Lazy Garbage Collection of Recovery State for Fault-Tolerant Distributed Shared Memory

Florin Sultan, Thu D. Nguyen, *Member*, *IEEE Computer Society*, and
Liviu Iftode, *Member*, *IEEE Computer Society*

Abstract—In this paper, we address the problem of garbage collection in a single-failure fault-tolerant home-based lazy release consistency (HLRC) distributed shared-memory (DSM) system based on independent checkpointing and logging. Our solution uses laziness in garbage collection and exploits consistency constraints of the HLRC memory model for low overhead and scalability. We prove safe bounds on the state that must be retained in the system to guarantee correct recovery after a failure. We devise two algorithms for garbage collection of checkpoints and logs, checkpoint garbage collection (CGC), and lazy log trimming (LLT). The proposed approach targets large-scale distributed shared-memory computing on local-area clusters of computers. In such systems, using global synchronization or extra communication for garbage collection is inefficient or simply impractical due to system scale and temporary disconnections in communication. The challenge lies in controlling the size of the logs and the number of checkpoints without global synchronization while tolerating transient disruptions in communication. Our garbage collection scheme is completely distributed, does not force processes to synchronize, does not add extra messages to the base DSM protocol, and uses only the available DSM protocol information. Evaluation results for real applications show that it effectively bounds the number of past checkpoints to be retained and the size of the logs in stable storage.

Index Terms—Fault tolerance, distributed shared memory, checkpointing, log-based rollback recovery, garbage collection.

1 Introduction

In the last decade, an impressive amount of research has been conducted in software distributed shared memory (DSM), mostly aiming for performance (e.g., relaxed consistency models, lazy protocols, and communication hardware support) [15], [1], [5], [20], [21], [31], [30]. Advances in performance are making it possible for DSM systems to use very large clusters as a cost-effective platform for data-intensive, long-running applications. More recently, projects like InterWeave [7] propose a shared-memory programming model to support applications that run on wide-area clusters of heterogeneous machines (metaclusters). As cluster size and application running times increase, adding fault tolerance to a DSM-based cluster becomes critical. At the same time, to preserve performance, the fault tolerance support itself must be both light-weight and scalable.

A common approach to building fault-tolerant systems is rollback recovery, where the state of a computation is periodically saved to stable storage (checkpointed) and used to restart the computation in case of a failure. For distributed computations, there are two options for checkpointing [10]: 1) coordinated checkpointing, where all processes coordinate to save a globally consistent state at each checkpoint and 2) independent (uncoordinated) checkpointing, where checkpointing is purely a local operation and, as a result, the set of the most recent checkpoints may not represent a globally consistent state of the system.

Coordinated checkpointing has typically been used in recoverable DSM [6], [16], [22], [3], [8] because it is simpler and more efficient to implement in small-scale clusters and when nonfailed nodes are assumed to be always accessible [11]. For very large clusters and metaclusters, however, independent checkpointing becomes more practical than coordinated checkpointing because of the increasing cost of global coordination and the likelihood of temporary disconnections in communication. Nonblocking consistent checkpointing protocols could reduce the coordination overhead by allowing processes to take local checkpoints and continue execution without explicit (blocking) synchronization. However, such protocols have been shown to be nonoptimal, forcing processes to take unnecessary checkpoints [4]. An additional advantage of independent checkpointing is that a process can conveniently choose when to checkpoint. This autonomy enables application-level optimizations on the checkpoint size, using techniques like *memory exclusion* [27] to include in the checkpoint only regions of the address space strictly needed for recovery.

Log-based rollback recovery protocols [10] use checkpoints and log messages received by a process for replay during recovery. They can ensure that the maximum recoverable state is exactly the state of the system before a crash and do not force valid processes to rollback when independent checkpoints are used.

• *The authors are with the Department of Computer Science, Rutgers University, Piscataway, NJ 08854-8019.*
E-mail: {sultan, tdnguyen, iftode}@cs.rutgers.edu.

Manuscript received 19 Mar. 2001; accepted 28 Dec. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 113825.

In [34], we addressed the problem of how to integrate independent checkpointing and logging with a scalable software DSM protocol to build a fault-tolerant DSM system that can be deployed on large-scale clusters. Our system uses a Home-based Lazy Release Consistency (HLRC) DSM protocol [14], extended with a log-based recovery algorithm using exclusively independent checkpointing in order to efficiently tolerate single-fault failures. The choice of the base DSM coherence protocol is important because the protocol 1) must scale well with cluster size, 2) must have low memory overhead, freeing memory for fault tolerance related tasks (for example, logging), and 3) must be light-weight in terms of state maintained and, thus, incur less overhead for logging and checkpointing. The HLRC protocol has been shown to have these properties [37].

This paper presents the theoretical results on which the system described in [34] was based. To make independent checkpointing practical, we have to control the size of the logs and checkpoints in stable storage. With independent checkpointing, processes cannot automatically discard their logs and old checkpoints when taking a new checkpoint because failed peer processes may need state or log entries saved prior to the last local checkpoint. We need a scheme for garbage collection of obsolete checkpoints and logs without forcing processes to synchronize or otherwise create extra system traffic and which is robust to temporary disruptions in communication.

In this paper, we present our solution to this problem, which is to allow laziness into the garbage collection mechanisms and to determine safe bounds on the state to be retained by taking into account the consistency constraints of the memory model. Using this approach, we devise two algorithms, Lazy Log Trimming (LLT) and Checkpoint Garbage Collection (CGC), to control the size of the logs and the number of checkpoints. We prove the correctness of our scheme in the context of fault-tolerant HLRC and, thus, address the issue of augmenting it with a scalable garbage collection mechanism. We present an evaluation of our logging, checkpointing, and garbage collection algorithms on three update-intensive applications from the SPLASH-2 benchmark suite [36], with an emphasis on the effectiveness of LLT and CGC. The overhead of logging and checkpointing for recovery of the shared memory space is under 7 percent of the base execution time. Our algorithms are effective in controlling the amount of recovery state in the system: Shared pages from no more than three checkpoints had to be retained at their home nodes and at least 60 percent of the logs created were discarded. We believe that the ideas and techniques used by our algorithms can be extended to any large-scale distributed system, possibly over wide-area, that uses versioning for replica consistency.

The remainder of the paper is structured as follows: Section 2 presents related work in 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 data structures for recovery support and Section 5 proves the correctness of our log trimming and garbage collection algorithms. Experimental results from a performance evaluation of LLT and CGC are presented in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

Recoverable DSM systems that use independent checkpointing and logging in volatile or stable storage are described in [28], [35], [8], [26], [24], [23]. Coordinated checkpointing has been used in recoverable DSM systems like [6], [16], [22], [3], [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. Independent checkpointing with dependency tracking applied to DSM was explored in [17], [18]. A transactional recoverable DSM was designed in [12] by using techniques from database systems.

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).

The idea of volatile logs of *accesses* to shared memory was proposed in [28] 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 [35].

In [26], sender-based logging [19] with independent checkpointing is added to an entry-consistent, single-writer, object-based DSM system. Object copies are logged to volatile memory after every update. The simple consistency model, which does not allow multiple writers, along with logging of full object copies makes garbage collection straightforward: All processes trim their object logs eagerly, whenever a new checkpoint is created in the system. To support this scheme, each process broadcasts control information needed for trimming after every checkpoint.

The system described in [12] uses log-based coherency, a mechanism based on transactional database techniques. This scheme is applied to RVM [29], a persistent virtual-memory, to build a recoverable entry-consistent DSM that provides a transactional programming model. Propagation of transaction logs between nodes achieves coherency, while recoverability is ensured by the underlying persistent store to which committed logs are saved. Several methods for log trimming are discussed, including broadcasting control information, but only offline trimming is performed. In contrast to the above two systems, our fault-tolerant system supports a relaxed multiwriter memory model at page granularity, performs online garbage collection, and does not require global operations for garbage collection.

In [35], log-based recovery was first used in a Lazy Release Consistent (LRC) [20] 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 garbage collection can be 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 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, garbage collection operations do not require a process to take a checkpoint.

In [8], Costa et al. have integrated log-based fault tolerance support into TreadMarks [21], a DSM system that implements the LRC model. Their work is different from ours in that their system leverages the global garbage collection 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 can be discarded at a global checkpoint, so their system does not face the problem of dynamic log trimming and checkpoint garbage collection.

In [23], a coherence-centric logging and recovery scheme is proposed for an HLRC protocol. The logging scheme is based on that proposed in [35], but combines receiver-based with sender-based logging to eliminate page logging. It flushes the logs to stable storage at release points, an operation which may generate frequent accesses to stable storage. No garbage collection mechanism is proposed. In our fault-tolerant HLRC system, logs can be saved to stable storage infrequently, only at checkpoint time. Home nodes retain copies of pages from a window of past independent checkpoints to support fast recovery of any failed process. We develop garbage collection algorithms that effectively bound the amount of state retained in the system for recovery support.

3 Overview

In this section, we briefly describe our fault-tolerant home-based distributed shared memory system [34]. We present the fault-tolerant system model, the base DSM protocol, and the proposed recovery scheme based on independent checkpointing.

3.1 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 [32] 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. We do not consider logging of I/O interactions for recovery support. For all our purposes, we assume that the stable storage used by a process remains available after the failure of the process, which can be restarted on the same or a different node.

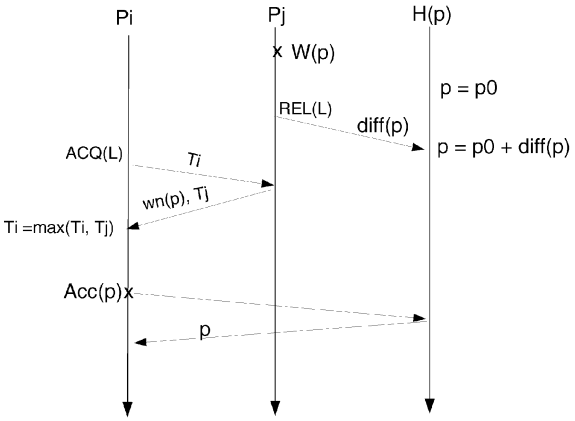

Fig. 1. Variables v_1 and v_2 mapped onto the chromosome.

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 software DSM, writes are completed at synchronization time [13]. Coherency data is propagated as page invalidations. The contents of a page is updated lazily at page fault time, either from its home [37] or from the last writer(s) [20]. 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 [21].

Lazy Release Consistency (LRC) is a variant of release consistency in which propagation of writes is delayed to the time of an acquire [20]. 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* [25] (denoted $\prec$) between intervals across nodes is captured as a vector of logical times called *vector timestamp* [20]. The vector timestamp T_i keeps track of intervals for which write notices were received by process P_i.

In the Home-based Lazy Release Consistency protocol (HLRC) [14], every shared page p has an assigned home $H(p)$ which maintains the most recent version of the page. Suppose (Fig. 1) that a process P_j has acquired a lock L before writing to page p (the write is labeled $W(p)$). 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 were applied to p; $p.v[i]$ advances with application of a diff from P_i.

Following the invalidation of p at the acquire, the nonhome process P_i records the version $p.v^N$ it $needs$ in case of an access, according to write notices it has received. 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 Fault-Tolerant HLRC

In fault-tolerant HLRC: 1) Processes take independent checkpoints (decisions of when and what to checkpoint are purely local decisions), 2) each process maintains logs of protocol data sent to its peers in volatile memory, and 3) a failed process will restart from its latest checkpoint and use logs from peer processes to deterministically replay its execution.

In our system, checkpointing can be transparent, or can be done at the request of the application. All logging operations take place transparently, only at synchronization points. We use *sender-based message logging (SBML)* [19] in which the sender of a message logs it in volatile memory. SBML has low overhead, as logging can be done out of the critical message path after the message is sent. Logs must be saved to stable storage at least on every checkpoint as they must survive a crash and restart from the latest checkpoint.

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 reexecution, 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 the previous normal execution. The execution replay needs only enforce that a *read access* to a page returns the same value, rather than indiscriminately applying all writes to the page.

Changes in the DSM protocol state at process P_i that 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. Also, 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 an older checkpoint of $H(p)$.

4 Data Structures for Recovery Support

In this section, we describe the data structures for recovery support (checkpoint timestamps, *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.

4.1 Checkpoint Timestamping

The basic idea of our garbage collection algorithms is to (partially) order checkpoints using vector timestamps: A process P_i *timestamps* a checkpoint with a vector T^i_{ckp}, equal to its vector timestamp T_i at the moment the checkpoint is taken.

Ideally, a *perfect* checkpoint timestamp T^i_{ckp} would be a *global vector time* T_g defined as $T_g[i] = T_i[i]\ \forall i$ (i.e., T_g describes the global state of the system at the moment the checkpoint was taken, assuming instant knowledge of logical times of all nodes). This is because, when reexecuting after a restart from that checkpoint, the state of process P_i is guaranteed not to be affected by events at some other process P_j that happened (and were logged) before $T_g[j]$. These events with earlier timestamps will not be needed for recovery. The closer T^i_{ckp} is to T_g, the better it reflects the global state and, thus, allows a more efficient trimming of recovery logs at peer processes P_j. Since, in practice, T_g is not available, we use the best approximation available to a process about the state of the system without any extra synchronization, which is its local vector timestamp, i.e., $T^i_{ckp} = T_i$ at the moment the checkpoint is taken.

4.2 Support for Synchronization 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.

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.

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 reexecution, 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 $acq\_sent\_log[i]$ 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 reexecution. 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 and reside on distinct nodes. As a result, a process does not actually 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 Support for 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 during recovery by locally and dynamically applying an ordered sequence of logged diffs to the page to obtain a copy which is coherent according to the protocol semantics.

In HLRC, a process P_i where a page p is invalid $needs$ a version $p.v^N$, according to write notices received up to its first attempted access to p after the invalidation. This is the minimal version of page p that P_i will request from $H(p)$ on its access and miss to p. During normal operation, $H(p)$ may reply with the requested version or with a higher version of p. In the latter case, HLRC guarantees that the copy of p at $H(p)$ incorporates only concurrent writes which are not conflicting with the faulting access by P_i (i.e., they did not happen-before according to the partial order enforced by synchronization operations). For this reason, during replay, it is sufficient for the access to p to be serviced with the 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 would be the case if pure message logging of page transfers were used.

In our system a page p is only checkpointed by its home $H(p)$. The home retains a sequence p_{ckp} of copies of p from past checkpoints. Any writer P_j of p logs every diff it creates 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 reexecution, a recovering process 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 dynamically applies partially ordered diffs obtained from all writers' logs *diff_log(p)*.

The following lemma provides safety conditions for a $maximal$ starting copy p_0 (i.e., to which nothing can be added without potentially compromising 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. In the following, we define the $\leq$ relation on a pair of vectors to yield true iff the component-wise $\leq$ relationship holds for all elements.

Lemma 1 (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 a safe $p_0 \in p_{ckp}$ must have a version $p_0.v \leq T^i_{ckp}$. 2) To ensure the correct replay of p starting from the above p_0, any writer P_j must supply diffs with timestamps $diff.T[j] > p_0.v[j]$.*

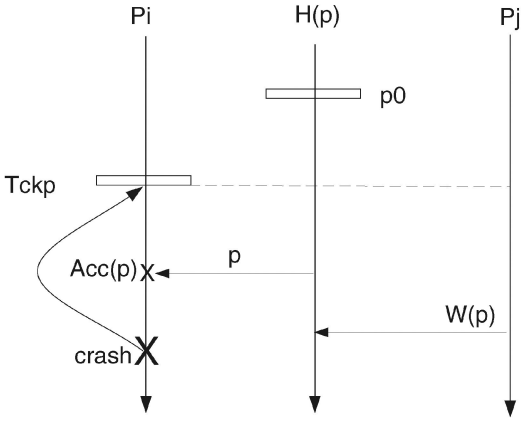

Fig. 2. A randomly generated task graph and height methods.

Proof. We look at the first access Acc(p) of P_i to page p after a crash and restart from its latest checkpoint time-stamped T^i_{ckp}, as shown in Fig. 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 $H(p)$.)

1. 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 with respect to the restart checkpoint of P_i. In Fig. 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 $T_j[j] > T^i_{ckp}[j]$, generating a version with $p.v[j] > T^i_{ckp}[j]$. Therefore, a page with $p_0.v \leq T^i_{ckp}$ is always safe for replay.[1]
2. Next, we show that if we select p_0 from the $last$ checkpoint of $H(p)$ in which the version $p_0.v \leq T^i_{ckp}$, then this p_0 is *sufficient* for a correct replay of Acc(p) under the conditions of Part 2 of the lemma. Note that, according to Part 1 of the lemma, such a p_0 would be *safe* for replay.

We prove p_0 is sufficient for two cases, depending on whether, during normal operation, p was valid (*Case A*) or not (*Case B*) at the time of Acc(p).

Case A. Suppose p was invalid at Acc(p) during normal operation, so it is also invalid after the restart of P_i. Then, p's page table entry in the checkpoint records $p.v^N$, the $minimal$ version that P_i will need on Acc(p) during recovery. This version is minimal in the sense that servicing Acc(p) with a page that has a lower version (missing some previous writes) would compromise the correctness of the protocol.

We consider two instances, depending on whether the checkpointed p_0 incorporates all the writes needed for P_i's access or not, i.e., whether either $p.v^N \leq p_0.v$ or there exists j such that $p_0.v[j] < p.v^N[j]$.

1. Note that, under LRC, the condition $p_0.v \leq T^i_{ckp}$ may not be strictly necessary. A safe p_0 can be any copy of p containing writes $W(p)$ that occurred after T^i_{ckp}, but which are not related to Acc(p) under the $\prec$ order.

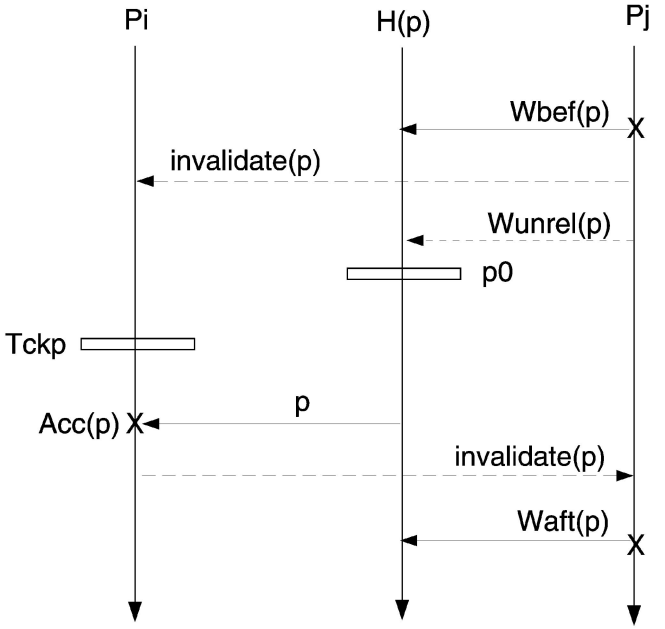

Fig. 3. A task graph and priority methods.

Case A.1. The checkpointed p_0 contains all writes needed by Acc(p), i.e., $p.v^N \leq p_0.v$. This case is trivial, in the sense that p_0 alone is sufficient for replay.

Fig. 3 shows a simplified scenario in which P_i first synchronizes with P_j, resulting in the invalidation of p. 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 alone 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 the moment of the invalidation, due to LRC constraints.

Note that if, for some j, $p.v^N[j] < p_0.v[j]$, the higher version of $p_0.v[j]$ corresponds to writes of P_j not related to Acc(p), and included in p_0 after $W_{bef}(p)$. The safety condition from Part 1 of the Lemma ($p_0.v[j] \leq T^i_{ckp}[j]$) always ensures that no conflicting writes are incorporated in p_0. 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 Fig. 3). Then, the writes of P_j (labeled $W_{unrel}(p)$ in Fig. 3) between the invalidation of p and the moment of W_{aft} are concurrent with Acc(p) but not related to it. Some of them may be included in the checkpointed copy p_0 of $H(p)$, as shown in the figure, making $p_0.v[j]$ larger than $p.v^N[j]$, but this would not affect the correct replay of Acc(p).

Case A.2. The checkpointed p_0 does not contain all related writes needed by Acc(p), i.e., there exists j such that $p_0.v[j] < p.v^N[j]$.

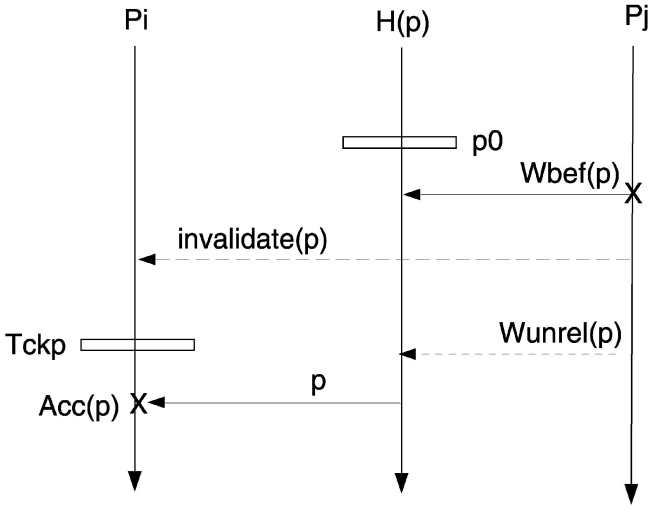

Fig. 4. Average computation times for different population sizes.

This means (see for example Fig. 4) that 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 of p_0. 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 have occurred at logical times $T_j[j] > p_0.v[j]$, so P_j must supply all diffs with a timestamp $diff.T[j] > p_0.v[j]$.

Case B. Suppose p was valid at the moment of Acc(p) during normal operation. Since Acc(p) is the first access after the checkpoint, p was also valid when P_i took the checkpoint. Then, there must have been an access of P_i to

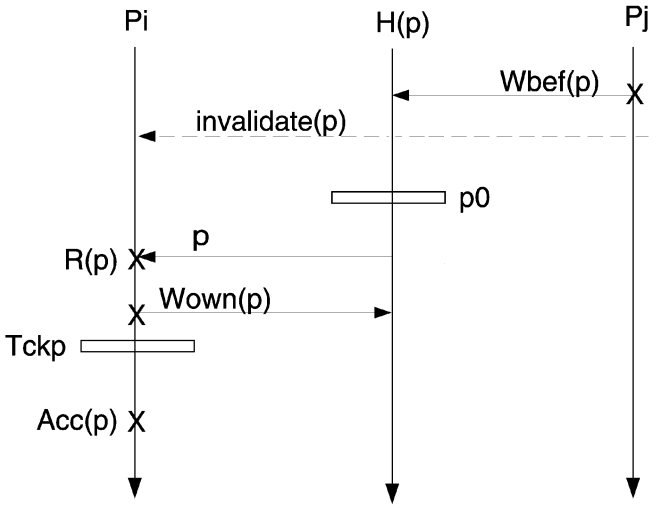

Fig. 5. Average speedup for different population sizes.

p after it received the invalidation for p but before its last checkpoint at T^i_{ckp}. Suppose that such an access was a read $R(p)$ and that there was a write $W_{own}(p)$ of P_i such that $R(p) \prec W_{own}(p) \prec T^i_{ckp} \prec Acc(p)$ (Fig. 5).

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

We must only ensure that the replay of Acc(p) is correct with respect to all the writes like W_{own} issued by P_i. The starting p_0 alone cannot be used for replay since it may not contain W_{own}. For correct replay, P_i must save diffs for its writes that it created and logged after $H(p)$ took its checkpoint of page p_0, i.e., diffs with $diff.T[i] > p_0.v[i]$. □

Note that Lemma 1 links the starting page p_0 with its corresponding diffs: 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.

5 Garbage Collection of Recovery State

In this section, we prove the core results that ensure correct recovery of any process while logs and checkpoints are discarded to reclaim space. Our approach relies on checkpoint timestamping as described in Section 4 and allows for laziness in garbage collection operations. The mechanisms we propose allow a process to locally initiate and execute these operations without involving other processes.

5.1 Checkpoint Timestamping Issues

Recall that, in our system, the most recent checkpoint (the restart checkpoint) of a process P_i has a timestamp vector T^i_{ckp} equal to the vector timestamp T_i at the moment the checkpoint is taken. In a previous work [33], we showed how the global vector time T_g (or, in practice, an approximation of T_g that preserves a total order on the set of all checkpoint timestamps) can be used to timestamp checkpoints and perform garbage collection of recovery state. We proved a set of rules for log trimming and checkpoint garbage collection assuming instant availability of the global vector time for timestamping checkpoints. In this paper, we prove the garbage collection rules for the case where $T^i_{ckp} = T_i$ at the time of checkpoint.

We note that, in practice, each P_i maintains, for any P_j, its last known value $\hat{T}^j_{ckp}$ of P_j's checkpoint timestamp T^j_{ckp}. The policy of propagating the T^j_{ckp} of the last checkpoint from P_j to other nodes is flexible: It can be broadcast periodically, sent in reply to a query, piggybacked on protocol messages, etc. In this paper, we are not concerned with proposing such a policy or analyzing its impact. The only thing to note is that our log trimming and garbage collection algorithms are robust with respect to checkpoint timestamps: 1) Their correctness is not impeded by stale timestamps and 2) the known value of a given checkpoint timestamp does not need to be consistent across processes (so the algorithms do not require processes to synchronize). In Section 6, we show that our scheme achieves good performance using piggybacking on protocol messages.

5.2 Synchronization Log Trimming

The next two lemmas provide bounds on intervals corresponding to the oldest entries that a process must retain from its write notice and acquire logs in order to be able to support the recovery of another process.

5.2.1 Write Notice Trimming

Lemma 2 (Wn log trimming). *A process P_i can support the wn replay of any process P_j restarting from a checkpoint time-stamped T^j_{ckp} by retaining only entries of wn_log corresponding to intervals starting from $ckp\_int = \min_{j \neq i} T^j_{ckp}[i] + 1$.*

Proof. During reexecution, 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 ith element of P_j's checkpoint timestamp (which is its vector timestamp), $T^j_{ckp}[i]$, 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 $T^j_{ckp}[i]$. 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. □

As noted before, since the $\hat{T}^j_{ckp}$ vectors available at P_i may not be consistent with the real checkpoint timestamps, the local estimate of ckp_int may not be exact. If this is the case, it defines a superset of the minimal set of intervals needed to support recovery at any node since it is computed as a $\min$ over logical time (monotonically increasing).

5.2.2 Acquire Log Trimming

Lemma 3 (ACQ log trimming). *A process P_i can support the ACQ replay of a process P_j restarting from a checkpoint with timestamp T^j_{ckp} by retaining only entries of $acq\_sent\_log[j]$ with $T_j[j] > T^j_{ckp}[j]$ 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 $T_i[i] > T^i_{ckp}[i]$.*

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 $T_j[j] \leq T^j_{ckp}[j]$ (recall that the checkpoint timestamp is equal to the vector timestamp at the moment the checkpoint was taken).

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 $T_i[i] > T^i_{ckp}[i]$. To back it up, P_i must retain from $acq\_rcvd\_log[j]$ only entries for which $T_i[i] > T^i_{ckp}[i]$. □

In practice, because the checkpoint timestamps $\hat{T}^j_{ckp}$ known to P_i may lag behind the timestamps T^j_{ckp} of the most recent checkpoints, trimming of $acq\_sent\_log[j]$ may not be optimal. Trimming of $acq\_rcvd\_log[j]$ is always optimal as it uses only the local checkpoint timestamp.

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

Lemma 1 proved conditions for a home $H(p)$ 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 depends on the particular page itself (it involves individual page versions). Also note that, for efficiency reasons, a home node H may choose to keep a $window$ of past checkpoints of all pages it manages, starting with the oldest checkpoint determined by Lemma 1.

The following theorem simply extends Lemma 1 to provide conditions for the garbage collection of checkpointed copies of a page that are not needed for recovery of any process. It also shows that, after $H(p)$ performs a CGC operation, a writer can, at a later moment, "lazily" trim its diff logs using information from $H(p)$, namely the version of p_0.

Theorem 1 (CGC and LLT). *1) A home process H can support the page replay of any process if it retains page p_0 from a checkpoint such that $p_0.v \leq T_{min} = \min_{j \neq H} T^j_{ckp}$. 2) A writer of page p, P_i, can support recovery replay for p by retaining only diff_log(p) entries with $diff.T[i] > p_0.v[i]$.*

Proof. Directly from Lemma 1 by considering, for complete recovery support, processes $P_j, j \neq H$. □

Note that the bounds for retaining checkpoints and diff logs in Theorem 1 (T_{min} and, indirectly, $p_0.v$) depend on the checkpoint timestamps T^j_{ckp} of the restart checkpoints. In practice, the efficiency of CGC and LLT depends on the $\hat{T}^j_{ckp}$ timestamps available at P_i, which could be stale with respect to the most recent checkpoints of peer processes P_j.

5.3.1 Aggregate Page Management for LLT and CGC

Lemma 1 and its proof implicitly assume that management of pages by homes is done on a per-page basis. This represents the optimal case, when a home process retains for *each* page exactly the restart version. As a result, for different pages, their restart versions may come from different checkpoints. In practice, such an implementation would have to selectively discard unneeded pages from checkpoints.

In practice, an implementation may trade optimal storage consumption for simplicity and efficiency and it may choose to discard a whole checkpoint only when none of its pages is needed, instead of selectively discarding per-page checkpointed copies.

To simplify such an approach, *aggregate page management* is possible by using the *page version timestamp* V_H of a home process H. Any home process H records in the checkpoint a *page version timestamp* V_H, representing the largest diff versions that H has received from all processes, i.e., $V[i] = \max_p p.v[i]$ over all pages p homed by H.

The following versions of Lemma 1 and Theorem 1 establish the use of a single vector V_H for tracking dependencies between restart checkpoints and page checkpoints to be retained by a home process and between home checkpoints and diffs to be retained by page writers.

Lemma 4 (Maximal p_0 for Aggregate Management). *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 (i.e., most recent) checkpoint of H(p) for which $V_H \leq T^i_{ckp}$. 2) To ensure the correct replay of p starting from the above p_0 retained by home H, any writer P_j must supply diffs with timestamps $diff.T[j] > V_H[j]$.*

Proof. 1) In Part 1 of lemma 1, since V_H is an absolute bound on all page versions in a checkpoint, we have $p_0.v \leq V_H$. Therefore, the *safety* condition $p_0.v \leq T^i_{ckp}$ for the checkpoint from which p_0 is selected is automatically satisfied by a checkpoint with $V_H \leq T^i_{ckp}$. 2) Because the vector V_H tracks the latest diffs received by home H from all processes by the time of the checkpoint, any future (i.e., occurring after the checkpoint was taken) writes by some writer P_j can only generate diffs with a timestamp $diff.T[j] > V_H[j]$. Therefore, throughout the proof of Part 2 of Lemma 1, $V_H[j]$ can safely replace the per-page bound $p_0.v[j]$ for diffs to be retained by P_j. □

Extending the previous Lemma, the following theorem describes how garbage collection of whole page checkpoints and diff log trimming can be performed at a process by using the aggregate page version timestamp V_H instead of individual page versions.

Theorem 2 (Aggregate CGC and LLT). *1) A home process H can support the page replay of any process if it retains whole page checkpoints starting with the most recent checkpoint having a page version timestamp V_H such that $V_H \leq T_{min} = \min_{j \neq H} T^j_{ckp}$. 2) A writer of page p, P_i, can support recovery replay for p by retaining only diff_log(p) entries with $diff.T[i] > V_{H(p)}[i]$.*

An important observation on results proven in Theorems 1 and 2 is that, approximate values of the checkpoint timestamps $\hat{T}^j_{ckp} \leq T^j_{ckp}$ can be tolerated in the bounds T_{min} on $p_0.v$ and V_H, respectively. For example, in Theorem 1, the approximate $\hat{p}_0.v \leq \hat{T}_{min} \leq T_{min}$ can only push p_0 to a checkpoint older than the ideal (optimal) checkpoint computed based on the exact T_{min}. If an older checkpoint with $\hat{p}_0.v < p_0.v$ is retained, Part 2 of Lemma 1 ensures that p can still be reconstructed by retaining necessary diffs for the older approximation of p_0.

5.4 An Example

Fig. 6 shows an example of how whole checkpoints are garbage collected and diff logs are trimmed using Theorem 2 (a similar example can be devised using Theorem 1 for a given page). For ease of presentation, assume that checkpoint timestamps in the example are such that they can be totally ordered about an imaginary global time axis running vertically, as shown in the figure. Vertical brackets mark the $window$ of checkpoints from which page copies are to be retained by each home. The upper limit of a checkpoint window is set by aggregate page management using page version timestamps of Theorem 2.

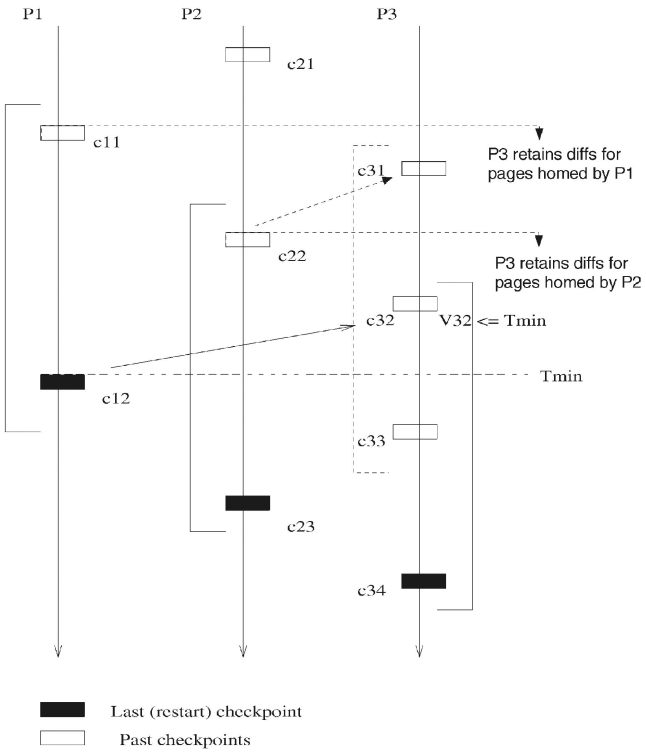

Fig. 6. Crossover type and the average speedup.

In this example, process P_3 is about to take checkpoint c_{34}, at which time it also performs a CGC operation. The solid and dotted arrows show the dependency (from T_{min} to the first checkpoint to be retained by P_3, using its page version timestamps V_3) 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 Theorem 2 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 page version timestamps of c_{11} and c_{22}.

5.5 Using Approximate Information in Computing Trimming Bounds

Lemmas 2, 3, and Theorems 1 and 2 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: 1) the checkpoint timestamp T^j_{ckp} of the restart checkpoint, for trimming the wn and ACQ logs and for its CGC, and 2) $p_0.v[i]$ (or $V_j[i]$), if P_i is a writer to a page homed by P_j, for its LLT.

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^i_{ckp} and one integer (a per-page version, $p_0.v[j]$, or the aggregate value $V_i[j]$). Piggybacking, or any other form of lazy propagation, may make this information stale at a particular process. The corresponding values that P_j uses for trimming logs ($\hat{T}^i_{ckp}[j]$, $\hat{T}^i_{ckp}[i]$, and $\hat{p}_0.v[j]$ or $\hat{V}_i[j]$) may become obsolete, depending on the sharing/communication pattern. The logs of write notices and the synchronization logs will be trimmed less efficiently by P_j if $T^i_{ckp}[j]$ and $T^i_{ckp}[i]$ are stale (Lemmas 2 and 3). For the page checkpoints, the bound $\hat{T}_{min}$ may be less than T_{min} of Theorems 1 and 2 because of stale $\hat{T}^i_{ckp} \leq T^i_{ckp}$ and it may force process P_j, as a home of some page p, to push back its checkpoint window to include an older checkpoint than actually needed. This may also increase the diff log size at some writer P_i by the feedback effect through $p_0.v[i]$ ($V_j[i]$, respectively).

6 Evaluation of CGC and LLT

To evaluate LLT and CGC, we have extended the HLRC protocol to include: 1) LLT, 2) a checkpointing policy to decide when to checkpoint at each node, and 3) a simulation of CGC.[2] Our goals are to: 1) evaluate the impact of checkpointing and logging on application performance and 2) assess the efficiency of LLT and CGC in terms of both volatile and stable storage requirements, using as metrics the diff log size and the checkpoint window size.

In all experiments, log trimming, garbage collection of checkpoints, and saving logs to stable storage take place only at checkpoint time. This is a limitation that we impose in order to stress the system to the greatest extent. Since

2. The prototype used in our experiments generates checkpoints to stable storage but does not implement recovery and checkpoint storage management. Nevertheless, this implementation is sufficient to evaluate CGC on real-world applications.

TABLE 1
Applications Used and Their Characteristics

Application and problem size	*Shared memory (MB)*	*Base execution time (s)*
Barnes 256 k	43	1,663
Water-Nsquared 19,683	12.6	1,634
Water-Spatial 256 k	257.3	2,569

TABLE 2
Message Traffic Overhead of CGC and LLT in an HLRC DSM System

Application and problem size	*HLRC traffic (MB)*	*CGC traffic (MB)*	*% overhead*
Barnes 256 k	2,224	3.3	0.15
Water-Nsq. 19,683	68.5	0.1	0.2
Water-Sp. 256 k	174.7	0.5	0.25

TABLE 3
Performance of the Independent Checkpointing Scheme with CGC and LLT in an HLRC DSM System

Application and problem size	*L %*	*Ckp. taken*	*Execution time*		*Time logging (s)*	*Time disk write (s)*	*% overhead*
			(s)	*% increase*			
Barnes 256 k	100	6 - 10	2,677	61	16.2	96.8	6.8
Water-Nsq. 19,683	10	9	1,644	0.6	0.9	5.3	0.4
Water-Sp. 256 k	10	5	2,737	7	13	79.4	3.6

The last three columns show the time spent logging and writing to disk in absolute value and as overhead relative to the base execution time.

LLT and CGC mechanisms are totally decoupled from checkpointing operations and can run at any point during execution, a more aggressive implementation could asynchronously write logs to stable storage, overlapping saving logs with computation. In our experiments, writing logs to stable storage was measured as added overhead.

We have selected three applications from the SPLASH-2 benchmark suite [36], Barnes, Water-Nsquared, and Water-Spatial, to drive our prototype system. We chose these applications because they have the longest running times in the suite, have various memory requirements, have a fair amount of synchronization, and generate large volumes of diffs, therefore presenting the worst-case scenario for LLT. We had initially run our experiments on all programs in the SPLASH-2 suite and eliminated those for which we could not scale the running times or that were generating small logs or no logs at all. They are irrelevant to our log garbage collecting scheme. We modified Barnes to run for 60 steps, while, for the other applications, we used the default parameters in the benchmark, except for increasing problem size. Table 1 shows the shared memory footprint of each application and the execution times with the base protocol (without fault tolerance support).

The checkpointing policy used in the experiments was based on limiting the log size in volatile memory: A checkpoint is taken when the size of volatile logs exceeds a threshold expressed as a fraction L of the application's shared memory footprint. The checkpointing decision is independently made by each node and is transparent to the application.

All experiments were run on a cluster of eight 300 MHz Pentium II PCs with 512 MB of memory each, running Linux. Communication is performed over a Myrinet LAN [2], using user-level virtual memory-mapped communication (VMMC) [9], to which we added fault tolerance support. Logs and checkpoints were saved to the local disk. We consider only the diff logs for trimming as they consume the largest amount of space. Results are for one run of each application with data averaged over all nodes.

6.1 Performance with Fault-Tolerant HLRC

Table 2 shows that the message traffic overhead of the LLT/CGC control data piggybacked on protocol messages is negligible relative to the base protocol traffic. Table 3 shows the impact of integrating support for recovery of the DSM shared space on application performance. For each application, with logging and checkpointing enabled, we show:

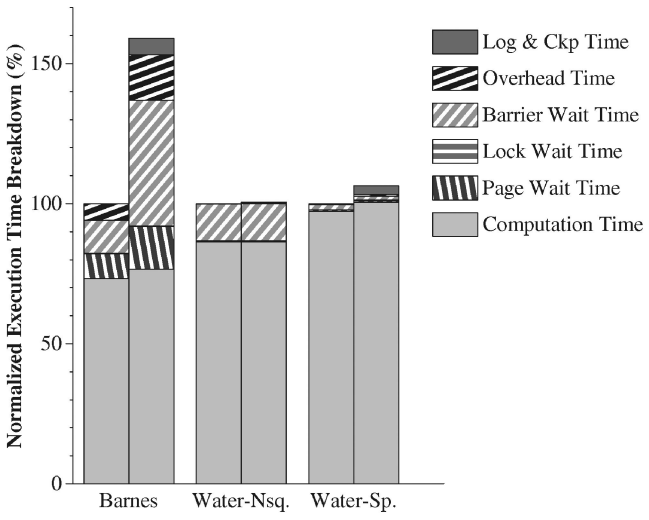

Fig. 7. Average time for each type of crossovers.

1. the memory usage threshold L allowed for logs by the log-overflow checkpointing policy (as a percentage of the shared memory size),
2. number of checkpoints taken,
3. execution time,
4. percentage of its increase over the base execution time,
5. overhead of logging,
6. overhead of writing to stable storage, and
7. overhead over the base execution time represented by the time to log and write to disk.

To estimate the overhead of writing to stable storage, at every checkpoint, we write to the local disk the pages homed by a process, along with its volatile logs. This accounts for the direct overhead introduced by fault tolerance support in the DSM system. We chose to only assess the overhead of checkpointing the DSM space as other components of the checkpointing overhead (e.g., checkpointing private data of a process) are unavoidable and therefore present in any fault-tolerant system.

Table 3 shows that the time spent with DSM logging and checkpointing is fairly small in all cases. This is also reflected by the small increase in running time, except for Barnes, which takes a performance hit of about 60 percent. To see where this severe degradation in performance comes from, we measured various components of the execution time both for the base protocol and with logging and checkpointing enabled.

Fig. 7 compares the averaged breakdown of execution time when running with base HLRC (left bar) and with fault-tolerant HLRC (right bar). The graphs are normalized with respect to the base execution times. The measured overheads are: time spent waiting for pages from home nodes, time spent waiting for locks, time spent waiting at barriers, protocol overhead time (including execution of page fault and message handlers, and of synchronization primitives), and time spent with logging and checkpointing DSM state.

For Barnes, we see that barrier waiting times have the largest contribution to the performance degradation observed, increasing from 12 percent to 28 percent of the execution time. The reason is an imbalance in the distribution of homed pages and diffs (and therefore logs) generated across nodes: in our sample run the volume of logs created varied from 290 to 460 MB across nodes. With a log-overflow checkpointing policy, this imbalance spreads checkpoints unevenly over time across processes. Since Barnes has many barriers, it is likely that processes checkpoint in *different* intervals delimited by consecutive barriers. Therefore, the overhead incurred by only one node or a small set of nodes checkpointing in some interval adds to the barrier waiting time of all other processes. The reason is that the checkpointing policy used is unaware (and does does not take advantage) of the intense global synchronization in the application. A checkpointing policy that forces all processes to checkpoint in the same barrier intervals would be better suited for applications with a large number of barriers. For the other applications, which have fewer barriers and have regular access and update patterns, this effect is not visible.

6.2 Efficiency of CGC and LLT

Table 4 shows how efficient CGC and LLT are in limiting the amount of state in stable storage. We measured, per node:

1. maximum size of the checkpoint window (W_{max}),
2. maximum amount of stable storage consumed by diff logs,
3. total amount of checkpointed pages and diff logs written to stable storage,

TABLE 4
Overall Efficiency of CGC and LLT in an HLRC DSM System

Application and problem size	W_{max}	*Max log disk (MB)*	*Disk traffic (MB)*	*Logs created (MB)*	*Saved logs*		*Discarded logs*	
					(MB)	*%*	*(MB)*	*%*
Barnes 256 k	3	80	390.2	371.2	348.3	94	281	76
Water-Nsq. 19,683	3	3.5	28.4	14.2	14.2	100	11.4	80
Water-Sp. 256 k	3	75.7	339.2	178.3	178.3	100	102.9	58

Disk traffic is generated by checkpointing homed pages and saving logs after in-memory trimming.

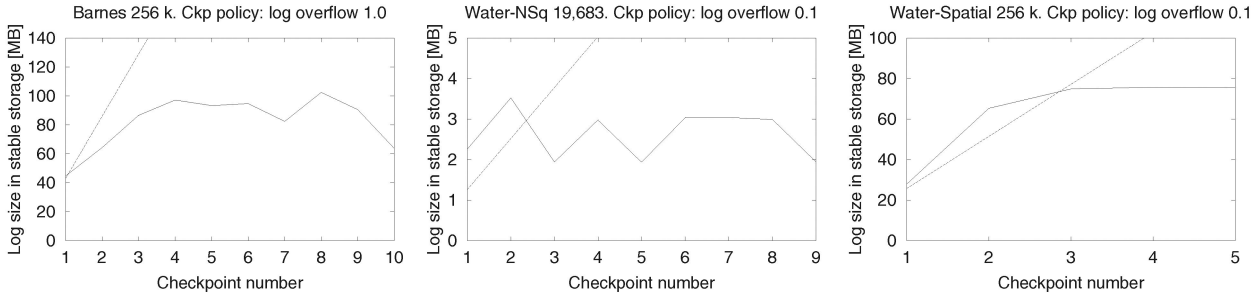

Fig. 8. Average speedup for each mutation type.

4. size of volatile logs created during execution,
5. size of volatile logs saved in stable storage,
6. percentage of saved logs from the volatile logs created,
7. size of logs discarded as a result of LLT,
8. percentage of discarded logs from volatile logs created.

The checkpoint window size is at most three, showing that CGC is effective even with the lazy propagation of information. The good overall efficiency of our LLT scheme can be seen from the large fraction of logs discarded. The smaller value in the case of Water-Nsquare is only due to the small number of checkpoints taken. Note that almost no trimming is done in memory. This means that most of the time the home nodes are forced to retain at least two checkpoints, otherwise they could just discard their volatile logs at the time they take a new checkpoint (this assumes homes write to their homed pages, which is the case with all three applications).

Fig. 8 plots the variation of diff log size on stable storage against checkpoints for a single node, showing that our scheme effectively bounds the log size over time. The straight line in each graph represents the theoretical growth of the log without LLT. In our implementation, sampling the size of the log takes place only at synchronization points in the application, as LLT is completely transparent. Due to this imprecision, the actual log size suffers from a minor start-up effect: With the first checkpoint, more than the threshold size of logs might be saved. However, as can be seen with Water-Nsquared and Water-Spatial, this effect is quickly canceled out by the effective trimming and within three checkpoints the total log size falls under the theoretical unbounded increase without LLT.

Water-Spatial is an interesting case as it is very regular and the volatile log-size threshold policy forces checkpoints in every iteration. The regularity effect can be observed in Fig. 8 in the gradual build-up of the saved log in the first two checkpoints, which retains the necessary state for recovery of the last iteration. After this point, every iteration adds and discards about the same amount of logs to/from stable storage. This behavior is expected as, in the first iteration, nodes perform computations and generate diffs for pages they become home for on the first access. Some trimming is performed at the second checkpoint as writers start learning about the checkpoint windows of home nodes. In the next iteration, the trimming information has reached all writers and, starting from the third checkpoint on, diffs which are no longer needed for recovery of one past iteration are massively discarded at every new checkpoint, causing the curve to flatten out. This behavior exhibits a self-synchronizing effect of the independent checkpoints in the case of Water-Spatial, without any explicit checkpoint coordination.

7 Conclusions

This paper describes the design and implementation of a fault-tolerant distributed shared memory (DSM) protocol based on independent checkpointing and logging. Independent checkpointing is particularly well-suited to large LAN-based clusters, as well as metaclusters over a WAN. In order to retain its advantages in such environments, the protocol must control the size of the logs and checkpoints without global coordination and must be robust to temporary disconnections in communication between nodes.

In this paper, we have specifically addressed the problem of garbage collection of the recovery state for a home-based lazy release consistency DSM protocol. We have defined the minimal state of the protocol that must be checkpointed and logged to support recovery from single-fault failures. We have proven safe bounds on the state that must be retained to ensure recoverability of the system by exploiting laziness and using only locally available protocol information. Based on these results, we have developed two garbage collection algorithms, lazy log trimming (LLT) and checkpoint garbage collection (CGC), that do not require processes to synchronize.

We have implemented our proposed algorithms in a fault-tolerant DSM system and evaluated their performance using three long-running and update-intensive applications from the SPLASH-2 benchmark suite. The experimental results show that our logging and checkpointing scheme has low overhead and that garbage collection effectively bounds the size of the log and the number of checkpoints that must be retained.

Acknowledgments

This work was supported in part by the US National Science Foundation CISE-9986046, ANI-0121416, USENIX and Rutgers University Information Sciences Council Research Grants. The authors are grateful to Xiang Yu from Princeton University, who helped us to extend the communication library with fault tolerance support. The authors also thank Murali Rangarajan for insights into protocol and application

behavior, and for all the help with the experimental part. The experimental results included in this paper were presented at Supercomputing 2000, Dallas, Texas, 4-10 November, 2000.

REFERENCES

[1] R. Bianchini, L.I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim, "Hiding Communication Latency and Coherence Overhead in Software DSMs," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-7),* Oct. 1996.

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

[3] G. Cabillic, G. Muller, and I. Puaut, "The Performance of Consistent Checkpointing in Distributed Shared Memory Systems," *Proc. 14th Symp. Reliable Distributed Systems (SRDS-14),* Sept. 1995.

[4] G. Cao and M. Singhal, "On Coordinated Checkpointing in Distributed Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 12, pp. 1213-1225, Dec. 1998.

[5] J.B. Carter, J.K. Bennet, and W. Zwaenepoel, "Implementation and Performance of Munin," *Proc. 13th Symp. Operating Systems Principles (SOSP-13),* Oct. 1991.

[6] J.B. Carter, A.L. Cox, S. Dwarkadas, E.N. Elnozahy, D.B. Johnson, P. Keleher, S. Rodrigues, W. Yu, and W. Zwaenepoel, "Network Multicomputing Using Recoverable Distributed Shared Memory," *Proc. IEEE Int'l COMPCON Conf. '93,* Feb. 1993.

[7] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M.L. Scott, "InterWeave: A Middleware System for Distributed Shared State," *Proc. Fifth Workshop Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR 2000),* May 2000.

[8] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro, "Lightweight Logging for Lazy Release Consistent Distributed Shared Memory," *Proc. Second USENIX Symp. Operating Systems Design and Implementation (OSDI-2),* Oct. 1996.

[9] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li, "Software Support for Virtual Memory-Mapped Communication," *Proc. 10th Int'l Parallel Processing Symp. (IPPS-10),* Apr. 1996.

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

[11] E.N. Elnozahy and D.B. Johnson, "The Performance of Consistent Checkpointing," *Proc. 11th Symp. Reliable Distributed Systems (SRDS-11),* Oct. 1992.

[12] M.J. Feeley, J.S. Chase, V.R. Narasayya, and H.M. Levy, "Integrating Coherency and Recoverability in Distributed Systems," *Proc. First Symp. Operating Systems Design and Implementation (OSDI-1),* Nov. 1994.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Int'l Symp. Computer Architecture (ISCA-17),* May 1990.

[14] L. Iftode, "Home-Based Shared Virtual Memory," PhD thesis, Princeton Univ., June 1998.

[15] L. Iftode and J.P. Singh, "Shared Virtual Memory: Progress and Challenges," *Proc. IEEE,* vol. 83, no. 3, Mar. 1999.

[16] G. Janakiraman and Y. Tamir, "Coordinated Checkpointing Rollback Error Recovery for Distributed Shared Memory Multicomputers," *Proc. 13th Symp. Reliable Distributed Systems (SRDS-13),* Oct. 1994.

[17] B. Janssens and W.K. Fuchs, "Reducing Interprocessor Dependence in Recoverable Distributed Shared Memory," *Proc. 13th Symp. Reliable Distributed Systems (SRDS-13),* Oct. 1994.

[18] B. Janssens and W.K. Fuchs, "Ensuring Correct Rollback Recovery in Distributed Shared Memory Systems," *J. Parallel and Distributed Computing,* Oct. 1995.

[19] D.B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *Proc. 17th Int'l Fault-Tolerant Computing Symp. (FTCS-17),* June 1987.

[20] P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. 19th Int'l Symp. Computer Architecture (ISCA-19),* May 1992.

[21] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proc. Winter '94 USENIX Conf.,* Jan. 1994.

[22] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A Recoverable Distributed Shared Memory Integrating Coherency and Recoverability," *Proc. 25th Int'l Symp. Fault-Tolerant Computing Systems (FTCS-25),* June 1995.

[23] A. Kongmunvattana and N.-F. Tzeng, "Coherence-Centric Logging and Recovery for Home-Based Software Distributed Shared Memory," *Proc. 1999 Int'l Conf. Parallel Processing (ICPP '99),* Sept. 1999.

[24] A. Kongmunvattana and N.-F. Tzeng "Lazy Logging and Prefetch-Based Crash Recovery in Software Distributed Shared Memory Systems," *Proc. 13th Int'l Parallel Processing Symp. (IPPS-13),* Apr. 1999.

[25] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.

[26] N. Neves, M. Castro, and P. Guedes, "A Checkpoint Protocol for an Entry Consistent Shared Memory System," *Proc. 13th Symp. Principles of Distributed Computing (PODC-13),* Aug. 1994.

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

[28] G.G. Richard III and M. Singhal, "Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory," *Proc. 12th Symp. Reliable Distributed Systems (SRDS-12),* Oct. 1993.

[29] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J. Kistler, "Lightweight Recoverable Virtual Memory," *ACM Trans. Computer Systems,* vol. 12, no. 4, pp. 33-57, Feb. 1994.

[30] D.J. Scales, K. Gharachorloo, and C.A. Thekkath, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-7),* Oct. 1996.

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

[32] R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 3, pp. 204-226, Aug. 1985.

[33] F. Sultan, T. Nguyen, and L. Iftode, "Limited-Size Logging for Fault-Tolerant Distributed Shared Memory with Independent Checkpointing," Technical Report DCS-TR-409, Dept. of Computer Science, Rutgers Univ., Feb. 2000.

[34] F. Sultan, T. Nguyen, and L. Iftode, "Scalable Fault-Tolerant Distributed Shared Memory," *Proc. SC 2000 High Performance Networking and Computing Conf.,* Nov. 2000.

[35] G. Suri, B. Janssens, and W.K. Fuchs, "Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory," *Proc. 25th Int'l Fault-Tolerant Computing Symp. (FTCS-25)* June 1995.

[36] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA-22),* June 1995.

[37] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," *Proc. Second USENIX Symp. Operating Systems Design and Implementation (OSDI-2),* Oct. 1996.

Florin Sultan received the BSE degree from the Polytechnic Institute of Bucharest, Romania, in 1989 and the MS degree in computer science from Old Dominion University, Norfolk, Virginia, in 1996. He is currently a PhD candidate in computer science at Rutgers University, Piscataway, New Jersey. His research interests include distributed systems, support for fault tolerance and high availability of network services, and operating systems.

Thu D. Nguyen received the BS degree from the University of California, Berkeley, the MS degree from the Massachusetts Institute of Technology (MIT), Cambridge, and the PhD degree from the University of Washington, Seattle. He is currently an assistant professor in the Department of Computer Science at Rutgers University, Piscataway, New Jersey. His research interests include distributed systems, particularly in the areas of security and peer-to-peer computing, highly available network servers, and operating systems. He is a member of the IEEE Computer Society.

Liviu Iftode received the BSE degree from the Polytechnic Institute of Bucharest, Romania, in 1984. He received the MA and the PhD degrees in computer science from Princeton University, Princeton, New Jersey, in 1993 and 1998, respectively. Since 1998, he has been an assistant professor in the Department of Computer Science at Rutgers University, Piscataway, New Jersey. His research interests include distributed systems, operating systems, mobile and embedded computing, and networking. He is the vicechair of the IEEE Technical Committee on Operating Systems and a member of the editorial board of *IEEE Distributed Systems Online*. He is a member of the IEEE Computer Society.