs, as shown in Table 1, were used to evaluate the overhead of recoverable limited update protocol. Figures 7, 8, 9, and 10 show the results. 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 presented above 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*.

5 Related Work

As discussed earlier, the *limited* update protocol is based on [1, 7]. As the focus of this report is on recoverable DSM, we now summarize the related work in this area.

Many recoverable DSM schemes have been presented in the literature. Many of them use stable storage (disk) to save recovery data [22, 21, 18, 11, 10, 5]. Some of them use main memory for checkpointing, replicating shared memory or logging the shared memory accesses [20, 2, 4, 15, 9, 13]. Proposed recoverable DSM belongs to the second category (uses main memory). [20], like proposed protocol, is based on update (full-replication) protocol, while [2, 4, 15, 9, 13] are based on invalidate (read-replication) protocol.

Stumm and Zhou extended four basic 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 "limited" update protocol (some copies are invalidated to reduce overhead). Additionally, our scheme supports *release* consistency.

Backward error recovery on a Cache Only Memory Architecture is implemented using the standard memories by Banatre et al. [2]. (A similar scheme was implemented on an Intel Paragon by Kermarrec et al. [13].) This scheme periodically take system-wide *consistent* checkpoints. Recovery data are replicated and mixed with current data in node memories in a transparent way using an extended

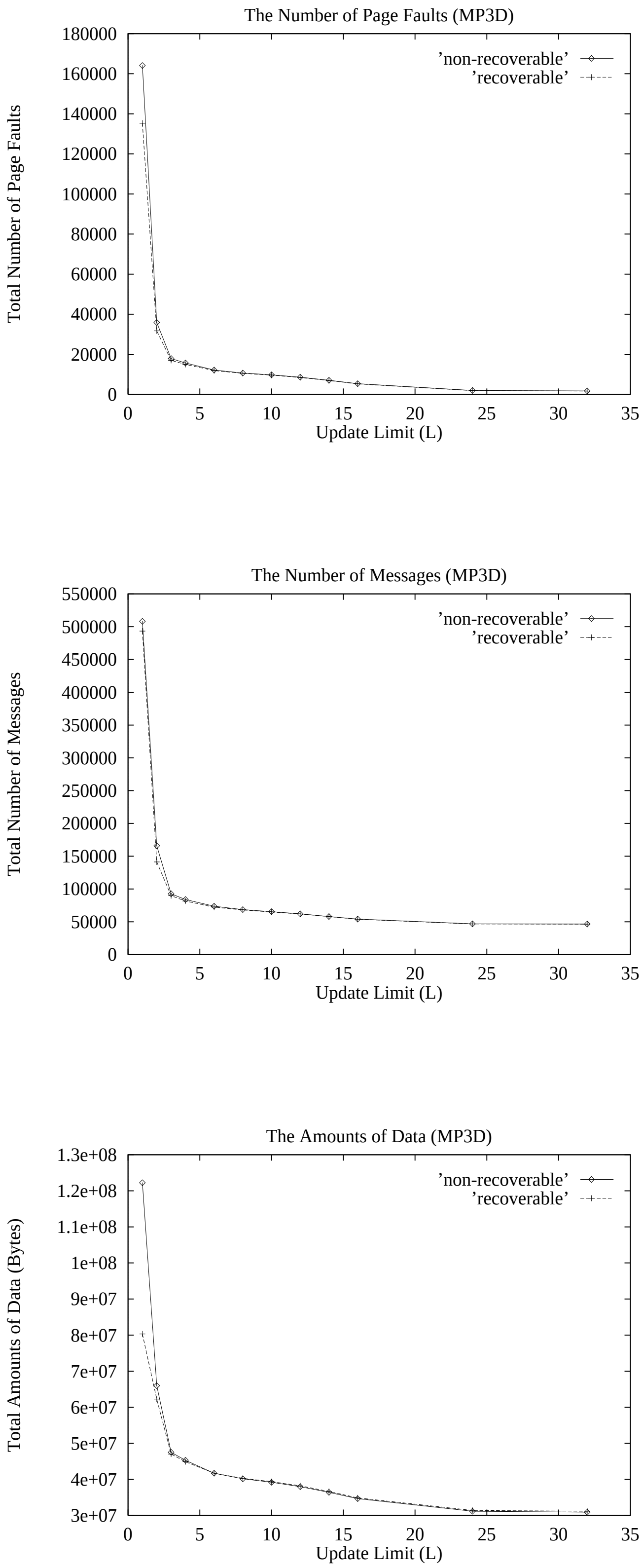
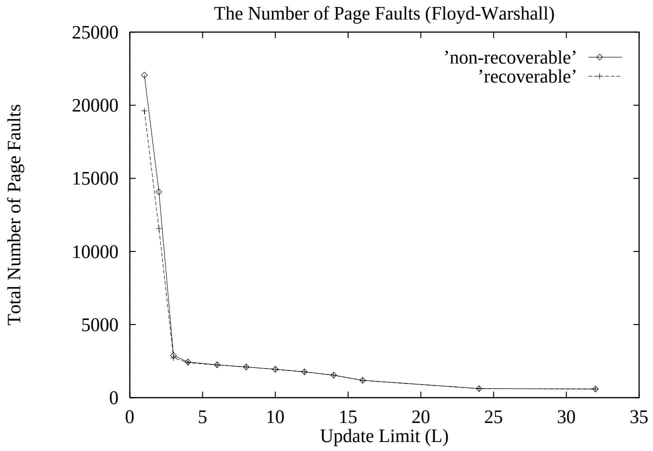
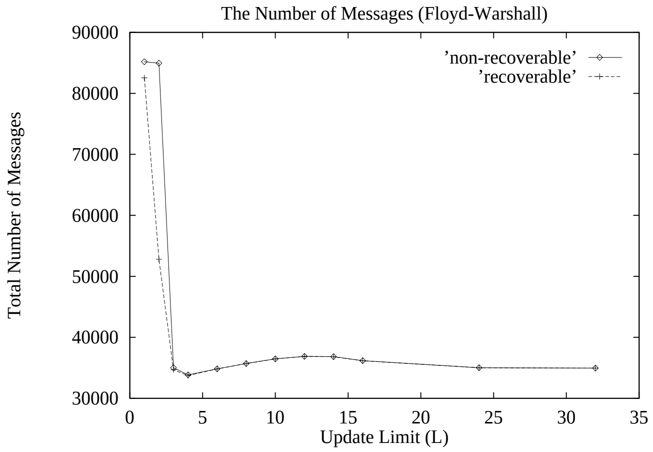


Figure 7: Overhead for Recoverable Scheme (MP3D)





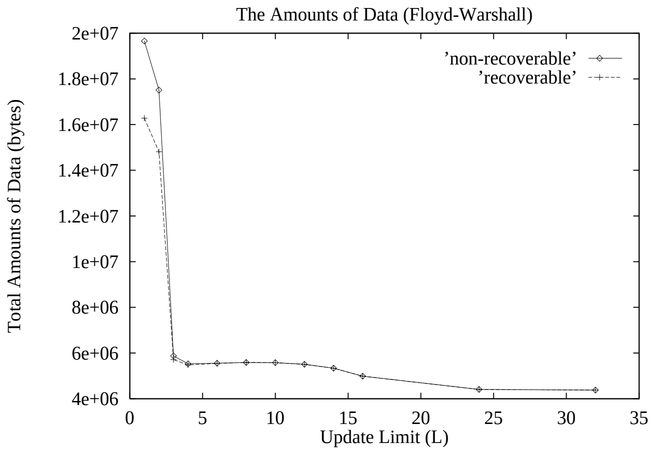


Figure 8: Overhead for Recoverable Scheme (Floyd-Warshall)

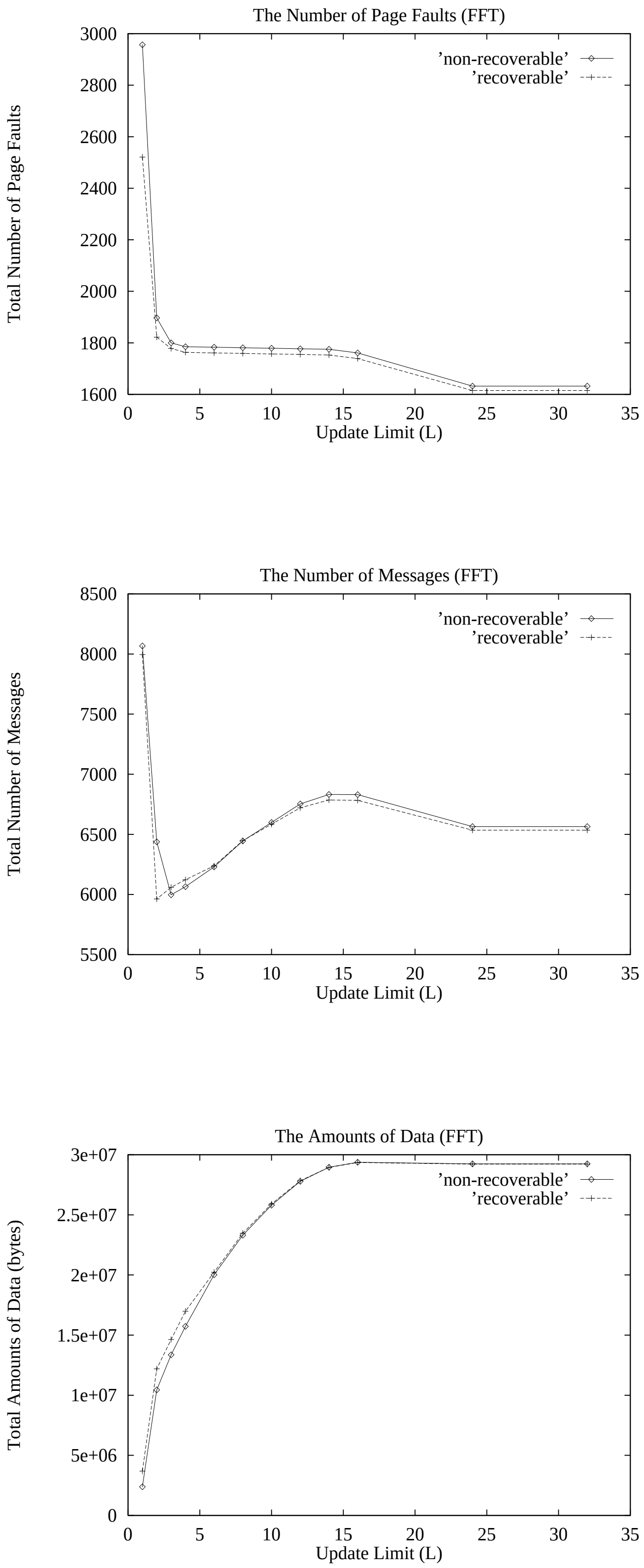


Figure 9: Overhead for Recoverable Scheme (FFT)

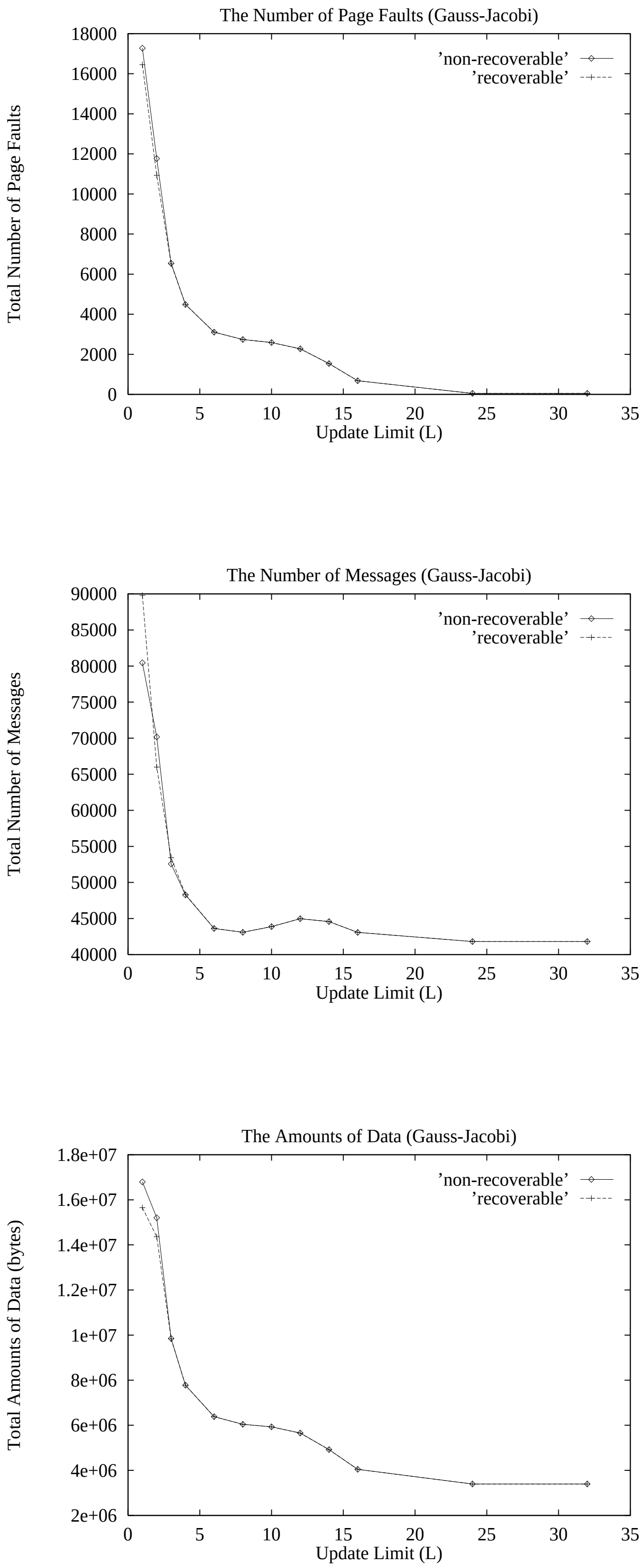


Figure 10: Overhead for Recoverable Scheme (Gauss-Jacobi)

coherence protocol based on *invalidate* protocol. Replicas are needed in order to ensure that two checkpoint copies for each page always exist. The memory overhead induced by this protocol varies with time (at least two copies are needed for each page, three copies are maintained for the modified pages, and four copies may be needed while establishing a consistent checkpoint). After a node fails, all nodes need to rollback to the last checkpoint.

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 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]. Additionally, our protocol incorporates the *release* consistency model, unlike the sequential consistency model used in [4] as well as [2, 13]

Neves et al. presented a checkpoint protocol for a multi-threaded distributed shared memory system based on the entry consistency memory model [15]. Their algorithm needs to maintain log of shared data accesses in the volatile memory. These logs are used to reconstruct failed processes from the last checkpoint. Fuchi and Tokoro proposed a mechanism for recoverable shared virtual memory [9]. 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. When the node of primary process fails, the backup process re-executes with the logged events from the last checkpoint. Above two scheme are similar to our scheme in the sense that they use volatile memory and provide recoverability from a single point of failure. However, they recover processes by a "replay" mechanism, whereas in our scheme no replay is necessary.

Janssens and Fuchs [11] present a recoverable DSM that uses relaxed consistency models. However, their approach is based on checkpointing, and results in a large number of checkpoints. Our approach, on the other hand, achieves recoverability by maintaining at least two copies of each page.

6 Conclusion and Future Work

This report 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 *limited 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 limited 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 *limited update* protocol. Recovery is simple because an active back-up copy exists for each page.

With a small number of exceptions, the previous work on recoverable DSM deals with the *invalidate* protocol and *sequential* consistency model. The report presents a comparison of the proposed scheme with the previous work.

Preliminary performance evaluation results indicate that the proposed scheme does not significantly increase the number or size of messages required by an application. Additional work is necessary to evaluate the performance of this scheme experimentally.

References

[1] J. Archibald, "A cache coherence approach for large multiprocessor systems," in *International Conference on Supercomputing*, pp. 337–345, July 1988.

[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 Documentation*, 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.

[5] G. Cabillic, G. Muller, and I. Puaut, "The performance of consistent checkpointing in distributed shared memory systems," Tech. Rep. 924, INRIA, 1995.

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

[7] F. Dahlgren, M. Dubois, and P. Stenstrom, "Combined performance gains of simple cache protocol extentions," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 187–197, Apr. 1994.

[8] S. Eggers and R. Katz, "A characterization of sharing in parallel prograns and its application to coherency protocol evaluation," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 373–382, May 1988.

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

[10] G. Janakiraman and Y. Tamir, "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputer," in *13th Symposium on Reliable Distributed Systems*, Oct. 1994.

[11] 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.

[12] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 13–21, May 1992.

[13] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A recoverable distributed shared memory integrating coherence and recoverability," Tech. Rep. 897, INRIA, 1995.

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

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

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

[17] N. Oba, A. Moriwaki, and S. Shimizu, "Top-1: A snoop-cache-based multiprocessor," in *Proc. 1990 International Phoenix Conference on Computers and Communication*, pp. 101–108, Oct. 1990.

[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 *Proceedings of International Conference on Parallel and Distributed Processing*, pp. 719–724, 1990.

[21] V.-O. Tam and M. Hsu, "Fast recovery in distributed shared virtual memory systems," in *Symposium on Distributed Computing Systems*, pp. 38–45, June 1990.

[22] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory: Memory coherence and storage structures," in *Proc. 19th Int. Symp. on Fault-Tolerant Computing*, pp. 520–527, 1989.