Limited-size Logging for Fault-Tolerant Distributed Shared Memory with Independent Checkpointing

Florin Sultan, Thu Nguyen, Liviu Iftode
Department of Computer Science
Rutgers University, New Jersey, Piscataway, NJ 08854-8019
{*sultan, tdnguyen, iftode*}*@cs.rutgers.edu*

Department of Computer Science Technical Report DCS-TR-409

February 2000

Supported in part by Rutgers University Information Sciences Council Pilot Projects Program and by a USENIX Research Grant.

Contents

Abstract

This paper presents a fault tolerance algorithm for a home-based lazy release consistency distributed shared memory (DSM) system based on volatile logging and independent checkpointing.

The proposed approach targets large-scale distributed shared-memory computing on local-area clusters of computers as well as collaborative shared-memory applications on wide-area meta-clusters over the Internet. The challenge in building such systems lies in controlling the size of the logs and to garbage collect the unnecessary checkpoints in the absence of global coordination.

In this paper we define a set of rules for lazy log trimming (LLT) and checkpoint garbage collection (CGC) and prove that they do not affect the recoverability of the system. We have implemented our logging algorithm in a home-based DSM system and showed on three representative applications that our scheme effectively bounds the size of the logs and the number of checkpointed page versions kept in stable storage.

1 Introduction

We present the design of a scalable fault-tolerant protocol based on rollback recovery targeted towards both today's local-area clusters as well as tomorrow's wide-area meta-clusters. Rollback recovery is a well-known fault tolerance technique where a computation's state is saved to stable storage (checkpointing); this saved state is then used to restart the computation in the case of a failure. For distributed computations, there are two options for checkpointing [10]: *coordinated checkpointing*, where all processes coordinate to save a globally consistent state at each checkpoint, and *independent checkpointing*, where checkpointing is purely a local operation and, as a result, a set of local checkpoints does not represent a globally consistent state of the system. Coordinated checkpointing is particularly efficient for small-scale clusters and simpler to implement if non-failed nodes are assumed to always be accessible (no temporary disconnection in communication) [11].

Traditionally, it has been reasonable to assume that clusters are relatively small and all non-failed nodes are always connected. This traditional picture is changing rapidly, however. First, high-performance, yet commodity, communication technologies have brought clusters closer to delivering scalable super-computing performance. As a result, very large clusters are being built to support long running, intensive applications such as data mining and parallel scientific workloads. Second, the explosion of web-based applications and the increasing demand for the scalable computing infrastructure to support them are bringing meta-clustering (clusters of local-area clusters connected by the Internet) to the horizon [36, 15, 12]. These trends make fault tolerance today as important as, if not more important than, performance. Furthermore, the evolution of meta-clusters changes the basic underlying assumptions: clustering protocols must now deal with temporary disconnects and network unavailabilities, making checkpoint coordination difficult to realize.

Since we are targeting both local-area and wide-area clusters, our proposed protocol is based exclusively on independent checkpointing. In particular, we implement independent checkpointing in the context of a software shared-memory system to provide a robust, yet fa-

miliar programming model. In the last decade, an impressive amount of research has been conducted in software shared memory [17, 2, 6, 25, 26, 37, 35, 34, 19], mostly aiming for performance (e.g., relaxed consistency models, lazy protocols and communication hardware support). More recently, projects like InterWeave [36] propose a shared-memory programming model to support applications that run on wide-area clusters of heterogeneous machines. The specific question that we address is how to efficiently combine independent checkpointing and a software DSM protocol in order to provide a high-performance fault-tolerant shared-memory programming environment on both local-area and wide-area clusters. Specifically, we show how a Home-based Lazy Release Consistency (HLRC) DSM protocol [18] can be extended with independent checkpointing in order to efficiently tolerate single-fault failures.

While independent checkpointing avoids global coordination, it is susceptible to rollback propagation (i.e., the domino effect [32]), where a single failure may recursively force peer processes to rollback through multiple checkpoints, possibly to the beginning of the computation [10]. A solution to the rollback propagation problem is message logging: when a node crash is detected, the process that was running on that node is restarted from its last checkpoint and recovers its lost state using protocol data structures and supporting logs of protocol data resident on other nodes. The challenge in building such a system lies in controlling the size of the logs and checkpoints in stable storage, without global coordination. Processes cannot discard their logs and old checkpoints when checkpointing because checkpoints are not coordinated: failed peer processes may require state or log entries saved prior to the last local checkpoint.

In this paper we present two algorithms, Lazy Log Trimming (LLT) and Checkpoint Garbage Collection (CGC), that can be used to build a fault-tolerant Home-based LRC DSM system. Home-based LRC protocols [18, 47] are easier to implement relative to many other DSM protocols and have low communication and memory overhead, making them good candidates for large-scale local-area clusters as well as for meta-clustering. We address two specific questions: *(i)* what are the rules for LLT and CGC, and *(ii)* how effective are these algorithms for real applications.

To answer the first question, we define the rules for trimming each component of the log and prove that they are correct, that is, the state can be recovered using the trimmed logs. We also define rules for garbage collecting old checkpoints and show that they are correct. These rules are the main contribution of the paper: they describe the information necessary to recover the state of the computation in the DSM protocol. We develop our algorithms in two steps. We first assume that nodes have knowledge of a global vector time, which simplifies the formal statement of the rules. We then show a practical solution to get a good approximation of the global vector time, using a global vector time manager. (Note that while the global vector time manager allows LLT and CGC to work efficiently, its availability is not critical. If the node on which the manager runs crashes, it can be easily restarted elsewhere. If the manager is temporarily unavailable due to network unavailability, logs will grow in size on stable storage but the correctness of the on-going computation will *not* be affected.)

To answer the second question, we have implemented LLT and CGC in a Home-based LRC prototype that runs on a local-area cluster of PCs interconnected with a Myrinet LAN [3].

Using this system, we have studied three applications from the SPLASH-2 [45] benchmark suite. Our results show that our scheme effectively bounds the size of logs and the number of checkpoints that must be kept. For all three applications, we never have to retain more than four checkpoints (only three in all but one process of one application) and logs never grow beyond about one third of the applications' shared-memory footprints.

The remainder of the paper is structured as follows. Section 2 presents related work in rollback recovery in message passing systems and fault-tolerant DSM systems. An overview of our approach to building a fault-tolerant DSM system is given in Section 3. Section 4 describes the recovery supporting data structures and proves the correctness of our rules for log trimming and garbage collection. Section 5 describes a practical scheme for implementing LLT and CGC. Experimental results and an evaluation of LLT and CGC are presented in Section 6. Limitations of our scheme and possible improvements are discussed in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

In this section we first briefly present rollback recovery methods based on independent checkpointing in general message passing systems. We then summarize previous work on recoverable DSM and present several fault-tolerant DSM systems that use log-based recovery and/or independent checkpointing.

An exhaustive survey of general rollback recovery in distributed systems is given in [10]. In *uncoordinated checkpointing* [1, 42] processes take checkpoints independently and track message dependencies between them in order to determine a consistent global checkpoint by rolling back processes in response to a failure. After a failure and rollback, a recovering process collects and aggregates dependency information from all processes in the form of a *rollback-dependency graph* [1] or a *checkpoint graph* [42]. It then determines the recovery line and implicitly which processes need to also rollback. This approach suffers from the domino effect [32].

An optimal checkpoint garbage collection algorithm based on dependency tracking was devised in [43]. It was also proved that there exists a tight bound of $n(n+1)/2$ on the number of *useful* past checkpoints that need to be retained in a system consisting of n processes. Our fault-tolerant DSM also uses past checkpoints, but only partial information from these checkpoints is needed for recovery of some process. Although our system is not based on dependency tracking, a comparison is nevertheless interesting: preliminary evaluation of our CGC algorithm shows that the number of past checkpoints it uses is within the above theoretical bound.

Communication-induced checkpointing [4, 16, 44] eliminates the domino effect in systems based on independent checkpointing by piggybacking protocol information on all application messages and *forcing* processes to take additional checkpoints to advance the recovery line. Forced checkpoints are always taken on the critical path, i.e., before a message is delivered, therefore they incur high latency. The approach adds overhead, sacrifices autonomy of processes in deciding when to take checkpoints and may result in useless checkpoints being

taken. Recent work in the area has focused on eliminating useless checkpoints and reducing the number of forced checkpoints [16].

Log-based rollback recovery assumes the piece-wise deterministic (PWD) execution model [38]; that is, a process executes a sequence of deterministic state intervals delimited by nondeterministic events, which can be identified and logged during failure-free execution. Under this assumption, the state of a failed process can be exactly recovered by restarting it from the last checkpoint and replaying logged events, combined with the deterministic re-execution of its state intervals. In *sender-based message logging (SBML)* [24] messages received by a process are logged at the sender processes. Logging overhead is reduced by taking logging out of the critical path and by using volatile memory instead of stable storage. SBML can only support single failures and the only type of nondeterministic event it can handle is the receipt of a message. In [40], where logging was first used in a recoverable DSM system based on Lazy Release Consistency, it was noted that the changes in page state occur at deterministic points, therefore it suffices to log acquires and access misses during normal execution. Our logging schemes are also based on this property, but they exploit specifics of the HLRC protocol to reduce the amount of logged and checkpointed state.

A comprehensive survey of recoverable DSM systems with various consistency models and implementation techniques is given in [29]. Coordinated checkpointing is used in systems like [7, 20, 27, 5, 8] either by forcing a synchronization of all processes when taking a checkpoint, or by leveraging global synchronization existent in the application or in the operation of the underlying DSM system. Communication-induced checkpointing is used in [46, 41, 39, 23]. Systems using independent checkpointing and logging in volatile or stable storage are described in [33, 40, 30, 8]. Uncoordinated checkpointing with dependency tracking applied to DSM is explored in [21, 22]. A transactional recoverable DSM is designed in [13] by taking advantage of techniques from database systems.

To date, much of the work on recoverable DSM systems using *log-based recovery* has focused on developing new logging schemes to reduce the high failure-free logging overhead (caused by the typically high communication frequency of DSM-based applications). In [8], Costa et al. have integrated log-based fault tolerance support into TreadMarks [26], a DSM system that also implements Release Consistency. Their work is different from ours in that their system leverages the global garbage collection (GC) phases of TreadMarks to take coordinated checkpoints. While they also use intermediate independent checkpoints and volatile logs to speed up recovery from single-node failures, the recovery is not entirely based on independent checkpoints, as it may need to use pages from the last global consistent checkpoint. Another difference is that all logs and past checkpoints are discarded at a global checkpoint, so their system does not face the problem of dynamic log trimming and checkpoint garbage collection.

The idea of volatile logs of *accesses* to shared memory was proposed in [33] for a sequentially-consistent DSM. Logs were flushed to stable storage on every page transfer which, combined with the potentially large size of the logs, made the scheme very expensive [40].

In [40], log-based recovery was first used in an LRC DSM. Their system uses independent checkpointing along with pessimistic logging by a receiver at synchronization points and access misses. To reduce the overhead of access to stable storage for each logged message, log

entries are stored in volatile memory and flushed to stable storage before sending a message to another process. Log flushing takes place on the critical path and can be expensive if synchronization is frequent. Every process must log all the data it needs for recovery, which leads to unnecessary replication of state and wastes stable storage. Because the log is saved in stable storage and is only used for the recovery of the process that creates it, the problem of resource consumption is easily addressed by taking a checkpoint and discarding the log when its size exceeds a limit. In our system, we also keep logs in volatile memory but only require that they are saved to stable storage at least with every checkpoint taken. We log only minimal state and do not replicate it across nodes. Our mechanisms for log trimming are totally decoupled from any policy that decides when trimming is to be performed, or when a checkpoint must be taken. In our system, LLT and CGC operations do not require a process to take a checkpoint.

In [30], a sender-based logging scheme is used for fault tolerance in DiSOM, an entry-consistent system. A new copy of an object is logged to volatile memory after every update (at release-write). Log trimming is made simple by the simple consistency model, which does not allow multiple writers: processes trim their logs whenever a new checkpoint is created in the system using logical times of logged objects. Our system allows multiple writers and uses a different consistency model, which makes the problem of garbage collection harder.

Other work on recoverable DSM based on uncoordinated checkpointing improves on the traditional data dependency tracking to detect processes that need to rollback in response to a failure. In [21, 22], a general model of dependencies in a typical DSM is developed, and it is shown that message dependencies that cause rollback propagation can be reduced. However, the model is applied to a variant of LRC modified to eliminate some dependencies, and rollback propagation is still a problem, especially if synchronization is frequent.

3 Overview

We propose a scalable fault tolerance algorithm for home-based software distributed shared memory, based exclusively on independent checkpointing. In this section we present the fault-tolerant system model, the base DSM protocol, and the proposed recovery scheme based on independent checkpointing.

3.1 The Fault-Tolerant System Model

We consider distributed applications running on clusters of interconnected computers, where each process of an executing application runs on a distinct node in the cluster. Processes communicate by message passing using reliable communication channels. Process execution is piece-wise deterministic in the interval between the receipt of consecutive messages. Failures are fail-stop. A single process can fail at a given moment in time (single-fault failure) and a process is considered failed with respect to the state of the computation until it has completely restored the state it had before the crash. We assume that there exists a mechanism for failure detection.

The support for stable storage can be either local to a node or remote, i.e., at another node in the cluster. If, for example, a process uses a local disk for saving its checkpoint, then it is assumed that the disk (and therefore the restart checkpoint) will be available for restart and recovery after a failure that caused the process to die. If this is not the case, the process can save its checkpoints on another node responsible for providing its stable storage support, even in case of failure; when needed, the restart checkpoint can be simply retrieved from the supporting node. In this paper we do not address the problem of storage support, neither do we try to quantify the (possibly significant) impact of storing checkpoints across the network. For all our purposes, we assume that stable storage is local to a node, and that it remains available after the failure of the node.

3.2 The Software DSM Protocol

Software DSM uses the virtual memory mechanism and message passing to provide a shared-memory programming model on clusters. The synchronization operations, lock *acquire/release* and *global barriers*, are implemented using message passing. The memory consistency model dictates the order in which writes to shared pages on different nodes must be completed. In *release consistent* [14] software DSM, writes are completed at synchronization time. Coherency data is in general propagated as page invalidations. The contents of a page is updated lazily at page fault time, either from its home [47], or from the last writer(s) [25]. If the protocol supports multiple writers, then updates are detected as a difference (*diff*) between the modified page and a reference copy created before the first write following a page fault [26].

Lazy Release Consistency (LRC) [25] is a variant of release consistency in which propagation of writes is delayed to the time of an acquire. Writes of a process are grouped into *intervals* delimited by synchronization operations. Page invalidations are propagated in the form of *write notices (wn)*, where a write notice is a pair (p, t) that specifies that some page p was updated during some interval t. The local logical time of a process is defined as a counter of local intervals. The partial-order relation *happened-before* [28] (denoted $\prec$) between intervals across the cluster is captured as a vector of logical times called *vector timestamp* [25]. The vector timestamp T_i keeps track of intervals for which write notices were received by process P_i.

Our target DSM protocol is a Home-based Lazy Release Consistency protocol (HLRC) [18]. We chose HLRC because it is simple, scalable, has low communication and memory overhead [47], and has nice properties for fault tolerance. In HLRC (see Figure 1), every shared page p has an assigned home $H(p)$ which maintains the most recent version of the page. Suppose that a process P_j has acquired a lock L before writing to page p (the write is labeled $W(p)$ in Figure 1). After releasing L, P_j receives a request for L from P_i, which is acquiring the lock next. The lock granting message sent to P_i will include T_j and a write notice $wn(p)$ for p. Upon receiving L, P_i invalidates p and updates its vector timestamp as $T_i = T_i^{new} = max(T_i, T_j)$.

At release time, the writer P_j sends its diffs to $H(p)$, where they are applied to p. The home $H(p)$ stamps p with a version vector $p.v$ that records the most recent intervals whose writes

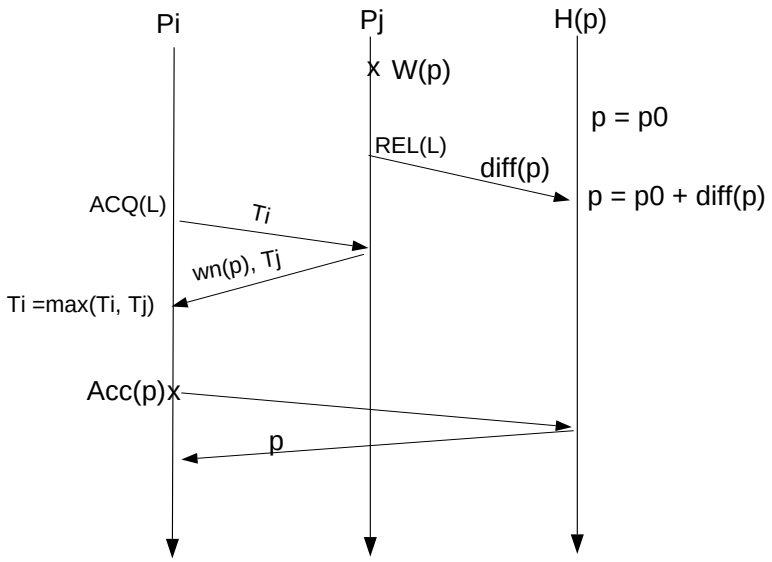

Figure 1: *The HLRC DSM protocol: process P_j writes to a page p homed by H(p) and which is later accessed by process P_i.*

were applied to p, i.e., for which it received the corresponding diffs.

Following the invalidation of p at the acquire, the non-home process P_i records the version $p.v^N$ it *needs* in case of an access, according to $wn(p)$. The first access $Acc(p)$ of P_i to p after the invalidation will miss. In the page fault handler, P_i will send a request to $H(p)$ for $p.v^N$, the minimal version of p it expects.

3.3 Recovery

Our approach in building a fault-tolerant home-based DSM is as follows:

- processes take independent checkpoints (decisions of when and what to checkpoint are purely local decisions),
- each process maintains logs of protocol data sent to its peers in volatile memory, and
- a failed process will restart from its latest checkpoint and use logs from peer processes to deterministically replay its execution.

Two important things to note: *(i)* between checkpoints, it is sufficient to maintain logs in volatile memory because we assume single-fault failures, and *(ii)* each process must save its log to stable storage at least on every checkpoint because if it crashes, after recovery, it may need its log to support recovery from subsequent crashes of other nodes.

To recover the state of a process in HLRC, we *checkpoint* shared pages only at home nodes and *log* those communication events that induce changes in the DSM state. A process P_i recovers from failure by log-based re-execution, starting from its last checkpoint. Because of

the relaxed memory model, intermediate states of P_i during recovery, as well as its recovered state, do not need to be the same as during normal execution. The execution replay needs only enforce that a *read access* to a page returns the same value, and not that all writes unrelated to that access are applied to the page.

The synchronous events that induce changes in the DSM protocol state d_i at process P_i and must be replayed during recovery are *synchronization operations* and *shared memory accesses*. When recovering, a process P_i performs the replay of write notices received at synchronization points using logs kept by its peer processes. To replay shared memory accesses, we exploit the HLRC protocol to avoid the expensive logging of page transfers: a page p is checkpointed only at its home node $H(p)$, and diffs for p are logged by its writers. We also use the protocol in that, since a request for a page p does not change the protocol state at $H(p)$, page requests do not need to be logged. For replay, a miss on p is serviced with a local copy of the page dynamically reconstructed from a checkpointed copy provided by $H(p)$, to which an ordered sequence of logged diffs has been applied. The home retains successive checkpointed copies of p from a sequence of past checkpoints. Because the checkpoints of P_i and $H(p)$ are not coordinated, the starting copy for P_i's replay of accesses to p may need to come from one of the older checkpoints of $H(p)$.

As previously discussed, the challenge in building a recoverable DSM system based on independent checkpointing and logging is how to limit the size of the logs and the number of checkpoints that must be kept on stable storage: old checkpoints cannot be discarded and logs can grow indefinitely because checkpointing is not globally coordinated. Intuitively, however, as the execution makes progress and all processes in the computation take checkpoints, old checkpoints and log entries will eventually be guaranteed to not be needed by any process for recovery. In order to limit the amount of data that must be kept on stable storage, we need to dynamically and safely identify the set of checkpoints and log entries that may be needed for rollback recovery at any time during the execution; all others are superfluous and can be discarded.

Our approach is to perform *lazy log trimming* (LLT) and *checkpoint garbage collection* (CGC). In the following section, we describe the state required for execution replay. We show how logs and checkpoints are used to recover from a failure and define the rules for LLT and CGC, assuming the instant availability of a global vector time.

4 Data Structures for Recovery

In this section we describe the log data structures for recovery support (*write-notice, synchronization* and *diff* logs) and show how they can be used along with checkpoints of homed pages for recovery from single-node failures. We also define the rules for log trimming and checkpoint garbage collection that allow correct recovery of any process.

4.1 Definitions and Assumptions

Let the state of a DSM system consisting of n processes be $S = \bigcup_{i=1}^{n}(c_i \cup l_i \cup d_i)$ where, for some process P_i, c_i is the recovery data on stable storage (checkpoints and saved logs), l_i are all logs in volatile storage, and d_i is the state of the DSM protocol at P_i. For example, d_i includes the vector timestamp, shared pages, page table etc., while c_i includes processor state, saved logs, the vector timestamp and other data structures in d_i essential to recovery.

For simplicity, we assume that c_i and l_i are trimmed, checkpoints are garbage collected, and volatile logs are saved to c_i only when a checkpoint is taken. Let T_i^c be the vector timestamp T_i at the moment of the most recent checkpoint of P_i (also called the *restart* checkpoint). The vector T_i^c *is* part of the checkpointed state.

We define the $\leq$ relation on a pair of vectors to yield true iff the component-wise $\leq$ relationship holds for all elements. Note that $\leq$ is a total order, so the *min* operation on a set of vectors is well defined.

Our log trimming and CGC rules are based on *timestamping* a checkpoint of P_i with a vector T_{ckp}^i. We assume that, ideally, at any instant any P_i has knowledge of the *global vector time* T_g, where $T_g[k]$ is the local logical time of process P_k. A P_i uses T_g to timestamp its checkpoints so that $T_{ckp}^i = T_g$ at the moment of the checkpoint. (In Section 5 we will show how this assumption can be relaxed by using an approximation of the global vector time T_g for timestamping checkpoints, and provide a practical scheme for approximating T_g.)

An important observation on the bounds we prove for log trimming and checkpoint GC is that they tolerate checkpoints to be timestamped with an approximation of the global vector time (as long as the approximation is less than or equal to the real global vector time), and stale information to be used for computing them. The only negative effect this may have is that c_i and l_i may not be minimal with respect to a global snapshot of the system state.

4.2 Write Notice Replay

Every process P_i logs write notices it generates in a volatile log, *wn_log*, for use in a log-based replay of synchronization operations by any recovering process P_j. A write notice log entry records the list of pages that were updated in a certain interval.

After a crash, the *wn_log* will be regenerated by deterministic re-execution from the last checkpoint on. At checkpoint time, P_i trims the whole log and saves its volatile part in c_i, if non-empty.

Lemma 1 (Wn log trimming) *A process P_i can support the wn replay by including in c_i only entries of wn_log corresponding to intervals starting from ckp_int $= \min_{j \neq i} T_j^c[i] + 1$, where T_j^c is the vector timestamp of P_j at the time of its last checkpoint.*

Proof. During re-execution, a process P_j will need *only* write notices generated by P_i that it had received after taking its last checkpoint. At the time that checkpoint was taken, the i^{th} element of P_j's vector timestamp ($T_j^c[i]$) recorded the last interval for which write notices

had been received from P_i. Therefore, to support execution replay of P_j, P_i must retain only write notices in intervals larger than $T_j^c[i]$. The minimum of this value over all P_j is the oldest interval from P_i received by any other process after its restart checkpoint. Hence, to cover recovery of any process, P_i must retain write notices from later intervals, i.e., those with a logical time at least equal to *ckp_int*.

■

4.3 Synchronization Replay

Every process P_j logs the lock acquire requests it services for any process P_i in $acq\_sent\_log[i]$, and the replies it receives from P_i to its lock acquire requests in $acq\_rcvd\_log[i]$. This means that each reply to an acquire message is logged at two nodes.

4.3.1 Supporting Data Structures

During normal execution of an acquire (ACQ), P_i sends its vector timestamp T_i to the process P_j, which owns the lock, and receives a set of write notices with the lock. During re-execution, P_i replays the ACQ using write notices from the wn_log's of other processes, and advances its vector timestamp T_i exactly as before crash. The previous lock owner P_j supports the replay of P_i by computing and logging in a per-process log $acq\_sent\_log[i]$ the *new* value of T_i, $T_i^{new} = \max(T_i, T_j)$. T_i^{new} is the new value of T_i after the ACQ has completed.

The $acq\_sent\_log[j]$ at a process P_i (as a lock owner) is a volatile data structure, subject to loss in a crash. Because it is created as a result of asynchronous events (lock acquire requests received from P_j), it cannot be regenerated during re-execution. In order to be able to recover $acq\_sent\_log[j]$, we simply mirror it at the acquirer P_j: after completing the acquire, P_j will log its new vector timestamp T_j^{new} in a per-process log $acq\_rcvd\_log[i]$.

Note that, for any pair of processes, the *acq_sent* and *acq_rcvd* logs are perfectly identical. As a result, a process does not need to save them in stable storage, since in case of failure they can be entirely restored from peer processes.

The support for barrier synchronization replay is similar.

4.3.2 Acquire Log Trimming

The following Lemma provides bounds on intervals corresponding to the oldest entries that a process must retain from its acquire logs in order to be able to support the recovery of another process.

Lemma 2 (ACQ log trimming) *A process P_i can support the ACQ replay of P_j by retaining only entries of $acq\_sent\_log[j]$ with $T_j[j] > T_j^c[j]$, and it can restore the strictly needed portion of the $acq\_sent\_log[i]$ of P_j by retaining only entries of $acq\_rcvd\_log[j]$ with $T_i[i] > T_i^c[i]$.*

Proof. For its *ACQ* replay, a recovering P_j will only need from some P_i entries of $acq\_sent\_log[j]$ logged for acquires that P_j has executed after its last checkpoint. Therefore P_i can safely trim $acq\_sent\_log[j]$ by discarding entries with $T_j[j] \leq T_j^c[j]$.

In order to recover its $acq\_sent\_log[i]$, a recovering P_j needs from the acquirer P_i entries of P_i's $acq\_rcvd\_log[j]$. Note that P_j needs to recover only the portion of $acq\_sent\_log[i]$ that would be strictly needed for a *potential ACQ* replay of P_i, in case a crash of P_i will occur sometime in the future. This portion of P_j's log consists of entries with $T_i[i] > T_i^c[i]$. To back it up, P_i must retain from $acq\_rcvd\_log[j]$ only entries for which $T_i[i] > T_i^c[i]$.

■

Note that the values of the log bounding intervals proved in Lemmas 1 and 2 do not depend on the checkpoint timestamps T_{ckp}^j and therefore they are always optimal, regardless of whether in practice an implementation uses some approximation of the global vector time instead of the real T_g for T_{ckp}^j.

4.4 Replay of Shared Memory Accesses

Every process checkpoints pages for which it is the home. Every writer (including the home process itself) logs the diffs it produces for all pages it writes to. Accesses to a page are replayed by locally and dynamically applying logged diffs to needed pages.

In HLRC, the home $H(p)$ of page p stamps it with a version $p.v$ as a vector of logical times corresponding to intervals in which p was written; $p.v[i]$ advances with application of a diff from P_i. A process P_i where p is invalid *needs* a version $p.v^N$, according to *wn*'s received so far. This is the minimal version of page p that P_i will request from $H(p)$ on its next access and miss to p. During normal operation, $H(p)$ may reply with the requested version or with a higher version that incorporates additional writes. During replay, by taking advantage of the HLRC protocol, it is sufficient for the access to p to be serviced with a minimal version of p, i.e., which contains only writes that "happened-before" the faulting access. As a result, changes in contents of a page after a miss and fetch from its home need not be reproduced exactly during replay, as it would be the case if pure message logging of page transfers were used.

4.4.1 Supporting Data Structures

A page p is only checkpointed by its home $H(p)$. A home $H(p)$ retains a sequence p_{ckp} of copies of p from past checkpoints. Any writer P_j logs every diff it sends to $H(p)$ in a per-page log $diff\_log(p)$. The logged *diff* entry is stamped with its vector timestamp T_j, referred to as $diff.T$.

During re-execution, P_i replays only the minimal changes of p after a miss by emulating the operation of $H(p)$. It maintains an evolving copy of p built from a *starting copy* p_0 obtained from $H(p)$, to which it applies partially ordered diffs obtained from all writers' logs $diff\_log(p)$.

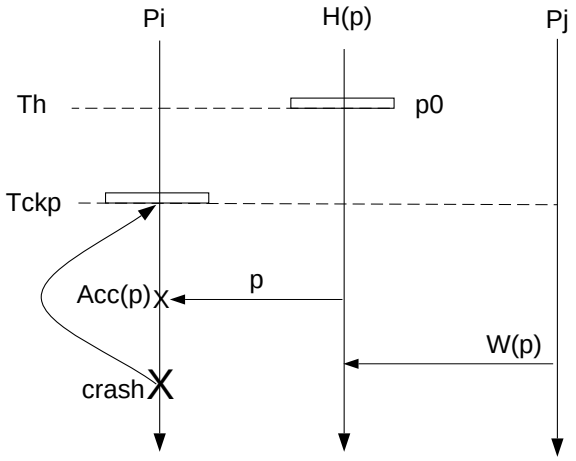

Figure 2: *Bounds for a checkpointed version p_0 of page p which is safe for replaying an access of P_i after crash and restart.*

The following Lemma provides conditions for a *maximal* starting copy p_0 (i.e., to which nothing can be added without risking to compromise correctness of the recovery). It also gives conditions on the diffs needed for a correct replay of accesses to p starting from the maximal p_0.

Lemma 3 (Maximal p_0) *1. If page p is needed for the replay of P_i starting from a checkpoint timestamped with T^i_{ckp}, then $p_0 \in p_{ckp}$ can come from the last checkpoint of H(p) timestamped T^H_{ckp} for which $T^H_{ckp} \leq T^i_{ckp}$.*

2. To ensure the correct replay of p starting from the above p_0, any writer P_j must supply diffs with timestamps $diff.T[j] > T^H_{ckp}[j]$.

Proof.

We look at the first access Acc(p) of P_i to page p after a crash and restart from its latest checkpoint timestamped T^i_{ckp}, as shown in Figure 2. (A diff sent by a writer of p to home $H(p)$ is represented by an arrow pointing towards $H(p)$; a fetch of p from $H(p)$ is shown as an arrow pointing away from $H(p)$.)

Recall the assumption that the checkpoint timestamp of a process is the global vector time T_g when a checkpoint is taken. Let T^H_{ckp} be a checkpoint timestamp of $H(p)$.

1. A starting page p_0 which is $safe$ for replay can be any copy of p with a version $p_0.v \leq T^i_{ckp}$. Indeed, regardless of the memory consistency model, a safe p_0 for P_i's access is always one that does not contain *any* writes to p that occurred in the future (in real time) with respect to the restart checkpoint of P_i. In Figure 2, $W(p)$ is such a write. In LRC terms, a future conflicting write $W(p)$ from a process P_j could only occur in some interval $T_j[j] > T^i_{ckp}[j]$, therefore a page with $p_0.v \leq T^i_{ckp}$ is always safe for replay. In particular, since $p_0.v \leq T^H_{ckp}$, p_0 can come from the last checkpoint of $H(p)$ with a timestamp $T^H_{ckp} \leq T^i_{ckp}$. (Note that in LRC the condition $p_0.v \leq T^i_{ckp}$ is not strictly necessary. A safe p_0 can be any copy of p

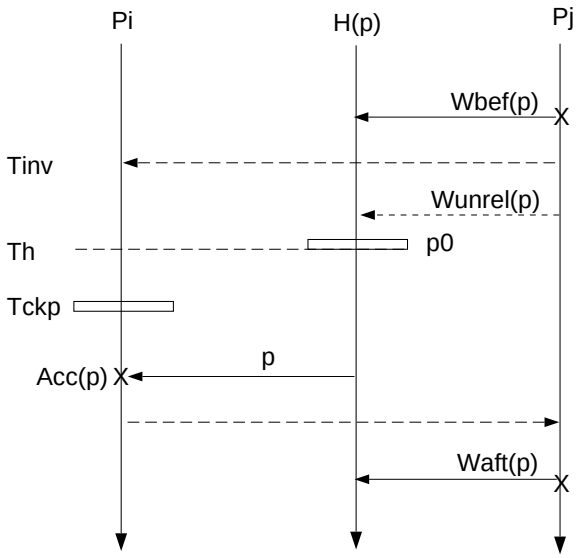

Figure 3: *Case A.1: Page p invalid in P_i's checkpoint and last invalidation of p at P_i is at most as recent as the last safe checkpoint of home H(p).*

containing writes $W(p)$ that occurred after T^i_{ckp}, but which are not related to Acc(p) under the $\prec$ order.)

2. Next, we show that a p_0 from the *last* checkpoint of $H(p)$ with timestamp $T^H_{ckp} \leq T^i_{ckp}$, proved above as *safe* for replay, is also *sufficient* for a correct replay of Acc(p), under the conditions of Part 2 of the Lemma.

Let T_{inv} be the global vector time when P_i received the *last* invalidation for p (for example, through an acquire) before the restart checkpoint. We prove p_0 is sufficient for two cases, depending on whether, during normal operation, p was valid or not at the time of Acc(p).

Case A. Suppose p was invalid at Acc(p) during normal operation, so it is also invalid after restart of P_i. Then P_i's page table entry in c_i records $p.v^N$, the *minimal* version P_i will expect on Acc(p) during recovery.

We consider two instances, depending on when the last invalidation of p at moment T_{inv} occurred in the $\leq$ order: before, or after the moment $H(p)$ took its last safe checkpoint of p_0, i.e., having $T^H_{ckp} \leq T^i_{ckp}$.

1. The invalidation of p occurs (in the total order defined by $\leq$) before or at the moment when $H(p)$ takes its checkpoint, i.e., $T_{inv} \leq T^H_{ckp}$ (Figure 3).

   The figure shows a simplified scenario in which P_i first synchronizes with P_j, resulting in the invalidation of p at T_{inv}. The dashed arrows between P_j and P_i represent the transfer of write notices at synchronization operations that protect conflicting accesses to p.

   For p_0 to be sufficient during replay, it must contain the last write $W_{bef}(p)$ of P_j that happened before Acc(p): $W_{bef}(p) \prec Acc(p)$. Note that $W_{bef}(p)$ occurs at P_j before T_{inv}, due to LRC constraints. Since it can also be enforced at $H(p)$ before the synchronization

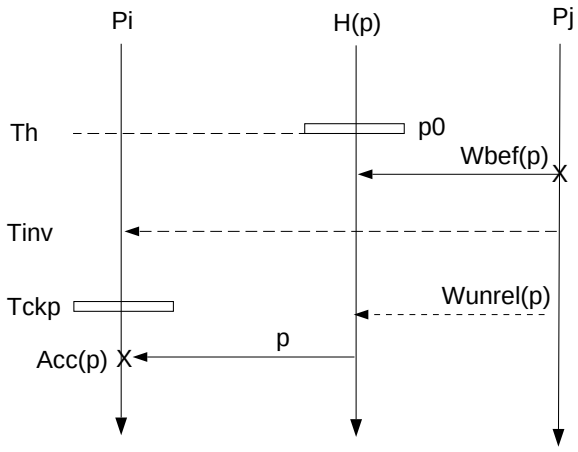

Figure 4: *Case A.2: Page p invalid in P_i's checkpoint and last invalidation of p at P_i is more recent than the last safe checkpoint of home H(p).*

takes place, the checkpointed p_0 also incorporates it. Therefore, in this case p_0 alone is sufficient for replay, which establishes the proof for Case A.1.

Note that other writes not related to Acc(p) may be included in p_0 after $W_{bef}(p)$. Suppose for example that the $first$ access conflicting with Acc(p) that occurs after it in the $\prec$ order is a write $W_{aft}(p)$, also performed by P_j ($Acc(p) \prec W_{aft}(p)$ in Figure 3). Then writes of P_j (labeled $W_{unrel}(p)$ in Figure 3) between T_{inv} and the moment of W_{aft} are not related to Acc(p). They can be incorporated in the checkpointed copy p_0 of $H(p)$ but will not affect the correct replay of Acc(p).

2. The invalidation of p occurs (in the total order defined by $\leq$) after $H(p)$ takes its checkpoint, i.e., $T^H_{ckp} < T_{inv}$ (Figure 4).

   Then there might exist conflicting writes W_{bef} previous to Acc(p) ($W_{bef} \prec Acc(p)$) that occurred prior to invalidation but after $H(p)$ took its checkpoint. Such writes would not have been captured in the checkpointed p_0, which makes p_0 alone unusable for replay (in our example, we would have $p_0.v[j] < p.v^N[j]$). To obtain the version $p.v^N$ needed for the replay of Acc(p), the recovering P_i must apply to p_0 writes like W_{bef}, logged by P_j as diffs. These writes would occur at logical times $T_j[j] > T^H_{ckp}[j]$, so P_j must supply all diffs with a timestamp $diff.T[j] > T^H_{ckp}[j]$.

Case B. Suppose p was valid at the moment of Acc(p) during normal operation, so it was also valid when P_i took its checkpoint. Then there must have been an access of P_i to p after T_{inv} but before its last checkpoint at T^i_{ckp}. Suppose that such an access was a read $R(p)$, and there was a write $W_{own}(p)$ such that $R(p) \prec W_{own} \prec T^i_{ckp} \prec Acc(p)$ (Figure 5).

With respect to some conflicting write W_{bef} of P_j for which $W_{bef} \prec W_{own}$, the replay of Acc(p) is correct, under the conditions proved above in Case A.

We must only ensure that the replay of Acc(p) is correct with respect to W_{own}. The starting p_0 alone cannot be used for replay since it may not contain W_{own}. For correct replay P_i must

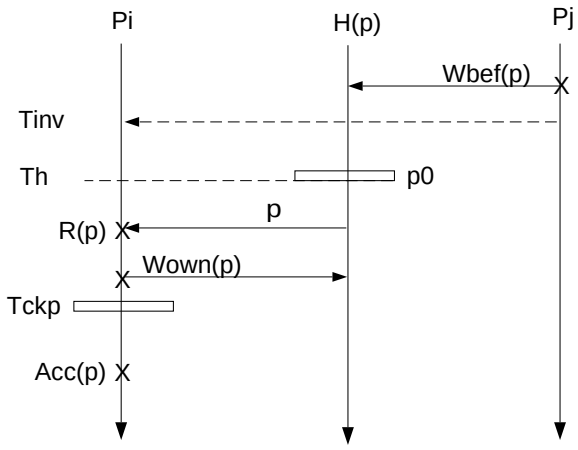

Figure 5: *Case B: Page p valid at the time of P_i's checkpoint. Local write $W_{own}(p)$ not captured by p_0 must be replayed.*

save diffs for its writes that it created and logged after $H(p)$ took its checkpoint, i.e., diffs with $diff.T[i] > T^H_{ckp}[i]$.

■

Note that Lemma 3 links the starting page p_0 with its corresponding diffs through the timestamp T^H_{ckp} of the checkpoint from which p_0 comes: if p_0 would be selected from an earlier checkpoint, then writers would have to keep more diffs for the correct replay of accesses to p.

4.4.2 Lazy Log Trimming (LLT) and Checkpoint Garbage Collection (CGC)

Lemma 3 proved conditions for a $H(p)$ to retain a checkpointed copy p_0 of a page p needed for recovery of one process in the system. Note that the selection of the checkpoint is independent of the page itself (it involves only checkpoint timestamps). Also note that, for efficiency reasons, a home node H will have to keep a *window* of past checkpoints of all pages it manages, starting with the checkpoint determined by Lemma 3.

The following Lemma simply generalizes the conditions for the garbage collection of checkpoints not needed for recovery of any process. It also shows that after a $H(p)$ performs a CGC operation a writer can, at a later moment, "lazily" trim its diff logs using information from $H(p)$, namely the timestamp of the oldest checkpoint retained.

Lemma 4 (CGC and LLT) *1. A home process H can support the page replay of any process if it retains pages from a "first" checkpoint with timestamp $T^H_f \leq T_{min} = \min_{j \neq H} T^j_{ckp}$.*

2. A writer of page p, P_i, can support recovery replay for p by retaining only $diff\_log$ entries with $diff.T[i] > T^{H(p)}_f[i]$.

Proof.

1. From Lemma 3, Part 1. To cover recovery of *any* process, H must keep a *window* of past checkpoints of its homed pages, starting with a checkpoint timestamped T_f^H at least as old (in the $\leq$ order) as the oldest restart checkpoint in the system. That is, $T_f^H \leq \min_{j \neq H} T_{ckp}^j$.

2. The checkpoint stamped T_f^H determined above also forces writers of a page p homed by H to keep diffs for a possible replay of accesses to p. According to Lemma 3, Part 2, applied to the timestamp $T_f^{H(p)}$ of this first checkpoint, a writer P_i must retain diffs with timestamps $diff.T[i] > T_f^{H(p)}[i]$.

■

Note that the bounds for retaining checkpoints and diff logs (T_{min} and T_f in Lemma 4) depend on the checkpoint timestamps T_{ckp}^j. This means that, in practice, the efficiency of CGC and LLT may be affected by how timestamps are assigned to checkpoints.

4.5 An Example

Figure 6 shows an example of how checkpoints are garbage collected and diff logs are trimmed using Lemma 4. Vertical brackets mark the *window* of checkpoints from which page copies are to be retained by each home. Note that the assumption of a global vector time available for timestamping checkpoints is also reflected in the figure, as checkpoints are ordered about an imaginary global real time axis running vertically.

In this example, process P_3 is about to take checkpoint c_{34} at global time T_g. The solid and dotted arrows show the dependency that sets the upper limit of P_3's checkpoint window at the moments when c_{34} and c_{33}, respectively, are taken. The dotted bracket represents the checkpoint window of P_3 at the time it took its previous checkpoint c_{33}. When taking c_{34}, the upper limit of P_3's window will become c_{32}, since Lemma 4 enforces a dependency from c_{12} (which determines the bound T_{min}) to c_{32}. The previous limit had been c_{31}, as set by the now obsolete dependency from c_{22} at the time c_{33} was taken. In effect, the window advances by including the new c_{34} and dropping the unneeded c_{31}. As a diff producer, P_3 trims its diff logs by retaining only diffs needed for pages homed by P_1 and P_2 if a replay would use pages from the first checkpoints in their windows (c_{11} and c_{22}, respectively). Earlier entries can be discarded. As noted before, trimming of diff logs can be performed lazily, when P_3 learns the timestamps of c_{11} and c_{22}.

An important observation on Lemma 4 is that an approximation of the checkpoint timestamps $T_{ckp}^j \leq T_g$ can be tolerated in the bound T_{min} on T_f. In this case, the approximate $\hat{T}_f \leq \hat{T}_{min} \leq T_{min}$ can only push p_0 to a checkpoint older than the ideal (optimal) T_f computed based on the global vector time T_g. If an older checkpoint with $\hat{T}_f < T_f$ is retained, Part 2 of Lemma 4 ensures that p can still be reconstructed by retaining necessary diffs for the older approximation of p_0. In the next section we present a practical solution based on approximations of global time and of T_{min} computation.

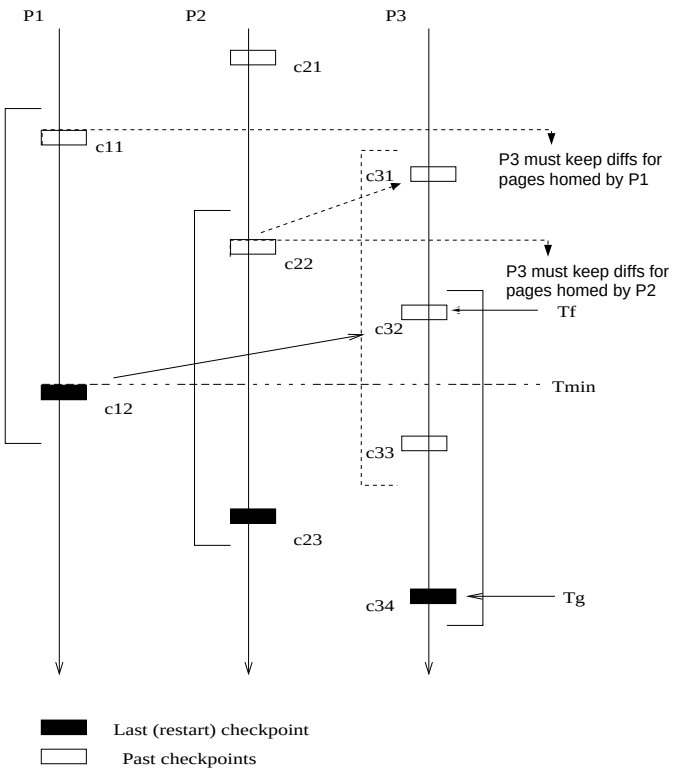

Figure 6: *Determining the checkpoint window for CGC and the diff log bounds for LLT by process P_3 at the time it takes c_{34}.*

5 Approximating Global Time

Lemmas 1, 2, and 4 prove ideal bounds for how each process can trim all unnecessary entries from its logs. A process P_i needs from all other processes P_j: *(i)* two elements ($T_j^c[i]$ and $T_j^c[j]$) of the vector timestamp at the moment of last checkpoint, T_j^c, for trimming the *wn* and *ACQ* logs, *(ii)* the checkpoint timestamp T_{ckp}^j of the restart checkpoint, for its CGC, and *(iii)* $T_f^j[i]$ (if P_i has logged diffs for pages homed by P_j), for its LLT. This information is distributed throughout the system. Moreover, so far we have assumed that any P_i has instant knowledge of the global vector time T_g used to timestamp its checkpoints.

We now relax these requirements. Specifically, we describe the effects of using an approximation instead of the ideal global vector time T_g and of using stale trimming information. We also describe a practical solution for obtaining a reasonable approximation of global time and "not too stale" trimming information in order to limit these effects.

First, consider a relaxation of the exact global time T_g to an approximate $\hat{T}_g \leq T_g$ which *preserves* the total order enforced by the $\leq$ relation on the set of checkpoints c_{ij} for all P_i. Assume instant knowledge of the *last* T_{ckp}^i at the home $H(p)$ of a page p. Because the order determined by $(T_g, \leq)$ is preserved in the new total order $(\hat{T}_g, \leq)$, the oldest restart checkpoint over all processes, stamped with $\hat{T}_{min}$, is in the same position with respect to the optimal checkpoint c_H^* that would ideally be selected by Lemma 4 if T_g were used. Hence $H(p)$ would still be able to determine the optimal c_H^* checkpoint for its CGC. However, a writer to p, P_j, may get an approximate bound for LLT of its diff log since now c_H^* has a timestamp $\hat{T}_f \leq T_f$, so it may be forced to keep additional log entries. Therefore, using the approximation $\hat{T}_g$ results in optimal checkpoint window size but may induce suboptimal diff logs.

Next, we examine the additional effect stale trimming information can have on the retained checkpoints and logs. One approach to distributing this information would be to propagate it lazily, for example piggybacked on protocol messages: a process P_i would only have to send to P_j a vector T_{ckp}^i and three integers $T_i^c[j]$, $T_i^c[i]$, and $T_f^i[j]$. These values may become obsolete, depending on the sharing/communication pattern. The logs of write notices and the synchronization logs will be trimmed less efficiently if $T_i^c[j]$ and $T_i^c[i]$ are stale (see Lemmas 1 and 2). For the page checkpoints and diff logs, note that even if the total checkpoint order is preserved by $\hat{T}_g$, the bound $\hat{T}_{min}$ may be less than T_{min} of Lemma 4 because of stale $\hat{T}_{ckp}^i \leq T_{ckp}^i$, and it may force a home $H(p)$ to push back its checkpoint window to a $c_H \prec c_H^*$. This may also increase the diff log size at some writer P_j by the feedback effect through $T_f[j]$.

Clearly, we are faced with two problems: the need for a good approximation of T_g and the need for accurate values of the bounds used for trimming logs. Our solution is to use a centralized manager M that stores three vectors per process P_i: T_c^i, T_{ckp}^i, and T_f^i. With this information, the manager can compute all the bounds needed for log trimming by any process. Processes propagate their local logical time to M so it can have a close approximation of the global time T_g. Updating logical time at M by a process P_i can be done as often as the logical time changes at that process, or in a lazy fashion if the rate of change is high

Application and problem size	*Size shared* (Mb)	*Exe time* (min)	*Ckp taken*	W_{max}	*Max diff log* (Mb)	*Ckp + log disk traffic* (Mb)	*Time disk write* (s)
Ocean 130x130	6.4	2	9-10	3	1.5	10.8	2.6
Water-Nsquared 17,576	11.8	40	10	3	3.0	27.5	8.5
Water-Spatial 32k	33.9	30	3-10	4	13.0	65.8	15.9

Table 1: *Evaluation of the independent checkpointing scheme with CGC and LLT in a HLRC DSM system. The last three columns refer to checkpoints of homed pages and diff logs on stable storage (disk).*

(note that this operation is *not* on the critical path).

A P_i sends to M its T_i^c and T_f^i, and obtains from M all the bounds needed to trim its logs. When taking a checkpoint, it obtains a vector timestamp $\hat{T}_g$ which is a close approximation of the actual T_g. After committing the new checkpoint, P_i confirms the new T_{ckp}^i to the manager. It is important to note that M also enforces the total order required by the Lemmas in Section 4 on the set of checkpoints, with respect to the approximation it provides for T_g.

While it may seem like the manager M is a centralized point of failure, it is not. Suppose M is collocated with one of the processes of the computation. In case of a crash, it is simply restarted and, once restarted, it just collects the vectors T_c^i, T_{ckp}^i and T_f^i from surviving processes. In the case where M is unavailable for a period of time because of network unavailability, processes will not be able to trim their logs and garbage collect their checkpoints during this period. However, this can only affect the amount of data kept on stable storage and it does not affect the correctness of the on-going computation.

6 Experimental Results

To evaluate our design, we have extended a Home-based LRC system [18] to include: *(i)* LLT, *(ii)* a checkpointing policy to decide when to checkpoint at each node, and *(iii)* a simulation of CGC. We have also implemented the manager that provides nodes of the cluster with approximations of the global vector time and the bounds needed by LLT and CGC. Our goal is to assess the overheads of LLT and CGC in terms of both volatile and stable storage requirements, or, equivalently, how effective LLT and CGC are at limiting the size of the diff logs and the number of checkpoints that must be retained for real applications.

Our policy for deciding when a node should checkpoint is to enforce a limit L on the size of the volatile log. The decision is independently made by each node: when the size of the log at a node exceeds L, that node takes a checkpoint. For each application, we set L to be 10% of the application's shared memory footprint. This low value forces frequent checkpoints to be taken, and thus presents a stressful test case for CGC. (For this reason, for example, a short-running application may unnecessarily take frequent checkpoints; this is solely for the purpose of evaluating effectiveness of our scheme and has no relevance in practice.)

Logs are trimmed and volatile logs are saved to stable storage only at checkpoint time. No optimizations (e.g., log compaction and incremental checkpointing) are used to reduce the

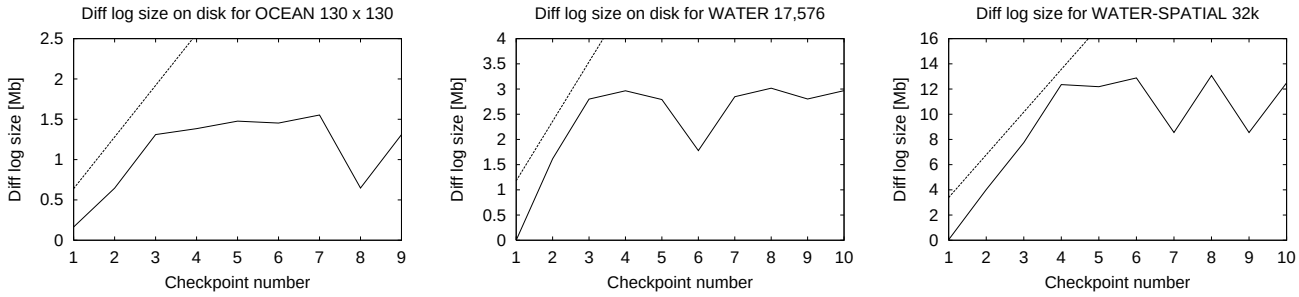

Figure 7: *Dynamics of log size in stable storage for three applications (logs are sampled at checkpoint time). Discrete points are connected to exhibit the trend under LLT control. Straight dotted lines show the unbounded log growth in the absence of LLT.*

space consumed by logs and checkpoints. Processes send updates of their logical time to the manager at every event that changes the local logical time. Updating logical time is not on the critical path at the sender, and is done at essentially zero cost at the receiver. The remainder of the trimming information is only sent to the manager when a process requests a global time approximation for checkpointing.

We have selected three applications from the SPLASH-2 benchmark suite [45], Ocean, Water-Nsquared, and Water-Spatial, to drive our prototype system. We choose these three applications in particular because they generate large volumes of diffs, and so present the worst-case scenario for LLT. **Ocean** is a fluid dynamics application that simulates large-scale ocean movements. **Water-Nsquared** simulates a system of water molecules in liquid state using a $O(n)$ brute force method with a cutoff radius. **Water-Spatial** solves the same problem in a 3-d setting. All our experiments run on a cluster of eight 300 MHz Pentium II PCs interconnected by a Myrinet LAN and using virtual memory-mapped communication [9].

Table 1 shows, for a run of each application, *(i)* the shared-memory footprint, *(ii)* total execution time, *(iii)* minimum and maximum number of checkpoints taken (on any node), *(iv)* the maximum size of the checkpoint window (W_{max}) over all nodes, *(v)* the maximum amount of stable storage consumed by diff logs over all nodes, *(vi)* the amount of checkpointed pages and diff logs written to stable storage at one (representative) node, and *(vii)* the estimated time that would be required to write this amount of data to local disk. Figure 7 plots the maximum (across the cluster) log size on stable storage against checkpoints. While we do not make definitive claims based on results for only three applications, we believe that these results are strong evidence for the following observations:

- *Independent checkpointing can be efficiently integrated with an HLRC DSM protocol to provide a robust shared-memory programming model.* Table 1 shows that the total expected time for saving checkpointed pages and volatile logs to disk is negligible compared to the execution time, even with our aggressive (small) limit on the size of the volatile log. Furthermore, the fact that the overhead of writing checkpoints and volatile logs to disk is small means that an aggressive limit on the size of volatile logs is practical. This means that the memory overhead of our scheme can be at most 10% of the shared-memory footprint.

- *LLT is effective at limiting log sizes.* Figure 7 shows that, for all three applications, the maximum log size across the cluster initially increases across the first several checkpoints but then flattens out. After the change in slope, the maximum log size is relatively constant although there are some variations from checkpoint to checkpoint. With no log trimming, the disk space consumed by logs would grow unbounded at a rate of L bytes per checkpoint, as shown by the straight line in each graph.

- *The size of the checkpoint window is small, implying that CGC is efficient at controlling the total size of checkpoint information kept on stable storage.* Across all applications, W_{max} was three pages on most of the nodes, with a value of four on a single node in Water-Spatial.

7 Limitations

Our fault tolerance scheme of Section 6 used a checkpointing policy based exclusively on a *local* optimization (aimed at limiting the size of the volatile log). This means that a node could arbitrarily delay taking a local checkpoint, which may increase the amount of state held at other nodes.

Intuitively, the problem lies in that a process which lags behind in taking checkpoints forces all other processes to accumulate large amounts of state (checkpoints and logs) in order to be able to support its recovery from a remote past checkpoint. In our case, this behavior is dictated by the T_{min} bound used in computing the timestamp T_f of the first checkpoint to be retained in its checkpoint window by each home node (Lemma 4). Since T_{min} is a minimum over timestamps of restart checkpoints, it can be held back indefinitely by a single lagging process. In turn, the upper limit of all checkpoint windows at all home nodes will be also held back, thus making CGC (and implicitly LLT) less effective.

Our policy from Section 6 may exhibit this behavior, as it enforces a checkpoint only when the local volatile logs overflow. An imbalance in the application running on the DSM system may cause one process to be less active in updating shared pages, so it will generate a lower volume of diff logs than other processes. As a result, during computation this process will take checkpoints at a lower rate than the others (in the worst case it may take none). This will force peer processes to keep more checkpoints and logged state for the recovery of the lazy process. Another example is an application-driven checkpointing policy that tries to optimize the size of checkpointed application state using memory exclusion [31]. If taking the checkpoint is indefinitely delayed by a process waiting for an opportunity of minimizing the checkpoint size, then this may cause an increase in the amount of recovery state at other nodes.

To overcome this problem, the solution is to force the lazy process to take a checkpoint. Determining such a process requires distributed information which is available to the manager M. Besides providing checkpoint timestamps and bounds needed for LLT and CGC, M could also: *(i) detect* the lagging process that stalls the CGC and LLT of other processes, and *(ii) induce* a forced checkpoint at that process.

We are currently in the process of implementing a checkpointing policy that integrates forced checkpointing through the manager.

8 Conclusions

In this paper we present the design of a fault-tolerant software DSM system based on independent checkpointing and logging. Independent checkpointing is particularly well suited to large LAN-based clusters, as well as meta-clusters over a WAN. In order to retain its advantages in such environments, the system must efficiently control the size of the logs and checkpoints without recourse to global coordination.

We describe the minimal state of the HLRC DSM protocol [18] that must be checkpointed and logged to support recovery from single-fault failures. We also: *(i)* define the rules for trimming each component of the log and show that these rules are correct, and *(ii)* define the rules for garbage collecting old checkpoints and show that these rules are correct. We have implemented our proposed algorithms in an HLRC DSM system and evaluated their performance using three update-intensive applications from the SPLASH-2 benchmark suite. Our results show that our scheme effectively bounds the size of the log and the number of checkpoints that must be kept. For all three applications, we never have to retain more than four checkpoints and logs never grow beyond about one third of the applications' shared-memory footprints.

We are currently in the process of exploring additional checkpointing policies, and completing the implementation of the fault-tolerant protocol.

References

[1] B. Bhargava, S. R. Lian. Independent Checkpointing and Concurrent Rollback for Recovery - An Optimistic Approach. *Proc. Symposium on Reliable Distributed Systems*, pp. 3-12, June 1988.

[2] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, v. 15, no. 1, pp. 29-36, February 1995.

[4] D. Briatico, A. Ciufoletti, L. Simoncini. A Distributed Domino Effect Free Recovery Algorithm. *Proc. IEEE International Symposium on Reliability, Distributed Software and Databases*, pp. 207-215, December 1984.

[5] G. Cabillic, G. Muller, I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. *Proc. 14th Symposium on Reliable Distributed Systems*, pp. 96-105, September 1995.

[6] J.B. Carter, J.K. Bennet, W. Zwaenepoel. Implementation and Performance of Munin. *Proc. 13th Symposium on Operating Systems Principles (SOSP)*, pp. 152-164, October 1991.

[7] J. B. Carter, A.L. Cox, S. Dwarkadas, E. N. Elnozahy, D. B. Johnson, P. Keleher, S. Rodrigues, W. Yu, W. Zwaenepoel. Network Multicomputing Using Recoverable Distributed Shared Memory. *Proc. IEEE International Conference CompCon '93*, February 1993.

[8] M. Costa, P. Guedes, M. Sequeira, N. Neves, M. Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 59-73, October 1996.

[9] C. Dubnicki, L. Iftode, E.W. Felten, K. Li. Software Support for Virtual Memory-Mapped Communication. *Proc. 10th International Parallel Processing Symposium*, April 1996.

[10] E. N. Elnozahy, L. Alvisi, D.B. Johnson, Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *Technical Report CMU-CS-99-148*, Carnegie Mellon University, June 1999.

[11] E. N. Elnozahy, D. B. Johnson. The Performance of Consistent Checkpointing. *Proc. 11th Symposium Reliable Distributed Systems*, pp. 86-95, October 1992.

[12] The Endeavour Project. *http://endeavour.cs.berkeley.edu*. University of California at Berkeley.

[13] M. J. Feeley, J. S. Chase, V. R. Narasayya, H. M. Levy. Integrating Coherency and Recoverability in Distributed Systems. *Proc. 1st Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proc. 17th International Symposium on Computer Architecture (ISCA)*, pp. 15-26, May 1990.

[15] A. Grimshaw, W. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, v. 40, no. 1, pp. 39-45, January 1997.

[16] J. M. Helary, A. Mostefaoui, R. H. Netzer, M. Raynal. Preventing Useless Checkpoints in Distributed Computations. *Proc. IEEE Symposium on Reliable Distributed Systems*, pp. 183-190, October 1997.

[17] L. Iftode, J. P. Singh. Shared Virtual Memory: Progress and Challenges. *Proceedings of the IEEE*, v. 83, no. 3, March 1999.

[18] L. Iftode. Home-Based Shared Virtual Memory. *Ph.D. Thesis*, Princeton University, June 1998.

[19] A. Itzkovitz, A. Schuster. MultiView and Millipage - Fine-Grain Sharing in Page-Based DSMs. *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 215-228, February 1999.

[20] G. Janakiraman, Y. Tamir. Coordinated Checkpointing Rollback Error Recovery for Distributed Shared Memory Multicomputers. *Proc. 13th Symposium on Reliable Distributed Systems*, October 1994.

[21] B. Janssens, W. K. Fuchs. Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory. *Proc. 13th Symposium on Reliable Distributed Systems*, pp. 34-41, October 1994.

[22] B. Janssens, W. K. Fuchs. Ensuring Correct Rollback Recovery in Distributed Shared Memory Systems. *Journal of Parallel and Distributed Computing*, October 1995.

[23] B. Janssens, W. K. Fuchs. Relaxing Consistency in Recoverable Distributed Shared Memory. *Proc. 23nd IEEE International Fault-Tolerant Computing Symposium (FTCS)*, pp. 155-163, July 1993.

[24] D. B. Johnson, W. Zwaenepoel. Sender-based Message Logging. *Proc. 17th International Fault-Tolerant Computing Symposium (FTCS)*, pp. 14-19, June 1987.

[25] P. Keleher, A. L. Cox, W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory, *Proc. 19th International Symposium on Computer Architecture (ISCA)*, pp. 13-21, May 1992.

[26] P. Keleher, S. Dwarkadas, A.L. Cox, W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *Proc. Winter 94 USENIX Conference*, pp. 115-132, January 1994.

[27] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, I. Puaut. A Recoverable Distributed Shared Memory Integrating Coherency and Recoverability. *Proc. 25th International Symposium on Fault-Tolerant Computing Systems (FTCS)*, June 1995.

[28] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, v. 21, no. 7, pp. 558-565, 1978.

[29] C. Morin, I. Puaut. A Survey of Recoverable Distributed Shared Virtual Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, v. 8, no. 9, September 1997.

[30] N. Neves, M. Castro, P. Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. *Proc. 13th Symposium on Principles of Distributed Computing (PODC)*, August 1994.

[31] J. S. Plank, Y. Chen, K. Li, M. Beck, G. Kingsley. Memory Exclusion: Optimizing the Performance of Checkpointing Systems. *Software - Practice and Experience*, v. 29, no. 2, pp. 125-142, 1999.

[32] B. Randell. System Structure for Software Fault-Tolerance. *IEEE Transactions on Software Engineering*, v. 1, no. 2 pp. 220-232, June 1975.

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

[34] D.J. Scales, K. Gharachorloo, C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[35] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 297-306, October 1994.

[36] M. L. Scott et. al. InterWeave: Object Caching Meets Software Distributed Shared Memory. *Work in Progress at the 17th Symposium on Operating Systems Principles*, December 1999.

[37] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, M. Scott. CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. *Proc. 16th Symposium on Operating Systems Principles (SOSP)*, October 1997.

[38] R. Strom, S Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, v. 3, no. 3, pp. 204-226, August 1985.

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

[40] G. Suri, B. Janssens, W. K. Fuchs. Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory. *Proc. 25th International Fault-Tolerant Computing Symposium (FTCS)*, pp. 279-288, June 1995.

[41] V. O. Tam, M. Hsu. Fast Recovery in Distributed Shared Virtual Memory Systems. *Proc. 10th International Conference on Distributed Computing Systems*, pp. 38-45, May 1990.

[42] Y. M. Wang. Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems. *Ph.D. Thesis*, University of Illinois at Urbana-Champaign, August 1993.

[43] Y. M. Wang, P. Y. Chung, I. J. Lin, W. K. Fuchs. Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, v. 6, no. 5, pp. 546-554, May 1995.

[44] Y. M. Wang. Consistent Global Checkpoints that Contain a Set of Local Checkpoints. *IEEE Transactions on Computers*, v. 46, no. 4, pp. 456-468, April 1997.

[45] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. 22nd International Symposium on Computer Architecture (ISCA)*, pp. 24-36, June 1995.

[46] K. L. Wu, W. K. Fuchs. Recoverable Distributed Shared Memory: Memory Coherence and Storage Structures. *IEEE Transactions on Computers*, v. 34, no. 4, pp. 460-469, April 1990.

[47] Y. Zhou, L. Iftode, K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 75-88, October 1996.