*Proc. 25th Int. Symp. on Fault-Tolerant Computing*, June 1995, to appear.

# Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory

Gaurav Suri

Bob Janssens

W. Kent Fuchs

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

## Abstract

*Rollback techniques that use message logging and deterministic replay can be used in parallel systems to recover a failed node without involving other nodes. Distributed shared memory (DSM) systems cannot directly apply message-passing logging techniques because they use inherently nondeterministic asynchronous communication. This paper presents new logging schemes that reduce the typically high overhead for logging in DSM. Our algorithm for sequentially consistent systems tracks rather than logs accesses to shared memory. In an extension of this method to lazy release consistency, the per-access overhead of tracking has been completely eliminated. Measurements with parallel applications show a significant reduction in failure-free overhead.*

## 1 Introduction

Distributed shared memory (DSM) provides the programming advantages of a shared memory image in a system with physically distributed processing nodes. DSM maintains consistency between processing nodes in software, using the virtual memory paging mechanism [15], or in hardware, using directory-based cache coherence [14]. Since DSM systems typically execute long-running applications on physically independent nodes, it is useful to have the ability to roll back to a previously saved state when a node crashes or is otherwise unable to continue the computation. Rollback techniques that use message logging and deterministic replay can be used in message passing systems to recover a failed node without involving other nodes. However, DSM systems can not directly apply these techniques, since DSM communication is inherently nondeterministic. Typically a DSM system has to incur the high overhead of logging all read accesses to implement deterministic replay.

This paper presents our implementation of reduced overhead logging algorithms for recovery in DSM systems. Unlike the only other logging scheme proposed for a general page-based DSM, by Richard and Singhal [18], our method does not need to log the contents of all accesses to shared memory. Instead it simply keeps a count of these accesses to maintain determinism. Furthermore, we show that, by using a relaxed memory consistency model and a multiple writer protocol, it is possible to eliminate all nondeterministic communication, making the tracking of shared accesses unnecessary. The algorithms have been integrated into a user-level software implementation of recoverable DSM on standard UNIX workstations. Measurements using parallel scientific applications show that our schemes reduce the failure-free overhead over the previous scheme [18] by more than an order of magnitude. Reexecution time after rollback is also significantly reduced.

Most previous work on rollback recovery in shared memory systems does not use logging; instead, checkpointing and/or recovery is coordinated between nodes to enable the system to roll back to a consistent state. The structure of shared memory simplifies recovery to a consistent state; allowing some dependencies caused by messages other than data transfers to be ignored [11]. Various early schemes for shared memory use communication-induced checkpointing, where data transfer induces the overhead of a checkpoint, ensuring that a rollback never results in an inconsistent state [1, 4, 10, 21]. Other schemes use consistent checkpointing where all nodes coordinate checkpointing to allow the system to always roll back to a consistent global state [1, 2, 8, 9].

A large body of research exists on logging in message passing systems. Logging allows a process to recover locally, and is therefore suited for environments where it is

This research was supported in part by the Office of Naval Research under grant N00014-91-J-1283, and by the National Aeronautics and Space Administration (NASA) under grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

impractical to involve other operational processes in recovery. Pure logging techniques that can tolerate an arbitrary number of failures need to log the contents and ordering of every message synchronously to stable storage upon receipt [17]. Execution needs to be piecewise deterministic, and a checkpoint has to be inserted for every nondeterministic event, so the system can recover to the exact state at the time of a failure by replaying the log after rollback. Asynchronous, or *optimistic* logging techniques reduce runtime overhead by grouping and delaying writes to stable storage [20]. Since not-yet-logged messages may be lost upon a failure, techniques ensuring rollback to a consistent state have to be implemented in addition to logging.

In DSM systems, the state of the computation on a node depends on the exact arrival times, with respect to the local computation, of request messages from other nodes. Therefore, the message-passing logging techniques cannot be used, since the arrival of every message would have to be considered a nondeterministic event and checkpointed. Fuchi and Tokoro propose solving this problem by implementing special hardware that pinpoints and logs the time of arrival of a message at clock cycle granularity [7]. This approach requires a modification of the microprocessor CPU hardware. Richard and Singhal avoid the nondeterminism of message arrivals by logging a copy of a shared page whenever a node reads it [18]. Since only the shared part of memory is directly affected by messages in a DSM, this scheme allows deterministic replay after rollback.

In this paper we present an improved algorithm for synchronous logging in DSM. A node logs a shared page to volatile storage every time it is obtained from another node. Logs are flushed to stable storage before exposing a modified page to another node. The non-determinism caused by request messages is avoided by *tracking* accesses to shared memory. Recovery is accomplished by rolling back the failed node to the last checkpoint and replaying its execution with the help of the logs.

Programmers of shared memory machines usually view the system as being sequentially consistent [13]. For most programs, however, it is possible to relax the system's consistency requirements while maintaining a sequentially consistent view for the programmer. By using relaxed consistency, and eliminating false sharing between nodes, it is possible to make all page transfers in DSM deterministic. Neves *et al.* use logging to implement recovery in an entry consistent memory model in an object-based DSM [16]. Since data is transferred at object rather than page granularity, and all objects are protected by associated synchronization variables, all data transfers can be performed at accesses to synchronization variables. Since synchronization accesses occur at deterministic times, it suffices to log object data transfers without tracking shared memory accesses. Feeley *et al.* use a similar consistency model in a recoverable transactional DSM [6]. All accesses consist of transactions, which are logged by commonly used database logging methods. A log-based coherency scheme is used where updates to shared data occur when local logs are flushed to other nodes at transaction commit time.

The relaxed consistency logging methods just described are not general enough to apply to a traditional DSM where the unit of data transfer is a fixed cache block or memory page. We present a logging method that uses release consistency with multiple writers [12] to eliminate nondeterministic data transfer in such systems. Since the tracking overhead for every shared access is eliminated, this method further reduces the logging overhead over our sequentially consistent scheme.

No measurements of the overhead of checkpointing and/or logging for actual implementations of recoverable DSM exist in the literature. Feeley *et al.* present performance data from a prototype implementation of their log-based coherency scheme for transaction processing [6]. However, since in this scheme logging is also needed to maintain coherence of the shared address space, it is not possible to measure the overhead of supplying recoverability. While most the communication-induced and consistent checkpointing schemes for shared memory have been evaluated by trace-driven simulation, none of the other logging schemes have been quantitatively evaluated. The measurements in this paper constitute the first performance overhead numbers for logging-based recoverable DSM that runs general scientific code. They also constitute the first results for an actual prototype implementation of a recoverable DSM for scientific programming.

## 2 Logging and Rollback in DSM Systems

The aim of logging with independent checkpointing is to provide local recovery capability in the event of a node failure. Local recovery means that the node that failed can recover to its pre-failure state without involving the other nodes in the system in the recovery process. Our schemes are designed to recover from transient stopping failures of nodes in a DSM system. The network is assumed to be fault-free and the disks reliable. Node failures are detected via timeouts or some other detection mechanism. The execution on every node is piecewise deterministic [20], so that a failed node's state can be rebuilt by rolling back to a previous checkpoint and replaying execution from the log. Since our aim is to provide recovery from multiple failures without inter-node coordination, we use synchronous logging to stable storage.

In synchronous logging in message passing systems, every node maintains a log of the messages it received from other nodes and, during recovery, the receipt of a message

Figure 1: Shared data access patterns in DSM.

is satisfied from this log. Since the receipt of a message is explicitly specified in the program, the ordering of the messages in the log suffices to apply the log entries to the correct receipt at the correct time. In DSM systems, however, a remote node may send a message at any time, which is then asynchronously received, possibly changing the local state. During recovery, without any additional information, it is not possible to determine when to apply the next receive in the log.

We illustrate the problem of logging asynchronous receives in DSM with the help of an example. The system consists of a number of processing nodes which can cache copies of the shared variables they access. A part of each node's address space is dedicated to shared pages. The fixed distributed manager protocol for software DSM [15] is used to maintain coherence. Each page has a home, which manages the ownership rights for that page. More than one node may have a read only copy of a page as long as there is no one writing to the page. There may be only one node writing to a page at any given time, and at that time no one else can access the page. When a node tries to access a page for which it does not have access rights, a page fault occurs and a fault handler is invoked which services the fault and retrieves an up-to-date copy of the page. In case of a write fault, invalidate messages are sent to all nodes that currently hold copies of the page.

Figure 1 shows an example of DSM shared page access patterns using one shared page and three nodes. Initially, node $A$ owns page $x$ along with exclusive write access, so its first two accesses to the page do not cause a fault. When node $B$ tries to write page $x$, a page fault occurs, the fault handler obtains a copy of the page from the owner $A$, and $B$ becomes the new owner. $A$ sets its access to invalid, causing its next read to fault and ask the owner $B$ for a copy. Node $B$ downgrades its access to read-only, and adds $A$ to the copyset for page $x$. Finally, Node $C$ requests write access to page $x$, causing node $B$ to invalidate its local copy and send the page and its copyset to $C$. Node $C$ then uses the copyset to invalidate the readable copy in node $A$.

Assume that now node $A$ rolls back to a checkpoint at the beginning of the diagram. If we could record the exact instance at which messages are received, it could recover by replaying these messages at the correct time. However, consider what happens when node $A$ starts recovery. The first entry in the log is the write request message $M_1$ from node $B$. For deterministic re-execution it is necessary to apply the resulting invalidate between the second and third accesses to $x$. However, unless additional timing information is logged, the log has no knowledge that the request actually occurred between these accesses.

One solution to this problem is to model communication as occurring between the application process and the global shared memory system. In this model every read access causes a message with the data contents to be received from the shared memory, and therefore needs to be logged. Richard and Singhal's scheme uses this model, logging a copy of a shared page whenever it is read [18]. We will refer to this scheme as *shared-read logging*. A better solution is to maintain enough ordering information in the log so that all accesses to shared memory can be correctly replayed. In our scheme, the only data logged are the contents of the pages received in response to a read or write request. All shared memory accesses are tracked, and a count is maintained of the number of accesses made between invalidates.

With access tracking, in the example of Figure 1, node $A$ increments a counter whenever it accesses page $x$ during normal execution, and logs and resets the counter when that page is invalidated. After the rollback, the first entry in the log now states that there will be two accesses before the first invalidate, so the read and write to $x$ are re-executed and the page is invalidated before the next read of $x$. Then when that read occurs, a page fault causes the node to consult the next entry of the log, which has the contents of page $x$ that were sent during normal execution. Next, the log will contain another record that $x$ was accessed twice before the next invalidation, so the message from node $C$ is also applied at the correct time.

From the example just described, it is clear that logging only the data transfers and tracking shared memory accesses to correctly log invalidate messages can reduce the amount of data logged considerably compared to schemes that log all accesses to shared data. The reduced size of the log also ensures faster recovery because the number of accesses to stable storage during recovery is greatly reduced.

**Read Fault Handler**

```
if (in_recovery)
   read next page from page log;
   copy page to address for the
    page in memory;
   set permission for page to read only;
else
   send page request to manager of page;
   get page from current owner;
   set permission for page to read only;
   log page to volatile log;
endif
```

**Write Fault Handler**

```
if (in_recovery)
   read next page from log;
   copy page to address for the
    page in memory;
   set permission for page to read-write;
else
   send page request to manager of page;
   get page from current owner;
   set permission for page to read-write;
   send invalidates to all members of copyset;
   log page to volatile log;
endif
```

**Read Request Server**

```
if (in_recovery)
   put request in pending queue;
else
   log change in permission with
    inv_ctr to volatile log;
   inv_ctr = 0;
   flush log to stable storage;
   send page to requesting node;
   set permission for page to read only;
   add requesting node to copyset;
endif
```

**Write Request Server**

```
if (in_recovery)
   put request in pending queue;
else
   log invalidate with inv_ctr
    to volatile log;
   inv_ctr = 0;
   flush log to stable storage;
   send page to requesting node;
   set permission for page to invalid;
endif
```

**Invalidate Request Server**

```
if (in_recovery)
   put request in pending queue;
else
   invalidate page;
   log invalidate with inv_ctr
    to volatile log;
   inv_ctr = 0;
endif
```

**Shared Memory Access Tracking**

```
if (in_recovery)
   inv_ctr++;
   if (inv_ctr == next_invalidate)
      invalidate page = pagenum from
       invalidate log;
      read next_invalidate from
       invalidate log;
      inv_ctr = 0;
   endif
else
   inv_ctr++;
endif
```

Figure 2: Pseudo-code for logging.

# 3 The Sequential Consistency Algorithm

The three important components of our algorithm for sequential consistency are the logging of shared pages during failure-free operation, periodic checkpointing of process state, and local recovery of the failed node with the help of the log maintained on stable storage. This section describes the components in detail.

## 3.1 Logging

In our algorithm, logging is integrated with the memory coherence mechanism. The system maintains two kinds of logs: one for shared pages and one for the invalidate requests serviced during failure-free operation. The page log consists of two fields, the page number and the page itself. The invalidate log has two fields: *inv_ctr*, which is the number of shared accesses before the next invalidate and *page_num*, the page number to be invalidated.

Pseudo-code for the logging algorithm integrated into the fixed distributed manager DSM algorithm is given in Figure 2. The shared access tracking procedure is invoked every time a shared access occurs. The read or write fault handler is called when a node tries to access a page for which it does not have read or write permission. The fault handler contacts the node that owns the page, which causes the respective request server to be called there. The write fault handler may also send invalidate messages, which are processed by invalidate servers on the receiving nodes. Once the page request has been serviced and the node has the page, it logs the page to its volatile log.

In order to ensure that the node can recover to a consistent state during recovery, the volatile logs are flushed to stable storage before a node grants a page request and transfers a page to another node [18]. Thus, the stable logs will always have sufficient data to reconstruct the process state to beyond a point where modifications that it made to shared data have been made visible to other nodes.

## 3.2 Checkpointing

The scheme allows for nodes to checkpoint independently. Checkpoints are taken periodically so that the logs do not grow without bound and, in case of failure, the node does not lose too much computation. A checkpoint consists of the complete processing node state including its shared pages and its pagetable. At the time of checkpointing, a node also flushes the contents of its volatile log to stable storage. Once the checkpoint has been completed and saved to stable storage, the previous checkpoint and the stable log for the node are discarded.

## 3.3 Recovery

When a node failure is detected, recovery is initiated by restarting its computation on the failed node from its most recent checkpoint. The *in_recovery* flag is set and execution starts from the restored state. All the page fault requests on the recovering node during this period are serviced from the page log. The invalidate log is used to invalidate shared pages at appropriate times so that the execution is a deterministic reconstruction of the pre-failure execution. While the node is recovering, it blocks all service requests from other nodes. This prevents operational nodes from accessing stale data from the recovering node. The recovery process is completed when the entire log is consumed. The node then unsets the *in_recovery* flag, unblocks service of requests from other nodes and resumes normal execution. The details of the recovery algorithm are included in the pseudo-code presented in Figure 2.

There is a possibility that a node initiates a rollback after it has received a request for a page, but before it sends the page. In this case, the requester will never receive the page, so it resends the request if it does not receive a reply after a specified timeout period. Duplicate requests are detected by using sequence numbers [3]. There is also a possibility that a node rolls back after acquiring ownership of a page, but before the volatile log is saved to stable storage. In this case the page manager has incorrect ownership information. To tolerate this, ownership information is maintained redundantly by using an ownership timestamp on every node for every page [11]. The owner of a page is guaranteed to have the largest ownership timestamp. If a node receives a request to a page it does not own, it determines the correct owner by requesting and comparing ownership timestamps from each node.

# 4 Relaxed Consistency

One of the main disadvantages of our logging algorithm is that it needs to keep a count of all read and write accesses to a shared page. In a software implementation, this means that for every shared access some code has to be inserted to update a counter. The only way to avoid this overhead for every access to shared data is to eliminate asynchronous requests that change access permissions to a local page.

To eliminate asynchronous requests, it is necessary to relax the memory consistency model. In the usual sequential consistency model, memory accesses are seen by all nodes in program order [13]. Release consistency allows accesses between synchronization points to be seen in any order, allowing the sending of invalidation messages to be delayed [12]. The only ordering between accesses on different nodes is enforced by pairs of release (unlock) and acquire (lock) accesses to synchronization variables. For ex-

**Read Fault Handler**

```
if ( in_recovery )
   read diffs from log;
   apply diffs to local page;
   set permission for page to read only;
else
   send page request to any node
    with a valid copy;
   get page from that node;
   get diffs from other writers;
   log diffs to volatile log;
   apply diffs to local page;
   set permission for page to read only;
endif
```

**Read Request Server**

```
if ( in_recovery )
   put request in pending queue;
else
   flush log to stable storage;
   create diffs from twinned pages;
   send diffs to requesting node;
   set permission for page to read only;
endif
```

**Write Fault Handler**

```
create twin of page;
set permission for page to read-write;
```

**Acquire**

```
if ( in_recovery )
   read write notices from log;
   apply write notices to invalidate pages;
else
   send acquire request to lock manager;
   receive write notices from last acquirer;
   log write notices to volatile log;
   apply write notices to invalidate pages;
endif
```

**Acquire Server**

```
if ( in_recovery )
   put request in pending queue;
else
   flush log to stable storage;
   wait until lock is released;
   send write notices to requester;
endif
```

Figure 3: Pseudo-code for logging with lazy release consistency.

ample, if node $A$ unlocks lock variable $s$ and node $B$ later locks $s$, then all data accesses before the unlock on $A$ are ordered before all data accesses after the lock on $B$. As long as the application programmer ensures that the program does not have any data races, a release consistent system is indistinguishable from a sequentially consistent system. A data race occurs when two data operations access the same memory location, they are not both reads, and they are not ordered by an acquire-release pair.

Lazy release consistency attempts to reduce the number of messages needed to maintain coherence in software implementations of DSM [12]. It delays propagation of modifications made to data protected by a synchronization variable until the acquire of the variable by another node. At that time, the last releaser sends a set of write notices to the acquirer. These write notices describe the modifications to shared pages that precede the acquire. The acquiring node then invalidates all pages for which it has an out-of-date copy. A multiple writer protocol is used to avoid messaging overhead due to false sharing of variables in the same page. A node that receives write permission for a page creates a twin copy of the page to which it makes modifications. When another node requests access to that page, the original and twin are compared to create a *diff* record which is sent to the requester. The requester then applies the diffs to its copy of the page before accessing it.

The important feature of lazy release consistency for logging purposes is that all changes to the access permissions of pages occur at deterministic points. Invalidations are delayed until a node acquires a lock and requests the write notices. The multiple writer protocol allows all nodes that have write permission to a page to keep that permis-

sion even if another node wants access. Therefore, it suffices to log the messages received during all acquires and access misses during normal execution. When the log is replayed, the correct invalidations are applied at acquire points, and the access misses occur at the correct shared memory accesses, ensuring deterministic re-execution to the state that existed right before the failure. Figure 3 contains the pseudo-code for the logging algorithm with lazy release consistency. It is similar to the sequential consistency case, except that shared memory accesses are not tracked. Instead invalidations are logged and replayed in the acquire handler, which is called every time an acquire of a lock succeeds.

# 5 Performance Overhead Results

To verify the correctness of our approach, and to measure overhead for real applications, we developed, a user-level implementation of DSM under UNIX and implemented Richard and Singhal's shared-read logging scheme [18] and our sequential consistency scheme. For these two schemes, both error-free overhead and recovery performance were measured. In addition, measurements were taken on the error-free overhead of logging if accesses are not tracked, which would be the case in the lazy release consistency algorithm.

## 5.1 Implementation of the prototype

The recoverable DSM testbed runs under both BSD and Solaris versions of SunOs on Sun Microsystems workstations. The implementation is wholly in user space; the kernel is involved only in handling system calls, passing messages, and passing on interrupts to the user application. The system consists of a library that is linked to a user application. Each node in the system runs its own copy of the application. At initialization, a user-specified area of virtual memory is designated shared and protected to disallow read and write accesses.

During execution, the application accesses memory as usual, but when a protection violation is detected by the kernel, an interrupt handler is called. The interrupt handler uses the address of the access to index into a page table containing the page handler for the affected page. The page handler then communicates through the kernel with the other nodes and changes its internal state. If necessary it receives a new copy of the page and stores it at the correct virtual memory address. Finally, it tells the kernel to modify the protections so the application can access the page. When a request for access to a page arrives from another node, the kernel interrupts the application and calls the interrupt handler to index into the pagetable. The corresponding page handler performs the required action. It might need to ask the kernel to change access permission to the page, or it might need to send a copy of the page to the requesting node. Then the system returns to executing the application.

The advantage of implementing the system in user space is that it can run on any Sun workstation, without modifications to the kernel. The disadvantage is that the interrupt handling and communication mechanism is slow. Since we do not have a high-speed network available, we ran our experiments for this paper on a four-processor shared memory SPARCsystem-600 server. The system spawns four processes and lets the operating system schedule them. Communication is through sockets which are in turn implemented on top of the physically shared memory. The page size for the Sun virtual memory architecture and therefore the page size used in our prototype is 4 kilobytes.

## 5.2 Experimental setup

Our results are based on 4-node execution of three parallel applications: water, SOR, and prefix. Water, obtained from the SPLASH suite [19], is a molecular dynamics simulation. SOR implements successive overrelaxation. Prefix uses successive matrix matrix multiplications to perform the prefix product on an array of matrices.

We implemented logging using both the shared-read logging scheme and our shared access tracking scheme. Shared-read logging logs a copy of every shared page that is read to volatile storage, detecting exact duplicate copies of a logged page to save space. The volatile log is flushed to disk before a node grants a page request. Our shared access tracking scheme is implemented exactly as described in Section 3. We ran two sets of experiments, one to measure failure-free overhead of the logging schemes, and one to measure the time it takes to recover after the system has rolled back. Due to the large disk space requirements for the shared-read logging scheme, it is not possible to run the recovery overhead measurements for very large datasets. Hence, the inputs to the benchmarks were varied between the two sets of experiments. The inputs used for the various benchmarks for experiments are presented in Table 1.

For the failure-free measurements, five different experiments were performed using the three benchmarks. Since the goal was to measure the overhead of logging, no checkpointing was done in the experiments. We ran the application without logging, with the shared-read logging scheme, and with our shared access tracking scheme. Three variations of our access tracking scheme were implemented. In the first variation, access tracking is performed using a function call. To reduce overhead, the second variation inlines access tracking into the application's code. In the third variation, access tracking is not performed at all. This third variation allows us to measure the overhead that could

Table 1: Inputs used for benchmarks.

| Benchmark | Description | Failure-free input | Recovery input |
|---|---|---|---|
| water | molecular dynamics | 8 steps of 216 molecules | 8 steps of 27 molecules |
| SOR | successive overrelaxation | 300 iter., $512^2$ entries | 100 iter., $512^2$ entries |
| prefix | matrix prefix | 15 100x100 matrices | 5 20x20 matrices |

Table 2: Failure-free overhead measurements for prefix. Total number of shared memory accesses is 56.3 million.

| Logging Scheme | Exec. time (s) | % overh. | pages logged | no. of flushes |
|---|---|---|---|---|
| None | 70.6 | - | - | - |
| Shared-read logging | 67560.4 | 95600 | 14.32M | 20046 |
| Func. call tracking | 99.2 | 40.6 | 5845 | 4836 |
| Inlined tracking | 77.0 | 9.04 | 5845 | 4836 |
| No tracking | 76.0 | 7.62 | 5845 | 4836 |

Table 3: Failure-free overhead measurements for SOR. Total number of shared memory accesses is 391.028 million.

| Logging Scheme | Exec. time (s) | % overh. | pages logged | no. of flushes |
|---|---|---|---|---|
| None | 72.4 | - | - | - |
| Shared-read logging | 900.6 | 1140 | 82718 | 2434 |
| Func. call tracking | 197.6 | 173 | 4424 | 2441 |
| Inlined tracking | 94.2 | 30.0 | 4424 | 2441 |
| No tracking | 85.4 | 17.9 | 4424 | 2441 |

be reduced by the lazy release consistency model, where access tracking is not performed.

For the recovery time measurements, the same two logging schemes were evaluated, but the three variations for the access tracking scheme were not introduced. The recovery time for a failed node depends on how recently a checkpoint has been taken. To obtain a measure of recovery overhead independent of checkpointing frequency, our recovery experiment runs the parallel application completely to the end, then rolls back node 0 to the beginning of the parallel part of the application, and re-executes it completely from the log.

### 5.3 Experimental results

Tables 2, 3 and 4 show the data for the failure-free overheads in terms of actual execution time, percentage overhead with respect to the normal execution times of the programs, the total number of pages logged, and the number of times the volatile log was flushed to stable storage. The number of pages logged is an indication of the disk space the log requires, while the number of flushes gives a measure of how frequently disk accesses need to be made.

Table 4: Failure-free overhead measurements for Water. Total number of shared memory accesses is 33.832 million.

| Logging Scheme | Exec. time (s) | % overh. | pages logged | no. of flushes |
|---|---|---|---|---|
| None | 409.9 | - | - | - |
| Shared-read logging | 11579.0 | 2730 | 2.538M | 73015 |
| Func. call tracking | 546.3 | 33.3 | 62150 | 56012 |
| Inlined tracking | 511.6 | 24.8 | 62150 | 56012 |
| No tracking | 496.7 | 21.2 | 62150 | 56012 |

Figure 4 shows a comparison of the failure-free overhead calculated as a percentage of normal execution time for the shared-read logging scheme, and the three variations of our access tracking scheme described earlier. The plot clearly shows that the tracking scheme outperforms the shared-read logging scheme by more than an order of magnitude. This is expected since only a small fraction of all accesses actually lead to page transfers between nodes and thus the tracking scheme logs a much smaller number of pages. The overhead for shared-read logging for Prefix is much higher than the other two programs. The shared memory access pattern for Prefix is such that it causes a page to be modified and thus logged again almost once every four accesses, leading to high overhead. The overhead with inlined tracking is significantly less than the overhead with tracking as a procedure call, and the overhead with no tracking is even less. This shows that the tracking overhead is a significant portion of the total. Though the exact contribution of the tracking overhead varies from program to program and depends on the relative frequency of page faults in the program, it is clear that any scheme that does not need tracking will suffer from lower overheads. Our scheme for lazy release consistency does not need tracking to ensure correct recovery and can thus be expected to have a lower overhead than the one for strict consistency.

One of the factors that will determine the checkpointing frequency for a DSM system with logging is the space taken

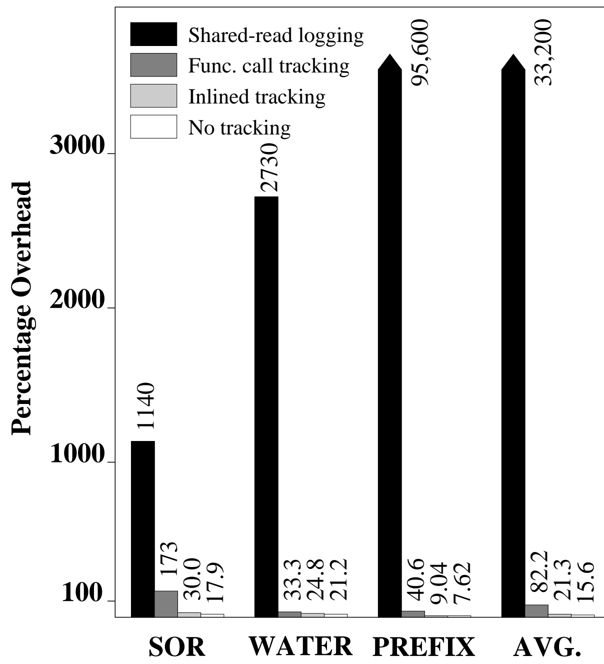


Figure 4: Failure free overhead as a percentage of execution time with no logging.

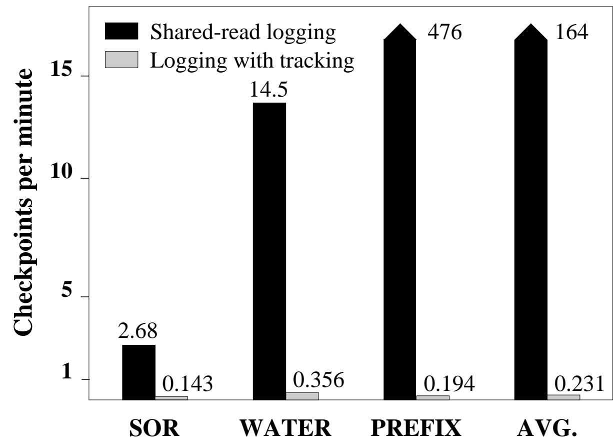


Figure 5: No. of checkpoints per minute of normal execution time for a bounded log size of 25 Megabytes.

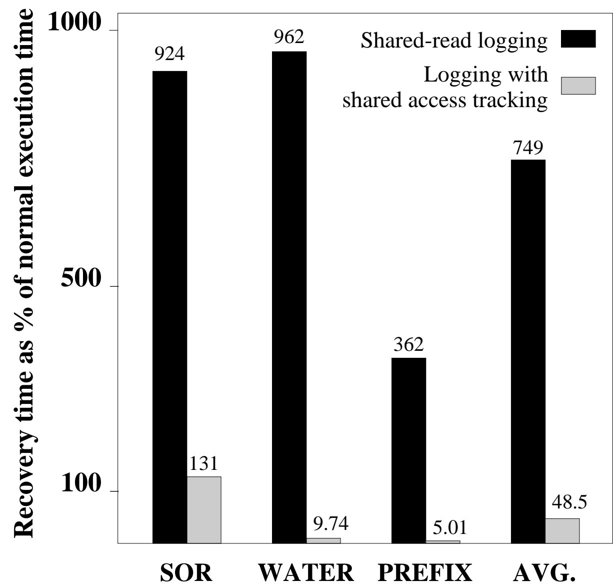


Figure 6: Recovery time as a percentage of normal execution time for the rolled back code.

by the log. If the disk space available for the log is exhausted, it is necessary to take a checkpoint and remove the log. Plots of the checkpointing frequency needed to limit the size of the log to 25 megabytes per node are given in Figure 5. The checkpoint frequency is given in number of checkpoints per minute of the normal execution time of the programs, i.e., the execution time with no logging or checkpointing.

The checkpointing frequency for the shared-read logging approach is higher than the tracking scheme in a proportion similar to the failure-free overheads. This is natural, since when there are more pages needing to be logged, the the frequency of logging is higher, and hence the log reaches the bound and is forced to checkpoint much sooner.

Figure 6 shows a plot of the recovery times for the three programs as a percentage of the normal execution times for the rolled back portion of the programs. This gives a measure of the additional overhead involved in recovering from an error, as a percentage of the runtime of the portion of the execution that was rolled back. As the graph shows, our schemes re-execute the rolled back code faster after rollback than originally before rollback for two of the benchmarks. This is because of the fact that a page fault, which resulted in network communication to get a copy of the page is now replaced by a disk read, which is a cheaper operation in most cases since the log is kept on the local disk of the workstation. Shared-read logging does not benefit from the same phenomenon because the number of pages that it logs is much larger than the number of page faults in the failure-free execution, and hence they have to make a correspondingly greater number of disk reads during recovery.

## 6 Summary and Conclusions

Logging and deterministic replay is a common technique used in parallel systems to allow recovery from errors during execution. The overhead of logging is high compared to coordinated checkpointing [5]. Therefore, logging is generally used only when local recovery of processing nodes is desired.

This paper has presented results on reducing the overhead of logging in DSM systems. A significant reduction was achieved by tracking rather than logging all accesses to shared data. The results show that tracking is still a significant proportion of the logging overhead. Lazy release consistency can be used to eliminate tracking of shared accesses altogether, restricting overhead to the logging of actual page transfers.

## Acknowledgements

We thank the other members of our research group, especially Nuno Neves and Sujoy Basu, for their comments.

## References

[1] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, "Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems," *Proc. 20th Int. Symp. on Fault-Tolerant Computing,* 1990, pp. 82–88.

[2] M. Banâtre, A. Gefflaut, P. Joubert, P. Lee, and C. Morin, "An architecture for tolerating processor failures in shared-memory multiprocessors," Tech. Report 707, IRISA, Rennes, France, Mar. 1993.

[3] J. Bartlett *et al.*, "Fault tolerance in Tandem computer systems," in D. P. Siewioriek and R. S. Swarz, *Reliable Computer Systems,* Bedford, MA: Digital Press, 1982, pp. 586–648.

[4] P. A. Bernstein, "Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing," *Computer,* Vol. 21, No. 2, Feb. 1988, pp. 37–45.

[5] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," *Proc. 11th Symp. on Reliable Distributed Systems,* 1992, pp. 39–47.

[6] M. J. Feeley, J. S. Chase, V. Narasayya, and H. M. Levy, "Integrating coherency and recovery in distributed systems," *Proc. Symp. on Operating Systems Design and Implementation,* 1994, pp. 215–227.

[7] T. Fuchi and M. Tokoro, "A mechanism for recoverable shared virtual memory," manuscript, U. of Tokyo, 1994.

[8] A. Gefflaut, C. Morin, and M. Banâtre, "Tolerating node failures in cache only memory architectures," *Proc. Supercomputing '94,* 1994, pp. 370–379.

[9] G. Janakiraman and Y. Tamir, "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers," *Proc. 13th Symp. on Reliable Distributed Systems,* 1994, pp. 42–51.

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

[11] B. Janssens and W. K. Fuchs, "Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory," *Proc. 13th Symp. on Reliable Distributed Systems,* Oct. 1994, pp. 34–41.

[12] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," *Proc. 19th Int. Symp. on Computer Architecture,* 1992, pp. 13–21.

[13] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Computers,* Vol C-28, No. 9, Sep. 1979, pp. 690–691.

[14] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," *Proc. 17th Int. Symp on Computer Architecture,* 1990, pp. 148–159.

[15] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. on Computer Systems,* Vol. 7, No. 4, Nov. 1989, pp. 321–359.

[16] N. Neves, M. Castro, P. Guedes, "A checkpoint protocol for an entry consistent shared memory system," *Proc. 13th ACM Symp. on Principles of Distributed Computing,* 1994, pp. 121–129.

[17] M. L. Powell and D. L. Presotto, "Publishing: a reliable broadcast communication mechanism," *Proc. 9th ACM Symp. on Operating Systems Principles,* 1983, pp. 100–109.

[18] G. G. Richard III and M. Singhal, "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory," *Proc. 12th Symp. on Reliable Distributed Systems,* 1993, pp 58–67.

[19] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," Tech. Report CSL-TR-91-469, Stanford U., Apr. 1991.

[20] R. E. Strom and S. Yemeni, "Optimistic recovery in distributed systems," *ACM Trans. on Computer Systems,* Vol. 3, No. 3, Aug. 1985, pp. 204–226.

[21] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Trans. on Computers,* Vol. 39, No. 4, Apr. 1990, pp. 460–469.