Lightweight Logging for Lazy Release Consistent Distributed Shared Memory

Manuel Costa, Paulo Guedes, Manuel Sequeira,

IST-NESC

Lightweight Logging for Lazy Release Consistent Distributed Shared Memory

Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves, Miguel Castro

IST - INESC
R. Alves Redol 9, 1000 Lisboa PORTUGAL
{msc, pjg, mds, nuno, miguel}@inesc.pt

Abstract

This paper presents a new logging and recovery algorithm for lazy release consistent distributed shared memory (DSM). The new algorithm tolerates single node failures by maintaining a distributed log of data dependencies in the volatile memory of processes.

The algorithm adds very little overhead to the memory consistency protocol: it sends no additional messages during failure-free periods; it adds only a minimal amount of data to one of the DSM protocol messages; it introduces no forced rollbacks of non-faulty processes; and it performs no communication-induced accesses to stable storage. Furthermore, the algorithm logs only a very small amount of data, because it uses the log of memory accesses already maintained by the memory consistency protocol.

The algorithm was implemented in TreadMarks, a state-of-the-art DSM system. Experimental results show that the algorithm has near zero time overhead and very low space overhead during failure-free execution, thus refuting the common belief that logging overhead is necessarily high in recoverable DSM systems.

1. Introduction

Networks of workstations are a cost-effective alternative to dedicated parallel computers for running compute-intensive applications [Anderson 95]. In these networks, the simplicity of programming with a shared memory abstraction can be retained by using distributed shared memory systems [Li 86, Keleher 94]. As these networks grow and the number of nodes running a DSM application increases, the probability of a failure also increases. This probability can become unacceptably high, especially for long-running applications. Therefore, DSM systems need mechanisms to tolerate faults in computing nodes.

This paper presents a new logging and recovery algorithm for a DSM system based on lazy release consistency (LRC) [Keleher 92]. This algorithm is inspired in our previous work where an approach with comparable properties was applied to an entry-consistent memory system [Neves 94]. We extended TreadMarks [Keleher 94], an efficient implementation of LRC, to implement the logging and recovery algorithm. Experimental results show that the algorithm has near zero time overhead and very low space overhead during failure-free execution, thus refuting the common belief that logging overhead is necessarily high in recoverable DSM systems [Cabillic 95, Suri 95]. We show that the logging overhead of the algorithm is much lower than the previously proposed logging algorithms for recovery of DSM systems based on LRC [Suri 95].

The algorithm tolerates single node failures by maintaining a distributed log of data dependencies in the volatile memory of processes. Whenever a process sends relevant information to another, it keeps in its volatile memory a log of that information. The algorithm achieves very good performance because it is tightly integrated with the LRC memory coherence protocol. Since the LRC protocol already keeps most of the relevant information in its data structures, the algorithm only has to log the instants at which the information is transferred. Whenever a remote synchronization operation occurs, it logs a pair with the vector times of the sender and receiver in the volatile memories of both processes. The algorithm does not log the value of shared memory updates, because it takes advantage of the log of memory accesses already maintained by the memory consistency protocol. This results in significantly reduced space overhead when compared to previously proposed solutions [Suri 95]. Furthermore, the algorithm sends no additional messages and sends only a very small amount of extra data, during failure-free periods, because most of the information necessary for recovery is already part of the LRC protocol messages. Only one of the LRC protocol messages is modified to include an additional integer.

Periodically, each process takes a checkpoint, independently from the other processes, where it records in stable storage all its state. Unlike other proposals [Janssens 93, Suri 95], our logging algorithm performs no communication-induced accesses to stable

---

Manuel Costa, Nuno Neves and Miguel Castro were supported in part by Praxis XXI fellowships from JNICT.

storage; independent checkpoints are used only to speed up recovery. When a process crashes, the system detects that it crashed and restarts the process on an available processor, using its latest saved checkpoint. The recovering process re-executes with information gathered from the logs of the surviving processes; the non-faulty processes are not forced to rollback after a failure. We assume processes execute in a piecewise deterministic manner [Strom 85] and the only non-deterministic events are shared memory reads. Therefore, the system will recover to a consistent state if the reads of the faulty process can be replayed to return the same values they returned before the crash. This is achieved by providing the recovering process with a sequence of page updates from other processes that is equivalent to the one received before the crash.

The logging algorithm is combined with a consistent checkpointing scheme, which allows multiple failures to be tolerated. It supports the more probable single failures with low overhead, and takes consistent checkpoints infrequently to tolerate the less probable multiple failures. Therefore, the combined algorithm is a kind of two-level recovery scheme as described in [Vaidya 95], where it was shown that this type of algorithm can achieve better performance than one-level recovery schemes. The consistent checkpointing scheme is integrated with the global garbage collection operations already performed by the LRC protocol, hence introducing no additional coordination overhead. During each garbage collection operation, all data structures needed for recovery are discarded and a consistent checkpoint is created. Although consistent checkpointing could also be used to recover from single failures, using the logging and recovery algorithm provides several benefits. Single process faults are recovered efficiently because only the failed process needs to rollback. Some applications can still make progress while one of the processes is recovering, and during recovery only the recovering process consumes resources. Recovery is potentially faster since idle time due to remote synchronization operations is eliminated.

Our implementation required only small changes to the existing TreadMarks implementation of LRC. Code was added to log dependencies and handle recovery. We retained the most significant optimizations used by TreadMarks, such as lazy diff and interval creation [Keleher 94]. In summary, this paper makes the following contributions:

- It describes a new logging and recovery algorithm for lazy release consistent DSM
- It explains how to integrate logging with the memory consistency protocol without sending additional messages, sending only a small amount of extra data, and logging a minimal amount of data
- It presents an implementation of the algorithm using a state-of-the-art LRC-based DSM system
- It presents results showing that the algorithm is very space and time efficient, refuting the common belief that logging overhead is necessarily high for recoverable DSM systems.

The remainder of the paper is organized as follows. The next section presents related work. Section 3 describes TreadMarks and lazy release consistency. Section 4 explains the logging and recovery algorithm. Section 5 presents experimental results showing the overhead introduced by our logging mechanism, and in section 6 we draw our conclusions.

2. Related Work

Several systems have used consistent or independent checkpointing to transparently recover from transient faults. In consistent checkpointing [Chandy 85, Koo 87] processes execute a coordination protocol to establish a new consistent global state. We use consistent checkpointing only to recover from multiple concurrent failures. Cabillic et al. [Cabillic 95] integrate consistent checkpointing with barrier crossings, simplifying the coordination protocol performed by the nodes of the system. We integrate consistent checkpointing with the LRC garbage collection protocol, introducing no additional coordination overhead.

In independent checkpointing [Johnson 87, Strom 85] there is no coordination, but communication-induced checkpoints are performed or some logging has to be done during failure-free periods to prevent rollback propagation during recovery. In sender-based message logging [Johnson 87, Johnson 89], whenever a process sends a message, it logs it in its volatile memory; when the message is received, the receiver returns a sequence number indicating the order of receipt, which the sender adds to its volatile log. The state of a failed process can then be recovered by replaying the logged messages from the distributed log at the surviving processes. We also log recovery information in the volatile memory of sender processes, and use that information to replay messages during recovery. However, message logging protocols are not directly applicable to DSM systems because they do not handle asynchronous message reception [Suri 95] and it has been shown that some potential message passing dependencies can be eliminated in DSM systems [Janssens 94, Janakiraman 94]. Our algorithm takes advantage of this kind of optimization since it is tightly integrated with the DSM consistency protocol and logs only the minimal information required for recovery.

Wu and Fuchs [Wu 90] proposed the first recovery algorithm for sequentially consistent DSM. In their checkpointing scheme, a process is required to save its state whenever a modified page is sent to another process. Janssens and Fuchs [Janssens 93] introduced checkpointing for relaxed memory models, proposing that a node should checkpoint its state whenever it releases or acquires a lock.

Richard and Singhal [Richard III 93] used logging to recover a sequentially consistent DSM from single node failures. In their protocol, shared pages are logged in volatile memory whenever they are read. The log is flushed to stable storage before transferring a modified page to another process. Suri, Janssens and Fuchs [Suri 95] improved this proposal by noting that accesses to shared memory need not be logged but only tracked. For lazy release consistency, they proposed logging the messages received at acquire points and access misses. The log is flushed to stable storage whenever page invalidates or page updates are sent to another process.

Our recovery algorithm for lazy release consistency improves this by logging less data and not requiring communication-induced flushing of the log to stable storage. On the other hand, in our algorithm it is necessary to involve operational processes in the recovery of a failed process. However, operational processes can continue their execution while the failed process is recovering. Our recovery algorithm keeps some of the properties of the protocol previously proposed by Neves, Castro and Guedes [Neves 94]. This protocol was designed to recover DiSOM [Castro 96], a multi-threaded object-based entry consistent DSM system based on an update protocol.

While the protocols described above recover the state of the shared memory as well as the execution state of the processes, some protocols only recover the shared memory state. Stumm and Zhou [Stumm 90] proposed a protocol which tolerates single node failures by ensuring that each page is kept in the memory of at least two nodes of the DSM system. Feeley et al. [Feeley 94] developed a transactional distributed shared memory where coherency is integrated with a mechanism for recoverability of persistent data. The protocol proposed by Kermarrec et al. [Kermarrec 95] establishes a recovery point by creating two recovery copies of every modified page and storing them at distinct nodes. A two-phase commit protocol is used to atomically update a recovery point.

3. LRC and TreadMarks

TreadMarks [Keleher 94] implements a relaxed memory model called lazy release consistency [Keleher 92]. TreadMarks ensures that all programs without data races behave as if they were executing on a conventional sequentially consistent (SC) memory. Most programs satisfy this condition and behave identically in both models, but LRC has the advantage that it can be implemented more efficiently. This section describes TreadMarks' implementation of LRC without the extensions we added to provide fault tolerance.

LRC divides the execution of each process into logical intervals that begin at each synchronization access. Synchronization accesses are classified as *release* or *acquire* accesses. Acquiring a lock is an example of an acquire, and releasing a lock is an example of a release. Waiting on a barrier is modeled as a release followed by an acquire. LRC defines the relation *corresponds* on synchronization accesses as follows: a release access on a lock corresponds to the next acquire on the lock to complete (in real time order); and a release access on a barrier wait corresponds to the acquire accesses executed by all the processes on the same barrier wait.

Intervals are partially ordered according to the transitive closure of the union of the following two relations: (i) intervals on a single process are totally ordered by program order; and (ii) an interval x precedes an interval y, if the release that ends x *corresponds* to the acquire that starts y. The partial order between intervals is represented by assigning a vector timestamp to each interval. TreadMarks implements lazy release consistency by ensuring that if interval x precedes interval y (according to this partial order), all shared memory updates performed during x are visible at the beginning of y.

3.1 Data Structures

Each process in TreadMarks maintains the following data structures in its local volatile memory:

pageArray: array with one entry per shared page.
procArray: array with one list of interval records per process.
dirtyList: identifiers of pages that were modified during the current interval.
VC: local vector clock.
pid: local process identifier.

Each **SharedPageEntry** has fields:
  status: operating system protection status for page (no-access, read-only or read-write)
  twin: original copy of the page
  writeNotices: array with one list of write notice records per process
  manager: identification of the page manager
  copyset: set of processes with copy of the page
Each **WriteNoticeRecord** has fields:
  diff: pointer to diff
  interval: pointer to corresponding interval record
  pageID: page number

Each **IntervalRecord** has fields:

- **idCreat:** id of process which created the interval
- **vc:** vector time of creator
- **writeNotices:** list of write notice records for this interval.

The status field of a page entry is the operating system protection status for the page, i.e. if the status is no-access then any access to the page triggers a page fault, and if the status is read-only a write access to the page triggers a page fault. The writeNotices field in the page entry describes modifications to the page. The entry for process *i* in the writeNotices array contains a list with all the write notices created by *i* for the page, that are known to the local process. Each of these write notice records describes the updates performed by *i* to the page in a given interval. The write notice record contains a pointer to the interval record describing that interval, and a pointer to a diff containing the words of the page that were updated in the interval. The interval records contain a backpointer to a list with one write notice for each page that was modified during the interval. Whenever an interval record is created, it is tagged with the vector time and the identity of its creator.

The *procArray* has an entry for each process. The entry for process *i* contains a list of interval records describing the intervals created by *i* that the local process knows about. This list is ordered by decreasing interval logical times. We refer to the value of VC_i as *i*'s vector time, and to the value of $VC_i[i]$ as *i*'s logical time. Similarly, the vector time of an interval created by *i* is the value of VC_i when the interval is created, and the logical time of the interval is the value of $VC_i[i]$.

3.2 Memory Consistency Algorithm

This subsection describes the implementation of TreadMarks' memory consistency algorithm. The description is based on the pseudo-code for this algorithm presented in Figure 3.1.

LockAcquire is executed when a process tries to acquire a lock and it was not the last process to hold the lock. It sends a request message to the lock manager. The message contains the current vector time of the acquirer. The lock manager forwards the message to the last acquirer.

LockAcquireServer is executed by the releaser when it receives the lock request message. If a new interval has not yet been created since the last local release of this lock, the releaser creates a new interval record for the current interval; and for all the pages modified during the interval it creates write notice records. The diffs encoding the modifications to each of these pages are not created immediately. Instead, they are created lazily when the process receives a diff request or a write notice for a page. The reply message includes a description of all the intervals with timestamps between the acquirer's and the releaser's vector times. We say that an interval *i*, created by process *p*, is between the acquirer's and the releaser's vector times if $VC_{acq}[p] < i.vc[p] \leq VC_{rel}[p]$.

The description of each interval contains the identifier of the process that created the interval, the vector timestamp of the interval, and the corresponding write notices (remember that interval record = [idCreat, vc, writeNotices]). Each write notice in the reply contains only the number of a page that was modified during the interval; no updates are transferred in this message. The information in the reply is obtained by traversing the lists of interval records in *procArray*. In the reply, each sequence of intervals created by process *p* is ordered by increasing logical time of *p*.

The acquirer calls IncorporateIntervals to incorporate the information received in the reply in its own data structures. Notice that the system creates a diff for writable pages for which a write notice was received. This is important to allow the system to distinguish the modifications made by concurrent writers to the page [Carter 91]. The status of the pages for which write notices were received in the reply is set to no-access. When the acquirer tries to access one of the invalidated pages the system invokes the PageFaultHandler. On the first fault for a page, a page copy is requested from the page manager. Pages are kept writable at the manager until the first page request arrives. When this happens, the page status is changed to read-only. After getting the page copy, if there are write notices for the page without the corresponding diffs, the system sends messages requesting those diffs, to the processes that cache them. In TreadMarks, a processor that modified a page in interval *i* is guaranteed to have all the diffs for that page for all intervals preceding *i*. After receiving the diffs, the handler applies them to the page in timestamp order. On a read miss, the system provides read-only access to the page. On a write miss, the system provides read-write access and creates a copy of the page (a “twin”), which is used to detect modifications to the page. The twin is later compared to the current contents of the page to produce a diff that encodes the modifications produced during the interval.

The Barrier routine is executed by the system when a process waits on a barrier. If a process is not the manager of the barrier, it sends to the manager its current vector time, plus all intervals between the logical time of the last local interval known at the manager and its current logical time. After this, the manager sends to each other process all the intervals between the current vector time of the process and the

```
LockAcquire:
send (AcqRequest, VC) to lock manager;
receive (intervals)
IncorporateIntervals(intervals);

LockAcquireServer:
receive (AcqRequest, VC$_{acq}$) from acquirer;
wait until lock is released;
if(an interval was not created since the
   last release of this lock)
   CreateInterval;
send (intervals between VC$_{acq}$ and VC) to
   acquirer;

LockRelease:
if(lock request pending)
   wakeup LockAcquireServer;

CreateInterval:
if(dirtyList is not empty) {
   VC[pid] := VC[pid] + 1;
   insert new interval i in
      procArray[pid];
   for each page in dirtyList
      create write notice record for
      interval i;
   clear dirtyList;
}

IncorporateIntervals(intervals):
for each i in intervals {
   insert record for i in
      procArray[i.idCreat];
   VC[i.idCreat] := i.vc[i.idCreat];
   for each write notice in i {
      store write notice;
      if (twin exists for the page write
               notice refers to){
         if(a write notice corresponding
           to the current writes was not
             already created)
               CreateInterval;
         create diff;
         delete twin;
      }
      set page status to no-access;
   }
}

DiffRequestServer:
receive (DiffRequest, pageId, diffId)
   from req node;
if (diff is not created) {
   create diff;
   delete twin;
   set page status to read-only;
}
send (diff) to req node;
```

```
PageFaultHandler:
if(page status = no-access) {
   if (local page copy not initialized) {
       send (PageRequest, pageId)
             to page manager;
       receive (page copy, copyset);
   }
   send (DiffRequest, pageId, diffId) to
      latest writers;
   receive (diffs) from latest writers;
   apply diffs to page in timestamp order;
}
if (write miss) {
  create twin;
  insert page in dirtyList;
  set page status to read-write;
} else
  set page status to read-only;

PageRequestServer:
receive (PageRequest, pageId) from req
   node;
if (copyset={pid} and
   page status=read-write)
   set page status to read-only;
copyset := copyset ∪ {req};
send (page copy, copyset) to req node;

Barrier:
CreateInterval;
if (not manager) {
   lastLt := logical time of last local
             interval known at manager;
   send (Barrier, VC, local intervals
          between lastLt and VC[pid])
     to manager;
   receive (intervals);
   IncorporateIntervals(intervals);
} else {
   for each client c {
      receive (Barrier, VC$_c$, intervals);
      IncorporateIntervals(intervals);
   }
   for each client c
    send(intervals between VC$_c$ and VC)
      to c;
}
```

Figure 3.1 -TreadMarks memory consistency algorithm.

manager's current vector time.

The storage for the diffs, the write notice records and the interval records is not freed until a garbage collection is performed, i.e. a process effectively maintains a log of all shared memory accesses since the last garbage collection. During garbage collection, all processes synchronize at a barrier. Then each process updates its local copies of shared pages (either the copy is discarded or all diffs are requested). After this, each process sends a message to the garbage collection manager, informing it that garbage collection is complete. After receiving messages from all other processes, the manager replies to each of them. Each process then frees all data structures used by the LRC protocol and resumes normal execution.

4. Recovery Algorithm

4.1 System Model

We consider a lazy-invalidate release consistent DSM system [Keleher 94] composed of a set of nodes strongly connected by an interconnection network. Each node executes exactly one process, which communicates with other processes by exchanging messages. Messages are transmitted reliably by using protocols specific to the operations of the DSM system. Nodes fail independently by halting and all surviving processes detect the failure within bounded time, according to the fail-stop model [Schneider 84]. Processes execute in a piecewise deterministic manner [Strom 85] and the only non-deterministic events are shared memory reads. Processes have no data races.

4.2 New Data Structures

The recovery algorithm uses the existing LRC data structures. In addition, each process maintains in its local volatile memory the following data structures:

sentLog[NPROCS]: array with lists of pairs $<VC_{acq}, VC_{rel}'>$ with the vector times used by the local process *rel* to determine which set of intervals to send to the acquiring process *acq*.

receivedLog[NPROCS]: array with lists of pairs $<VC_{bef}, VC_{aft}>$ with the vector times of the local process before and after incorporating the intervals received at acquire time from other processes.

sent_to_mgrLog[NPROCS]: array with lists of pairs $<lastLt_{mgr}, intLt_{p}>$ with the last logical time of p known to the barrier manager *mgr* and the logical time of the last local interval record of process p.

received_by_mgrLog: list of pairs $<VC_{bef}, VC_{aft}>$, maintained by the manager of the barrier, with the vector times of the local process before and after incorporating the intervals received at the barrier crossing.

The send logs *sentLog[i]* and *sent_to_mgrLog[i]* record information sent by the local process to process i and are necessary to reconstruct the state of process i if process i crashes. The receive logs *receivedLog[i]* and *received_by_mgrLog* record information received by the local process from process i and are necessary to reconstruct the send logs of process i if process i crashes.

4.3 Logging

This subsection describes the logging operations which were added to the base TreadMarks LRC algorithm. Figure 4.1 presents the pseudo-code for this algorithm with fault tolerance extensions in boldface. In Figure 4.1, *VC'* is the approximate vector time of each process. *VC'* is simply a vector time equal to *VC* except that *VC'[pid]* is the logical time of the last local interval record created, which may be smaller than *VC[pid]*. Note that *VC'* is implicitly updated whenever *VC* is updated or a local interval is created.

4.3.1 Logging at Acquires

During an acquire, the releasing process must log the instant at which it received the acquire request and the instant at which the acquiring process issued the request. This way, during recovery, the releaser will be able to inform the recovering process of the set of intervals received during that acquire. Therefore, the acquiring process will invalidate the same pages at the same execution point, which will cause the same read faults to occur and the same pages and diffs to be requested from the surviving processes. The diffs that will be received are the same, because the LRC protocol keeps these diffs in the memory of the processes which sent them. The pages that will be received may not be exactly the same that were received before the fault, but our algorithm guarantees that they differ only in the parts that will not be accessed by the process.

Each acquire operation is uniquely identified by the logical time of the process at the time the acquire is performed. In the base LRC protocol, incrementing the logical time is delayed until there is communication with other processes, to optimize the case when the lock is re-acquired by the same process. In the fault tolerant LRC protocol, the logical time is incremented at every acquire because during replay it is necessary to uniquely identify remote acquire operations. The logical time increment cannot be associated with interval record creation because interval creation can be asynchronous (in LockAcquireServer). During re-execution, it would not be possible to replay these asynchronous increments and therefore acquires would not be identifiable by logical time. The logical time of the process is also incremented at every release because every time a process executes a release a new logical interval begins. If an interval record is created for this new interval, its logical time must be different from the logical times of previous intervals. For the same reason, the logical time is also incremented at every barrier.

At each remote acquire, the releasing process creates a new entry in the *sentLog*, where it saves its approximate vector time, VC_{rel}', and the acquirer's vector time, VC_{acq}. The intervals between these vector

LockAcquire:

```
VC[pid] := VC[pid] + 1;
$VC_{bef}$ := VC;
send (AcqRequest, VC) to lock manager;
receive (intervals)
IncorporateIntervals(intervals);
log <$VC_{bef}$,VC> in receivedLog[$pid_{rel}$];
```

LockAcquireServer:

```
receive (AcqRequest, $VC_{acq}$) from acquirer;
wait until lock is released;
if(an interval was not created since the
   last release of this lock)
   CreateInterval;
log <$VC_{acq}$,VC'> in sentLog[$pid_{acq}$];
send (intervals between $VC_{acq}$ and VC') to
   acquirer;
```

LockRelease:

```
VC[pid] := VC[pid] + 1;
if(lock request pending)
   wakeup LockAcquireServer;
```

CreateInterval:

```
if(dirtyList is not empty) {
   -- removed logical time increment --
   insert new interval i in
      procArray[pid];
   for each page in dirtyList
         create write notice record for
        interval i;
   clear dirtyList;
}
```

IncorporateIntervals(intervals):

```
for each i in intervals {
   insert record for i in
      procArray[i.idCreat];
   VC[i.idCreat] := i.vc[i.idCreat];

   for each write notice in i {
      store write notice;
      if (twin exists for the page write
          notice refers to) {
          if(a write notice corresponding
            to the current writes was not
            already created)
                 CreateInterval;
         create diff tagged with VC[pid];
         delete twin;
      }
      set page status to no-access;
   }
}
```

DiffRequestServer:

```
receive (DiffRequest, pageId, diffId)
   from req node;
if (diff is not created) {
   create diff tagged with VC[pid];
   delete twin;
   set page status to read-only;
}
send (diff) to req node;
```

PageFaultHandler:

```
if(page status = no-access) {
   if (local page copy not initialized) {
       if(first GC done) {
           send (PageRequest, pageId)
              to page manager;
           receive (page copy, copyset);
       } else
           zero-fill local page copy;
   }
   send (DiffRequest, pageId, diffId) to
      latest writers;
   receive (diffs) from latest writers;
   apply diffs to page in timestamp order;
}
if (write miss) {
  create twin;
  insert page in dirtyList;
  set page status to read-write;
} else
  set page status to read-only;
```

PageRequestServer:

```
receive (PageRequest, pageId) from req
   node;
--removed possible page status change--
if (req $\notin$ copyset) {
   copyset := copyset $\cup$ {req};
   send (page copy, copyset) to
      req node;
} else {
   send (page copy from last consistent
   checkpoint, copyset) to req node;
}
```

Barrier:

```
VC[pid] := VC[pid] + 1;
CreateInterval;
if (not manager) {
   lastLt := logical time of last local
             interval known at manager;
   $VC_{bef}$ := VC;
   log <lastLt, VC'[pid]> in
      sent_to_mgrLog[$pid_{manager}$];
   send(Barrier, VC, local intervals
        between lastLt and VC'[pid])
      to manager;
   receive (intervals);
   IncorporateIntervals(intervals);
   log <$VC_{bef}$,VC> in
      receivedLog[$pid_{manager}$];
} else {
   $VC_{bef}$ := VC;
   for each client c {
      receive (Barrier, $VC_c$, intervals);
      IncorporateIntervals(intervals);
   }
   log <$VC_{bef}$,VC> in
      received_by_mgrLog;
   for each client c {
     log <$VC_c$,VC'> in sentLog[c];
     send(intervals between $VC_c$ and VC')
       to c;
   }
}
```

Figure 4.1 -TreadMarks memory consistency algorithm with fault tolerance extensions.

times can be obtained from *procArray*. During recovery, these intervals are sent to the recovering process to replay the acquire operation. VC_{rel}' is logged instead of VC_{rel} because when the acquire request is serviced, an interval record with logical time $VC_{rel}[pid]$ might not have been created. If later such an interval record is created, it is necessary to guarantee that the corresponding write notices are not sent to the acquirer during replay of the acquire operation.

The releaser does not need to log the whole vector times; it suffices to log the positions where $VC_{rel}'[i] > VC_{acq}[i]$. We currently do not use this optimization as its benefits would be minimal for our number of processors, but it could be used to compress the log if very large vector clocks were used.

After a failure, it is also necessary to recover the *sentLog* to tolerate subsequent faults in other processes that received write notices from the recovering process. After an acquire operation, the acquiring process records in its *receivedLog* its vector times from before and after incorporating the intervals. Although this might not be an exact copy of the log entry created by the releaser, it contains information about which intervals were received, because the positions where $VC_{aft}[i] > VC_{bef}[i]$ have the same values as the positions where $VC_{rel}'[i] > VC_{acq}[i]$ in the log of the releaser. During recovery, the processes that received intervals from the recovering process send it these log entries, so that the recovering process can recover its *sentLog*.

4.3.2 Logging at Barriers

During barrier crossing, the manager logs in its *sentLog* the vector times received from each process and its approximate vector time, *VC'*, at the time of reply. If a non-manager process is recovering, this information allows the manager to re-send exactly the same intervals as before the failure.

Each non-manager process logs in its *sent_to_mgrLog* a tuple *<lastLt, intLt>* with the logical time of the last local interval that is known at the manager and the logical time of the last local interval record created. This information is used to calculate the local intervals sent to the manager, but it does not uniquely identify the barrier crossing at which they were sent. However this information is implicit in the position of the tuple in the *sent_to_mgrLog*, because barrier crossings are global operations performed by every process and barrier managers are statically assigned.

When a process receives the reply from the manager, it logs in the *receivedLog* its vector times before and after incorporating the received intervals. This is exactly the same logging operation that is performed during lock acquires. To recover the information in the *sent_to_mgrLog* of failed processes, the manager logs in the *received_by_mgrLog* its vector times before and after incorporating the intervals at the barrier. This entry keeps the same information kept by the set of all corresponding *sent_to_mgrLog* entries at non-manager processes.

4.3.3 Page Logging

On the first access to a page, the LRC protocol requests an initial page copy from the page manager. This copy may have writes which do not have corresponding diffs, for two reasons. Since pages are initially kept writable at the manager, it may perform some writes which are not captured in diffs. Also, immediately after each GC operation, all the diffs are discarded and writes are kept only in page copies.

Suppose that some process requests a page copy and reads the value of one of these writes. Suppose later the process fails and is re-started. During recovery, the recovering process will request the same page copy. However, the manager cannot send its current copy because it may have writes that follow (according to the LRC partial order) the accesses the recovering process is about to perform. These writes possibly overwrite the values which the recovering process read during normal execution. Figure 4.2 exemplifies this problem[1]. Suppose P_1 is the manager of some page and no other process requested a copy of that page. The page is writable when w(1)3 is performed, and hence no page fault is generated and a twin is not created. Later P_2 acquires lock i and initiates a r(1) access. This will trigger a page fault and a page request is sent to P_1. Upon receipt of the reply, P_2 completes the r(1)3 access and performs w(1)1. Later P_1 acquires lock i and performs r(1). Hence the diff corresponding to w(1)1 will be requested from P_2 and applied to the local page copy. If P_2 fails, during recovery it will request the same page copy from P_1. If P_1 replies with its current copy, the r(1) access at P_2 will return 1, and thus P_2 will not reach the same state.

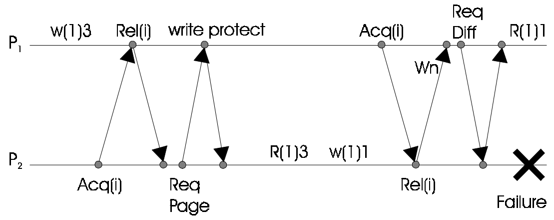

Figure 4.2 - The problem of sending current page copies during process recovery.

---

[1] r(x)y is used to denote a read operation from address x returning value y. w(x)y denotes a write of value y to address x. Wn is used to denote a set of write notices.

To avoid maintaining two copies of each shared page in the memory of page managers, we introduced the following modifications. When processes are started, all their shared pages are write protected; this includes page managers. When the first fault occurs for a page, it is initialized to all zeros. Thus all processes start from the same page contents. These modifications cause at most one additional page fault and diff to be created for each page. Until the first global garbage collection operation is executed, there is no need to issue page requests, as all writes have corresponding diffs. After the first global GC, all shared pages are saved in the consistent checkpoint of all the processes. After the first global GC, if a process receives a page request from a process which does not yet belong to the copyset of the page, it returns its current copy. If, on the other hand, the requesting process already belongs to the copyset, the manager must fetch the initial page copy from the last global checkpoint and return that copy.

With this algorithm, the page copies received during normal execution and recovery can be different, since during normal execution the pages may contain writes which do not precede the local accesses. However, this poses no problem as it is guaranteed that a process with no data races will not read these values and will therefore re-execute as before the failure.

4.3.4 Termination Related Logging

When a diff is created during asynchronous handling of a diff request, it can contain writes which were performed immediately before the request arrived. This means that recovery must end after these writes are performed, if the system is to be in a consistent state. Figure 4.3 shows an example where process P_1 acquires lock m from process P_2. During this acquire, P_1 receives a write notice for the page where w(0)1 was performed. P_2 then locally acquires lock b and performs w(2)4 on the same page. When process P_1 requests the diff for the write notice it received, P_2 write protects the page where the writes were performed and creates the diff. The diff contains both w(0)1 and w(2)4.

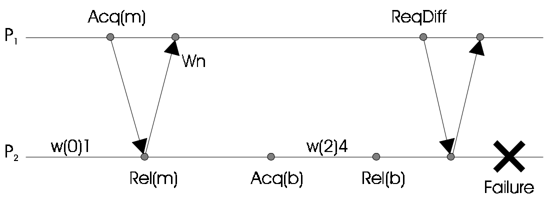

Figure 4.3 - Diff creation due to asynchronous message reception which keeps write done immediately before the request arrives.

Suppose recovery of P_2 ends at the Rel(m). If process P_1 acquires lock b and performs r(2), it will get the value 4, which was not written by any process.

To guarantee that recovery ends after the last diff creation, it is necessary to tag each diff with the logical time of the process where it was created. During recovery, when the logical time of the process becomes greater than the logical time of the last diff, it is guaranteed that all writes sent in diffs are performed.

Recovery must also terminate only when all intervals that were sent to other processes are finished. If this did not happen, then after recovery some processes would have intervals from the faulty process with logical times greater than the current logical time of the recovered process. These intervals would not be replayed and eventually intervals with the same logical times would be created but with possibly different write notices. This requirement does not introduce any new logging operation. Although interval record creation can be asynchronous, each interval is uniquely identified by its logical time, and during replay, the process will re-execute until the last interval sent to other processes is finished.

4.4 Checkpointing

Periodically, the processes can take independent checkpoints to optimize recovery time. When one of these checkpoints is completed, its previous independent checkpoint can be discarded.

Several optimizations can be used to speed up independent checkpointing. Initially all shared pages are excluded from the checkpoints. When there is a write fault, the corresponding page is included in the next checkpoint. When a page is invalidated, it is excluded from the next checkpoint, because its state can be recovered from the corresponding diffs. This is different from incremental checkpointing because the page may be written after one checkpoint and still need not be included in the next one. The diffs received from other processes are not included in the checkpoints because they can be easily recovered from the processes where they were created. The diffs created locally and sent to other processes also need not be included in the checkpoints because they can be recovered from the processes that received them.

During the global garbage collection operations already performed by the TreadMarks LRC protocol, a consistent global checkpoint is created, and the LRC data structures used for recovery are discarded. As processes already have to coordinate to perform GC, there is no additional coordination overhead to establish the global checkpoint. This consistent checkpointing mechanism is used to efficiently tolerate multiple faults.

If multiple faults are detected, all the processes are rolled back to the last consistent checkpoint.

The consistent checkpointing and garbage collection protocol proceeds as follows. First, all processes synchronize at a barrier. Then each process updates its local page copies, as in the normal GC protocol. Each process then sends a message to a checkpoint server requesting to save its checkpoint and waits for the reply. After having received and saved all the checkpoints, the checkpoint server sends acknowledgment messages to all processes. Each process then frees all entries of the send and receive logs (and all LRC data structures). Afterwards normal execution is resumed. This is exactly the protocol performed during LRC GC; the only additional overhead is due to checkpoint transmission and storage.

The incorporation of consistent checkpointing with garbage collection does not permit increasing the checkpointing interval. However GC operations must be relatively infrequent to not affect the performance of the DSM system. Therefore, increasing the amount of memory for the LRC data structures improves the DSM performance by reducing the frequency of both GC and consistent checkpointing.

Usual optimizations such as incremental and non-blocking checkpointing [Elnozahy 92, Li 90] can be used to reduce checkpoint overhead. The performance of copy-on-write checkpointing might be affected by the low amount of free memory at GC time. However, performing GC at a lower occupied memory threshold would solve this problem. If non-blocking checkpointing is used, single failures while the checkpoints are being saved would cause the system to rollback to the last consistent checkpoint. To avoid this, the LRC data structures needed for recovery could be freed only after the checkpoints are saved on stable storage. This would also require GC to be done at a lower occupied memory threshold, because processes would need memory to continue execution while the checkpoints are being saved.

4.5 Recovery

In order to simplify the following presentation, we do not refer to barrier crossings but only to lock acquires. Discussion of recovery for barriers is deferred until the next section.

4.5.1 Data Collection

When a process failure is detected, a new process is started from the most recent checkpoint of the failed process, *p*. The recovering process logically broadcasts a message informing it is in recovery. This message contains the current logical time of the process, T_{ckpt}, which is the one saved in the checkpoint file, and the logical time of the last local interval record created, *localIntLt*. All surviving processes which have $VC[p] > localIntLt$ reply by sending their vector times and the identifiers of the diffs created by the recovering process, which are kept in their local memories.

The recovering process then requests all its intervals with a logical time greater than *localIntLt* from the process which has the largest $VC[p]$. All diffs generated by the process after or at its current logical time are also requested from the processes that cache them. The process then builds a list of all the intervals it had created during the failure-free period. This list, ordered by descending logical time of the recovering process, is prepended to *procArray[p]*.

The recovering process also sends to each other process, *c*, the $VC_{acq}[c]$ of its last entry in *sentLog[c]*. Each process replies by sending its *receivedLog[p]* entries where $VC_{bef}[c]$ is greater than the received $VC_{acq}[c]$. These entries are appended to the *sentLog[c]* of the recovering process.

Each process also sends the identifiers of the pages for which it has a local copy and that are managed by the recovering process. This enables the recovery of the copyset information for each of these pages.

Each process, *c*, also sends a list of tuples $<VC_{acq}, VC_{rel}', intervalSet>$ corresponding to its *sentLog[p]* entries where $VC_{acq}[p] > T_{ckpt}$, where *intervalSet* is the sequence of intervals received during that acquire. The list of the prefixes $<VC_{acq}, VC_{rel}'>$ of these tuples is appended to the *receivedLog[c]* of the recovering process. All the lists received are merged into a single list, *acquireList*, of tuples $<lt, intervalSet>$, where *lt* is the logical time of the acquire at the recovering process. This list is ordered by ascending *lt*.

4.5.2 Execution Replay

After the data collection phase described above, the process starts to re-execute. During execution replay, requests from other processes are blocked and the page fault handler is the same as for the normal execution.

During the first phase of recovery, each remote acquire operation is replayed. The next remote acquire is identified by the logical time of the next entry in *acquireList*. The acquire replay is the same as the normal acquire, except that no local intervals are created since they were already recovered. The intervals from other processes are prepended to the *procArray* lists corresponding to the processes which created them.

There is however a case when the local data structures created during an acquire operation are affected by asynchronous message handling and these data structures are not recoverable from other processes. Figure 4.4 shows an example of this, when a diff is created during an acquire operation where a write notice is received for a writable page. In Figure 4.4, all writes are performed on the same page.

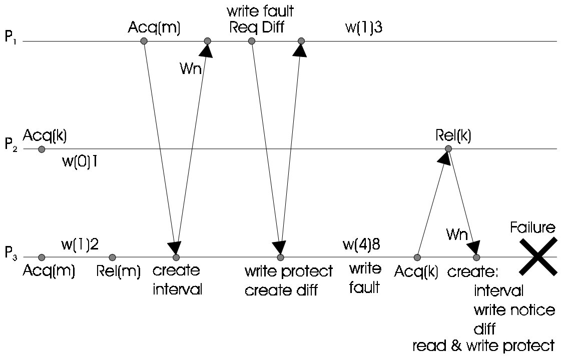

Figure 4.4 - Creation of a diff which is not sent to other processes, during an acquire operation.

If process P_3 fails and is re-started, the asynchronous reception of the lock and diff requests, sent by P_1, cannot be replayed. If the diff created during the last acquire of P_3 is not later sent to other processes, it must be recovered during replay of the acquire operation. The diff created during recovery cannot have all the writes that were performed since the last write invalidate. For instance, in the example of Figure 4.4, w(1)2 and w(4)8 cannot both belong to the diff for the write notice created in the last acquire, because w(1)2 precedes w(1)3 while w(4)8 is concurrent with w(1)3.

To solve this problem, when a process replays an acquire operation, a diff is created if a received interval, *r*, has a write notice for a writable page, and there is a local interval, *i*, with a write notice with no diff for the same page, such that *i.vc[r.idCreat]<r.vc[r.idCreat]*.

To create the diff, all diffs that correspond to write notices for the same page, which belong to intervals that follow the last acquire where the page was invalidated, are applied to the current twin. Only then is the new diff between the page and its twin calculated. This diff is associated with the write notice without diff. During acquire operations where it is not necessary to create a diff, the page twin is simply discarded.

The first phase of recovery proceeds until one of these situations occurs:

- There are no more entries in *acquireList,* and the logical time of the recovering process, *VC[p]*, becomes greater than both the logical time of the first interval in *procArray[p]* (the last interval that was sent to another process) and the logical time of creation of the last diff created by *p* during the failure-free execution.
- The logical time of the recovering process, *VC[p]*, is greater than the logical time of the first interval in *procArray[p]* and the process is about to re-execute the next acquire in *acquireList*.

When the first phase of recovery ends, the recovering process applies to the twin of each writable page all the diffs corresponding to write notices for the page which belong to intervals that follow the last acquire where a write notice was received for that page, and whose logical time of creation is smaller than the current logical time. If the last of these write notices does not have a diff or if the twin does not become equal to the page, it keeps the page writable. Otherwise it write invalidates the page. The *dirtyList* keeps only the writable pages whose last write notice has a diff or does not exist. This page state recovery procedure recovers the state of shared pages which can be affected by asynchronous handling of diff requests, which can in turn correspond to write notices for intervals created during asynchronous handling of acquire requests.

If after the first phase of recovery there are no more entries in *acquireList*, recovery terminates. Otherwise, replay of remote acquire operations must continue. However, after this phase, the intervals created were not sent to other processes and therefore acquire operations are executed as during normal execution. This is another situation where a LRC protocol data structure is created but is not transmitted to another process. To recreate these data structures the following procedure is used. Immediately before each of the subsequent remote acquire operations, but after having incremented the logical time of the process for that acquire, the recovering process executes the same page state recovery procedure as was executed at the end of the first phase of recovery. Next, the acquire operation is re-executed as during the failure-free period, creating local intervals if needed. Re-execution continues until there are no more entries in *acquireList*. At this point, if the logical time of the recovering process is greater than the logical time of creation of the last diff created during the failure-free execution, recovery terminates. Otherwise re-execution continues until the logical time becomes greater than the time of creation of the last diff. The recovering process then executes the same page state recovery procedure as was executed at the end of the first phase of recovery, after which recovery terminates. At this point, all intervals that were sent during failure-free operation are created and all writes sent in diffs are performed.

One straightforward extension to the algorithm, to allow faster recovery, consists of logging which diffs and pages were requested by each process. This allows the recovering process to prefetch all the diffs and pages needed for recovery. This would provide an interesting benefit. As the processes would not need to communicate with other processes during recovery except for the initial information gathering phase, immediately after this phase other failures could be tolerated.

4.6 Recovery for Barriers

For barrier crossings at barriers that are not managed by the recovering process, recovery proceeds exactly as for lock acquires. There are, however, some differences in the replay of barrier crossings when the process is the manager of the barrier. To replay these operations, when a process *p* fails, it sends the number *n* of entries in its *received_by_mgrLog* in its initial broadcast message. Each process then sends a list[2] of tuples *<intervalSet>* corresponding to its intervals between the logical times in its *<lastLt, intLt>* entries in $sent\_to\_mgrLog[p]_m$ where $m>n$.

The manager then builds a list, *barrierList*, where each $barrierList_i$ entry is an interval set calculated as the union of the ith entries in all the lists of intervals sent by each other process. Each entry in *barrierList* keeps all the intervals received by the manager at the crossing of one of the barriers it manages. Therefore these entries are used to supply the same write notices to the manager during replay of these crossings. The *received_by_mgrLog* is recovered incrementally as each barrier crossing is replayed.

To recover its *sent_to_mgrLog*, the process also sends in its initial broadcast message the number of entries, k_q, in each *sent_to_mgrLog[q]* list. Each process, *q*, then selects from all its $received\_by\_mgrLog_l$ entries, where $l>k_q$, the logical times corresponding to the recovering process in the pair $<VC_{bef}, VC_{aft}>$. It then sends to the recovering process a list of all the selected *<lastLt, intLt>* pairs. This list is then appended to the *sent_to_mgrLog[q]* list at the recovering process.

5. Experimental Results

5.1 Implementation

We implemented the logging part of our algorithm on top of TreadMarks as described in section 4.3; and we used the libckpt library [Plank 95] to create the process checkpoints. This library implements incremental and non-blocking checkpointing with copy-on-write. However, incremental checkpointing was disabled in our experiments because it write protects all data pages after each checkpoint, and this would interfere with TreadMarks' management of page protections. Currently, pages which are never invalidated are written in every checkpoint. We used the facilities provided by libckpt to include and exclude memory from the checkpoints, to manage the portion of the shared address space which is checkpointed.

The recovery part of our algorithm is partially implemented. At the end of a run, all processes wait for the re-execution of one process. This process rolls back to the start of the application, gathers the recovery information needed to build the *acquireList* (section 4.5.1), and re-executes its portion of the computation.

In order to evaluate our algorithm, we compare it with the logging algorithm proposed in [Suri 95] for LRC based systems. We implemented the logging algorithm of [Suri 95] on top of TreadMarks. This algorithm logs references to the diffs obtained from other processes and copies of the messages received at lock acquires and barrier crossings. When local diffs or intervals are transmitted to other processes, the log is synchronously flushed to a local disk in each workstation.

5.2 Experimental Setup

Our measurements were performed on a network of SPARCstations 10/30 running SunOS 4.1.3, connected by a 10Mbps Ethernet network. Each SPARCstation has a V8 SuperSPARC processor running at 36MHz, 36 Kbytes of internal cache and 32 Mbytes of memory. Checkpoints were written to a Seagate Barracuda disk via NFS. Our results are based on the execution of three parallel applications included in the TreadMarks distribution:

SOR: An implementation of the red-black successive over-relaxation algorithm [Keleher 94].

TSP: Solves the traveling salesperson problem, using a branch and bound algorithm [Carter 91].

Water: An N-body molecular dynamics simulation from the SPLASH benchmark suite [Singh 91], which calculates forces and potentials in a system of water molecules in liquid state.

We ran two sets of experiments. First, a series of experiments was performed to evaluate the overheads of our logging algorithm and the one presented in [Suri 95] when the number of processes running the application is increased. For this first set of benchmarks, SOR was run for 318 iterations on a 1024x1024 floating point matrix; TSP was run with a graph of 20 cities; and Water was run for 15 steps on 500 molecules.

The second set of experiments measures our logging algorithm in 4-processor runs, with and without checkpoint creation. For the second set of experiments, SOR was run for 1400 iterations on a 1278x2048 floating point matrix; TSP was run with a graph of 22 cities; and Water was run for 23 steps on 1472 molecules. No global garbage collections or consistent checkpoints were performed during the experiments.

---

[2] The ith element in list L is denoted by L_i.

In each series of experiments, the logging algorithms are compared to the TreadMarks version modified to not issue page requests before the first GC operation. We found that this modification had a very small positive impact on the performance of applications, because it reduces the amount of data transferred. Therefore, we chose to isolate this effect from the logging overhead.

5.3 Results

In the following figures, VC Logging (Vector Clock Logging) is used to label the results of our logging algorithm, while Message Logging is used to label the results of our implementation of the logging algorithm proposed in [Suri 95].

Figures 5.1, 5.2 and 5.3 show the running time for our three test applications in the first set of experiments; and Figure 5.4 shows the log sizes for each application in 4-processor runs. As expected, adding our logging algorithm to TreadMarks almost does not affect application execution time. The graphs show that the logging mechanism introduces almost zero time overhead. This happens because it does not send extra messages, does not access stable storage, and logs only a small amount of data. Figure 5.4 shows that the space overhead of our algorithm is also very small.

On the other hand, Message Logging introduces a significant execution time overhead, and the overhead grows with the number of processes. The main causes of this overhead are the accesses to stable storage and the large amount of data logged. The Message Logging approach flushes the log to disk with a synchronous write every time local diffs or intervals are sent to another process. The Message Logging approach logs diffs and write notices, whereas the VC Logging approach only logs pairs of vector times. Therefore, the Message Logging approach has significantly larger logs. Note that the Message Logging approach keeps most of the log on disk; only the portion of the log created since the last flush to disk is kept in memory.

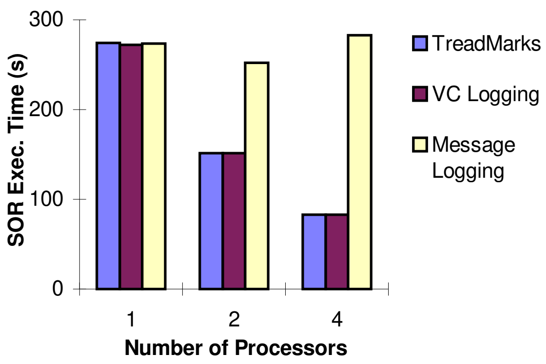

Figure 5.1 - Execution times for SOR.

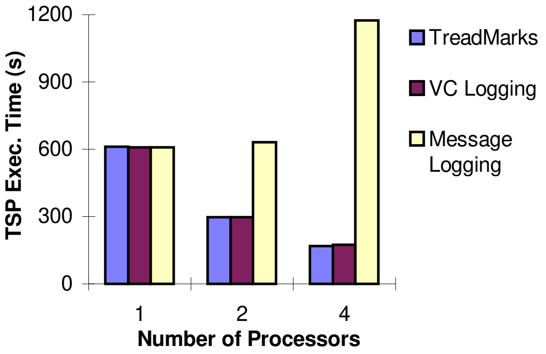

Figure 5.2 - Execution times for TSP.

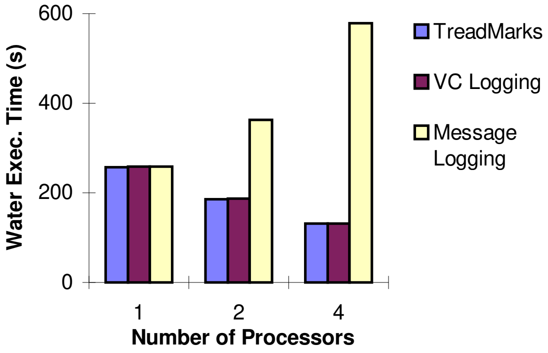

Figure 5.3 - Execution times for Water.

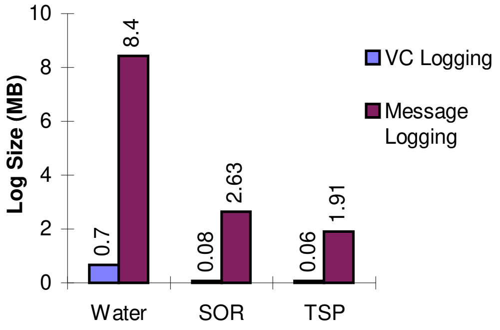

Figure 5.4 - Log sizes for first set of benchmarks.

For VC Logging, the log sizes of the applications are dictated by the number of remote synchronization operations, i.e. the number of times a lock is acquired from a remote process and the number of barrier crossings. Water uses significantly more log space than the other two applications, because it has an order of magnitude more remote synchronization operations. For Water, the *sentLog* and *receivedLog* keep 22073 VC pairs each, for a total of 353168 bytes each. For TSP, 1981 VC pairs are kept in *sentLog* and *receivedLog*. For Water and TSP most of the synchronization is done using locks, and therefore the log components related to logging at barrier managers are not significant. These components are more significant in SOR because most of the synchronization is done using barriers. SOR keeps 1914 VC pairs in *sentLog* and *receivedLog*, 1914 logical time pairs in *sent_to_mgrLog*, and 638 VC pairs in *received_by_mgrLog*.

Figure 5.5 shows the running time of our three test applications in 4-processor runs in the second set of experiments. Six checkpoints were created for each process, during the execution of Water and TSP. For SOR, only five checkpoints were created. Log sizes and average checkpoint sizes for these runs are presented in Table 5.1. These results confirm that our logging algorithm has near zero time overhead and small space overhead.

The overhead of checkpointing is below 2% for Water and TSP. For SOR, the checkpointing overhead is approximately 22%, because the average checkpoint size in SOR is much larger than for the other two applications (as shown in Table 5.1). SOR has a larger checkpoint size because each process touches significantly more memory than in the other two applications. This inefficiency can be reduced by integrating incremental checkpointing with the LRC protocol and using modern switched network technologies.

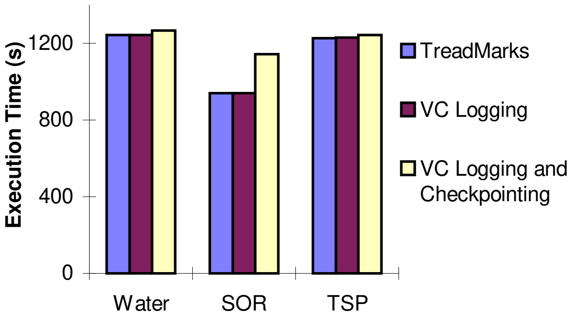

Figure 5.5 - Execution times for the second set of benchmarks.

Application	Log Size (MB)	Average Checkpoint Size (MB)
Water	3.10	3.05
SOR	0.33	7.84
TSP	0.05	2.49

Table 5.1 -Log sizes and average checkpoint sizes for benchmarks of Figure 5.5.

Figure 5.6 shows the breakdown of normal execution and recovery times for the first set of benchmarks, in 4-processor runs. Re-execution time is 72% of normal execution time for Water, 75% for SOR and 95% for TSP.

Two effects combine to make recovery faster than normal execution. First, idle time is reduced. Idle time results from waiting for locks and barriers and from remote communication latency. During recovery, the first component is completely eliminated and the second is reduced, because the recovering process is the only one using the network. Second, Unix overhead is reduced, because the number of messages sent and received is lower ([Keleher 94] shows that at least 80% of the kernel execution time is spent in the communications routines).

Water has a high remote synchronization rate, resulting in a large amount of idle time during normal execution. During recovery, idle time resulting from this activity is greatly reduced. SOR has a lower synchronization rate, but processes synchronize with barriers and initiate data transfers at approximately the same times, which results in idle time waiting for network availability. TSP has a very small amount of idle time, and therefore its recovery time is very close to its normal execution time.

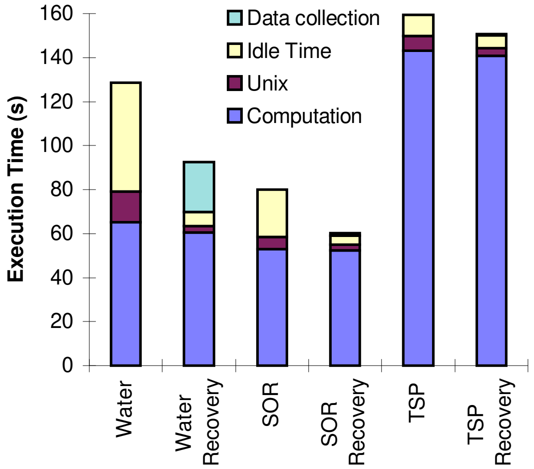

Figure 5.6 - Execution time breakdown for normal execution and recovery of the first set of benchmarks.

6. Conclusions

This paper presented a lightweight logging algorithm for lazy release consistent distributed shared memory. This algorithm handles recovery from single node failures, one of the most common failure scenarios in networks of workstations. Furthermore, the algorithm is integrated with a consistent checkpointing scheme, which allows multiple failures to be tolerated.

The logging algorithm is very efficient, because it is tightly integrated with the LRC memory coherence protocol. Our experiments show that the logging overhead for recovery of lazy release consistent DSM is very low, both in time and space.

Acknowledgments

The authors would like to thank the anonymous reviewers, Willy Zwaenepoel, Pete Keleher and especially our shepherd David Johnson for the comments which helped to improve the contents of this paper.

References

[Anderson 95] T. E. Anderson, D. E. Culler and D. A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[Cabillic 95] G. Cabillic, G. Muller and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Sytems. *Proceedings of the 14th Symposium on Reliable Distributed Systems*, September 1995.

[Carter 91] J. B. Carter, J. K. Bennett and W. Zwaenepoel. Implementation and performance of Munin. *Proceedings of the 13th Symposium on Operating Systems Principles*, pp. 152-164, October 1991.

[Castro 96] M. Castro, P. Guedes, M. Sequeira and M. Costa. Efficient and Flexible Object Sharing. *Proceedings of the 25th International Conference on Parallel Processing*, August 1996.

[Chandy 85] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63-75, February 1985.

[Elnozahy 92] E. N. Elnozahy, D. B. Johnson and W. Zwaenepoel. The Performance of Consistent Checkpointing. *Eleventh IEEE Symposium on Reliable Distributed Systems*, pp. 39-47, October 1992.

[Feeley 94] M. J. Feeley, J. S. Chase, V. R. Narasayya and H. M. Levy. Integrating Coherency and Recoverability in Distributed Systems. *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[Janakiraman 94] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. *Proceedings of the 13th Symposium on Reliable Distributed Systems*. pp. 42-51, October 1994.

[Janssens 93] B. Janssens and W. K. Fuchs. Relaxing Consistency in Recoverable Distributed Shared Memory. *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing: Digest of Papers*, pp. 155-163, June 1993.

[Janssens 94] B. Janssens and W. K. Fuchs. Reducing Interprocess Dependence in Recoverable Distributed Shared Memory. *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pp. 34-41, October 1994.

[Johnson 87] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing: Digest of Papers*, pp. 14-19, July 1987.

[Johnson 89] D. B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University, December 1989.

[Keleher 92] P. Keleher, A. Cox and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *Proceedings of the 19th Annual Symposium on Computer Architecture*, pp. 13-21, May 1992.

[Keleher 94] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proceedings of the 1994 Winter USENIX Conference*, January 1994.

[Kermarrec 95] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin and I. Puaut. A recoverable distributed shared memory integrating coherency and recoverability. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems*, Pasadena, CA, June 1995.

[Koo 87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23-31, January 1987.

[Li 86] K. Li. *Shared Virtual Memory on Loosely Coupled Microprocessors*. PhD thesis, Yale University, September 1986.

[Li 90] K. Li, J. F. Naughton and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pp. 79-88, March 1990.

[Neves 94] N. Neves, M. Castro and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. *Proceedings of the Thirteenth Annual Symposium on Principles of Distributed Computing*, pp. 121-129, August 1994.

[Plank 95] J.S. Plank, M. Beck, G. Kingsley and K. Li. Libckpt: Transparent Checkpointing under Unix. *Proceedings of the USENIX Winter 1995 Technical Conference*. January 1995.

[Richard III 93] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pp. 86-95, October 1993.

[Schneider 84] F. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145-154, May 1984.

[Singh 91] J. Singh, W. Weber and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

[Strom 85] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204-226, August 1985.

[Stumm 90] M. Stumm and S. Zhou. Fault Tolerant Distributed Shared Memory Algorithms. *Proceedings of the 2nd Symposium on Parallel and Distributed Processing*, pp. 719-724, December 1990.

[Suri 95] G. Suri, B. Janssens and W. K. Fuchs. Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory. *Proceedings of the 25th Annual Intenational Symposium on Fault-Tolerant Computing: Digest of Papers*, pp. 279-288, June 1995.

[Vaidya 95] N. H. Vaidya. A Case for Two-Level Distributed Recovery Schemes. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 64-73, May 1995.

[Wu 90] K. Wu and W. K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 460-469, April 1990.