

Scalable Fault-Tolerant Distributed Shared Memory

Florin Sultan*, Thu Nguyen, Liviu Iftode

Department of Computer Science

Rutgers University, Piscataway, NJ 08854-8019

{sultan, tdnguyen, iftode}@cs.rutgers.edu

Abstract

This paper shows how a state-of-the-art software distributed shared-memory (DSM) protocol can be efficiently extended to tolerate single-node failures. In particular, we extend a home-based lazy release consistency (HLRC) DSM system with independent checkpointing and logging to volatile memory, targeting shared-memory computing on very large LAN-based clusters. In these environments, where global coordination may be expensive, independent checkpointing becomes critical to scalability. However, independent checkpointing is only practical if we can control the size of the log and checkpoints in the absence of global coordination. In this paper we describe the design of our fault-tolerant DSM system and present our solutions to the problems of checkpoint and log management. We also present experimental results showing that our fault tolerance support is light-weight, adding only low messaging, logging and checkpointing overheads, and that our management algorithms can be expected to effectively bound the size of the checkpoints and logs for real applications.

1 Introduction

In the last five years, researchers have shown that commodity clusters have the potential to provide super-computing capabilities at a fraction of the cost of traditional multiprocessor systems [1, 8]. At the same time, much research has been conducted in software distributed shared memory (DSM) to make it as easy to program clusters as it is to program shared memory multiprocessors [15, 2, 6, 20, 21, 29, 27, 26, 17]. These advances in programmability and performance are making it possible to use very large clusters as a cost-effective platform for data-intensive, long-running applications. As cluster size and application running times increase, adding fault tolerance becomes critical. At the same time, to preserve performance, the fault tolerance support itself must be both light-weight and scalable.

In this paper, we address the problem of designing a fault-tolerant DSM system, specifically targeting scalable DSM protocols. The novelty of our work lies in the combination of fault tolerance techniques with a scalable DSM protocol using user-level memory-mapped communication, such that the resulting system can be used in very large local-area clusters. Furthermore, because our fault-tolerant protocol does not require global coordination for fault tolerance related operations, its ideas can also be extended to wide-area meta-clusters (clusters of local-area clusters connected by the Internet), which researchers are starting to explore [28, 14].

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/parallel computations, there are two options for checkpointing [11]: *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, the set of the most recent checkpoints may not represent a globally consistent state of the system.

Coordinated checkpointing has typically been the approach of choice for recoverable DSM [7, 18, 22, 4, 9] because it is simpler to implement when non-failed nodes are assumed to be always accessible [12]. For very large clusters and meta-clusters, however, coordinated checkpointing is much less practical because of the increasing cost of global coordination and the likelihood of temporary disconnects in communication. Non-blocking consistent checkpointing protocols, which aim at reducing the coordination overhead, have been shown to be non-optimal [5], forcing processes to take unnecessary checkpoints. An additional advantage of independent checkpointing is that, due to checkpointing autonomy, it enables application-level optimizations like *memory exclusion* [24] to reduce checkpoint size.

The problem that we address then is how to efficiently combine independent checkpointing with a state-of-the-art DSM protocol in order to provide a fault-tolerant yet high-performing shared memory program-

*Supported by a USENIX student research grant.

ming environment on commodity clusters. Specifically, we show how a Home-based Lazy Release Consistency (HLRC) DSM protocol [16] can be extended with independent checkpointing in order to efficiently tolerate single-fault failures. Our fault tolerance algorithms are completely distributed, do not force processes to synchronize, and only make use of available protocol information.

The choice of the base DSM coherence protocol is important because (i) the protocol must scale well with cluster size, (ii) the protocol must have low memory overhead, freeing memory for fault tolerance related tasks (for example, logging), and (iii) the protocol 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 [35].

The challenge is to keep HLRC efficient while integrating fault tolerance. To keep overhead low, our recovery support (i) uses volatile memory for logging, and (ii) aggressively exploits HLRC’s semantics to minimize the amount of state that must be logged or checkpointed. Furthermore, to make independent checkpointing practical, we must control the size of the logs and checkpoints in stable storage: without global coordination, 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. To address this problem, we have developed two algorithms, Lazy Log Trimming (LLT) and Checkpoint Garbage Collection (CGC) to control the size of the logs and number of checkpoints [31]. The rules at the core of their operation describe the information necessary to recover the state of the computation in the DSM protocol.

In [31], we prove the correctness of LLT and GCC when used in HLRC (and thus address the issue of correctly extending a complex protocol such as HLRC). In this paper, we present an experimental evaluation of the overhead of fault-tolerant HLRC and the effectiveness of LLT and CGC for three applications from the SPLASH-2 benchmark suite [34], on a local-area cluster of PCs interconnected with a Myrinet LAN [3]. Our results show that the messaging overhead of our algorithms is negligible. The impact of fault tolerance support on overall performance is low (running times increase by under 7 %), except for one case where performance degrades by about 60 %, due to interference of the particular checkpointing policy used with the irregularity and the intense global synchronization in the application. The overhead of logging and checkpointing for recovery of the shared memory space is low - under 7 % of the base execution time for all applications studied. Our algorithms are efficient at trimming logs and discarding past checkpoints: in the experiments,

shared pages from no more than three checkpoints had to be retained at their home nodes and at least 60 % of the created logs were discarded.

The remainder of the paper is structured as follows. Section 2 presents related work in fault-tolerant DSM systems based on checkpointing and logging. Section 3 presents the HLRC protocol used in our system. Section 4 describes our fault-tolerant system and data structures it uses for fault tolerance, and presents the LLT and CGC rules. Experimental results and an evaluation of LLT and CGC are presented in Section 5. Section 6 concludes the paper.

2 Related Work

A comprehensive survey of recoverable DSM systems with various consistency models and implementation techniques is given in [23]. Coordinated checkpointing is used in systems like [22, 4, 9] 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. Systems using independent checkpointing and logging in volatile or stable storage are described in [25, 32, 9].

The idea of volatile logs of *accesses* to shared memory combined with independent checkpointing was proposed in [25] for a strictly-consistent, single-writer DSM. Whole pages are logged, and logs are flushed to stable storage on every outgoing page transfer which, combined with their large size, makes the scheme very expensive [32]. Replication of pages at multiple nodes wastes storage. Our system uses a more efficient, relaxed memory model, allows multiple writers, logs only changes made to a page, and does not replicate them across nodes.

In [9], Costa et al. have integrated log-based fault tolerance support into TreadMarks [21], a DSM system that also implements Lazy Release Consistency (LRC)[20]. 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 the independent checkpoints, as it may need to use information from the last globally consistent checkpoint. 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. Recovery from multiple failures is possible, at the expense of rolling back all processes to the last consistent checkpoint.

In [32], log-based recovery was first used in an LRC DSM. Their system uses independent checkpointing

and logging by a receiver at synchronization points and access misses. To reduce the overhead, a process keeps logs in memory and flushes them 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. The system can independently recover multiple failed nodes. Because the log is saved in stable storage and is only used for the recovery of the process that creates it, GC is easily done 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. 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.

3 The HLRC 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. In *release consistent* [13] software DSM, writes must be completed at synchronization time. The coherency unit is a memory page, and maintaining coherence is performed through page invalidations called *write notices*. Lazy Release Consistency (LRC) [20] is a variant of release consistency in which propagation of writes is delayed to the acquire time. Local writes are grouped into *intervals* delimited by synchronization operations. The local logical time of a process is defined as a counter of local intervals. A local vector of logical times called *vector timestamp* [20] keeps track of the (partial) ordering of synchronization events occurring at any process.

In Home-based Lazy Release Consistency protocol (HLRC) [16], every shared page has an assigned home process which maintains the most recent version of the page. A process that has an invalid page fetches it from its home at page fault time. The protocol supports multiple writers: updates to a page are detected and propagated from a writer to its home as a difference (*diff*) [21] between the modified page and a reference copy (*twin*) created before the first write following a page invalidation.

Figure 1 shows a typical process interaction in HLRC. Process P_j acquires lock L before writing to a shared variable x , allocated in memory page p . At release time, P_j sends its update of x to home $H(p)$, which applies it to its initial copy p_0 of page 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 , i.e., for which it received the corresponding *diffs*.

After releasing the lock, P_j receives a request for L

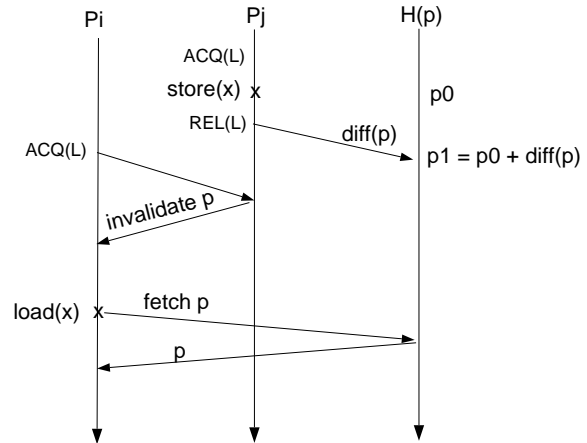


Figure 1: *The HLRC DSM protocol: process P_j writes to a page p homed by $H(p)$ and which is later accessed by process P_i .*

from P_i , which needs to access x . The lock granting message sent to P_i will include a write notice for p . Both processes send their vector timestamps with the lock request and grant messages, respectively. Following the invalidation of p , the read access to x at P_i will miss. To resolve the miss, P_i will send a request to $H(p)$ for the *minimal* version of p it needs as a result of the invalidation, which must incorporate the update to x . The home replies to this request with a copy of p that may also incorporate other writes, unrelated to P_i 's access.

4 Fault-tolerant HLRC

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 [30] 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 finished its recovery. We assume that there exists a mechanism for failure detection. We assume that the stable storage used by a node remains available after a failure, so that the process can be restarted on the same or on another node.

4.1 Design Overview

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.

There are two observations on the sender-based logging scheme we use: *(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 volatile log to stable storage at least on every checkpoint because after recovery from a crash it may need its log to support recovery of other nodes.

The changes in the DSM state at a process that must be replayed during recovery are generated by *synchronization operations* and *shared memory accesses*. To recover the shared memory space of a process in HLRC, we *checkpoint* shared pages only at home nodes and *log* those messages that induce changes in the DSM state. A process recovers from failure by log-based re-execution, starting from its last checkpoint (the restart checkpoint). A home process may have to retain *partial* information (shared pages) from its past checkpoints to cover recovery of other processes.

We next describe the logging algorithms along with their support data structures (logs) and show how they can be used along with checkpoints of homed pages for recovery from single-node failures. We also describe the rules (based on recovery correctness results proved in [31]) for log trimming and checkpoint garbage collection that allow correct recovery of any process.

4.2 Checkpointing and Logging for HLRC

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. In this section we describe the state to be kept in l_i and c_i for the recovery of the shared memory space.

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.

4.2.1 Support for Synchronization Replay

To support the replay of acquire operations, every process logs in volatile memory:

- write notices it generates, in a write notice log, wn_log ;
- replies it sends to lock acquire requests from a process P_i , in a per-process release log, $rel_log[i]$;
- lock acquire requests it sends to a process P_i , in a per-process acquire log, $acq_log[i]$.

Entries in the two logs generated by acquire operations record only the vector timestamp of the acquirer *after* the acquire completes. The release log kept by a lock owner is needed for re-execution of the acquires of other processes during recovery. The acquire log kept by an acquirer mirrors the lock owner's release log and is needed to recover it in case of a crash of the owner. This is because the owner's release log is created as a result of non-deterministic events (lock acquire requests received), so it cannot be regenerated by re-execution. Note that because the two logs are replicated on distinct nodes and can be restored from one another, they do not actually need to be saved to stable storage.

Logging synchronization events incurs low overhead because: *(i)* logging write notices is done as part of the base protocol, as nodes must retain their write notices, *(ii)* the logging overhead for lock recovery is low - a pair of vector timestamps is logged across the cluster per lock acquire, and *(iii)* the lock recovery logs do not need to be saved to stable storage.

The support for barrier synchronization replay is similar, logging a pair of logical times for every barrier.

4.2.2 Support for Replay of Shared Memory Accesses

Every process performs the following actions:

- the home of a page p includes a copy of p in every checkpoint;
- the home of a page p *retains* a sequence p_{ckp} of copies of p from past checkpoints;
- a writer of a page p creates and logs a diff for p in a per-page log $diff_log(p)$, along with the timestamp $diff.T$ of its creation.

Note that in the base protocol home nodes do not create diffs, while remote writers simply discard the diffs

after sending them to home nodes. The fault-tolerant protocol incurs therefore two components of logging overhead: (i) logging diffs at a non-home writer, and (ii) computing and logging diffs by a home for writes to its own pages. While the first component can be hidden in the diff computation in the base protocol, the second adds the overhead of creating twins on a first write page access after an invalidation, and of computing diffs and logging them at the end of an interval.

However, the advantage of this scheme is that it exploits HLRC features to avoid the more expensive logging of full-page transfers that would have to be done in a pure message-logging system. Also, despite the unavoidable costs, we keep the overhead to a minimum by not propagating or replicating diffs across nodes, as traditional LRC systems [21, 9] do during their failure-free operation. Diffs are stored only at their creator, and are moved to another node only in the less common case of its failure, since the recovery procedure of a process uses them to emulate operation of a home node.

For the replay of shared memory accesses, we exploit the HLRC protocol and design the recovery procedure such that we eliminate the expensive logging of page transfers, and avoid logging page requests, as a request for a page does not change the protocol state at its home.

4.3 Recovery

A process restarts by restoring the state of the processor from the last checkpoint and executes a local log-based recovery. Recovery of a process does not interfere with other operational processes: they can continue their execution unless they issue a request to the recovering process, in which case they block until recovery is finished.

To initialize the recovery of DSM state, a process P_i :

- obtains, for any non-homed page p that it has accessed before the crash, the *maximal* starting copy p_0 it needs from the home (for locally homed pages p_0 is in the restart checkpoint);
- collects $diff_log(p)$ from all writers of p if it has accessed p or if it is home for p ;
- collects wn_log 's from all processes from which it has applied page invalidations before crash;
- collects $rel_log[i]$ from processes from which it has acquired locks, thus restoring its corresponding acquire logs.

An initial handshake establishes exactly which entries of the logs are needed, based on the logical times of their creation. Note that because the checkpoints of

a recovering process and of the home are not coordinated, the starting copy p_0 for its replay of accesses to p may need to be one of home's older copies in its sequence p_{ckp} of checkpointed copies. The *maximal* starting copy [31] is the last copy in the p_{ckp} sequence that can be used for correct recovery of the process, i.e., a copy to which nothing can be added without compromising correctness of recovery.

The DSM recovery procedure of a process consists of: (i) re-execution of acquire operations using the write notice logs of all other processes and its acquire logs, recovered from processes from which it had acquired locks during normal execution, and (ii) replay of page misses by local emulation of a home, i.e., the recovering process maintains an evolving copy of p built from the starting copy p_0 obtained from p 's home, to which it dynamically applies partially ordered diffs obtained from all writers' $diff_log(p)$ logs.

As a result of replaying an acquire, the local vector timestamp of the recovering process (logged in some rel_log at a peer process) is advanced exactly as during normal operation. However, replay of page accesses does not need to result in the same copy of the page in memory as during the normal operation, after the miss is resolved. The page replay needs to only ensure that writes strictly related to the access (i.e., that happened before it) are applied to the replay page. Note that the recovering process must fully restore its homed pages, whether it accesses them or not.

Finally, to make recovery complete, the process also restores recovery supporting state lost in the crash. It dynamically rebuilds (by deterministic re-execution) the volatile tail of its write notice log and of its diff logs, and restores the lost release logs rel_log directly from the acq_log 's of other processes.

Note that because the replay of synchronization operations and page accesses is entirely local, it is expected that recovery will take less than the same sequence of the process' execution during normal operation.

4.4 Garbage Collection of Recovery Support State

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 become unnecessary for recovery of any process. 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 do *lazy log trimming* (LLT) and *checkpoint garbage collection* (CGC). The mechanism we propose allows a process to locally initiate and execute the trimming procedure without involving other processes.

The basic idea of our log trimming and CGC algorithms is to (partially) order checkpoints using vector timestamps: a process P_i timestamps a checkpoint with a vector T_{ckp}^i , equal to its vector timestamp T_i at the moment the checkpoint is taken.

Ideally, a perfect checkpoint timestamp would be a *global vector time* T_g , describing the global state reached by the system at the moment the checkpoint was taken. The closer T_{ckp}^i is to T_g , the better it reflects the global state, and thus allows a more efficient garbage collection of recovery state on other nodes. Intuitively, only changes in global state that may affect execution of P_i after T_g must be retained in the logs of other nodes.

In [31], we proved a set of rules for log trimming and garbage collection assuming instant availability of the global vector time for timestamping checkpoints. We present here the rules for the case where checkpoints are timestamped with the local vector timestamp. Their format and proofs are similar to those in [31], differing in only a few technical details.

4.4.1 Synchronization Log Trimming

The following rule gives necessary conditions on the write notice log entries that must be retained by a process.

Rule 1 (Wn log trimming) *A process P_i can support the replay of synchronization operations of any process by retaining only write notices in wn_log created in intervals starting from $\min_{j \neq i} T_{ckp}^j[i] + 1$.*

The next rule allows a process to retain log entries strictly needed to support acquire replay of another process and recovery of its release log. The logged vector timestamps are denoted as $acq.T$ and $rel.T$, respectively.

Rule 2 (ACQ log trimming) *A process P_i can support the acquire replay of a process P_j by retaining only entries of $rel_log[j]$ with $rel.T[j] > T_{ckp}^j[j]$, and it can restore the strictly needed portion of the $rel_log[i]$ of P_j by retaining only entries of its $acq_log[j]$ with $acq.T[i] > T_{ckp}^i[i]$.*

4.4.2 Lazy Log Trimming and Checkpoint Garbage Collection

The following rule identifies the oldest checkpoint from which a *maximal* starting p_0 must be retained by a home so that it can correctly cover recovery of any process. It also identifies the portion of the diff log a writer must keep for correct replay of accesses to p , if starting from p_0 .

Rule 3 (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 its checkpointed version satisfies: $p_0.v \leq T_{min} = \min_{j \neq H} T_{ckp}^j$.*

2. A writer of page p , P_i , can support recovery replay for p by retaining only $diff_log(p)$ entries with $diff.T[i] > p_0.v[i]$.

The main outcome of Rule 3 is that a home needs to only retain pages from a *window* of past checkpoints. This rule also shows that after a home $H(p)$ 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 .

Note that Rule 3 links the starting page p_0 with its corresponding diffs: intuitively, 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 .

4.4.3 An Example

Figure 2 shows an example of how checkpoints are garbage-collected and diff logs are trimmed in our system. 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 using page version timestamps according to Rule 3.

For ease of representation, 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. Also, assume that versions of checkpointed pages are equal to their respective checkpoint timestamp (in reality they may be more advanced).

Consider process P_3 , which is about to take checkpoint c_{34} at global time T_g . Then, with respect to P_3 as a home node for page p , T_{min} of Rule 3 at the moment c_{34} is taken will be exactly $T_{c_{12}}^1$. Rule 3 will force c_{32} , which has a copy of p such that $p_0.v = T_{c_{32}}^3 < T_{c_{12}}^1$, to be retained by P_3 . The solid arrow shows this dependency from c_{12} to c_{32} that sets the upper limit of P_3 's checkpoint window at the moment c_{34} is taken. The dotted bracket represents the checkpoint window of P_3 at the time it took its previous checkpoint c_{33} . The previous limit of the window had been c_{31} , as set by the now obsolete dependency from c_{22} at the time c_{33}

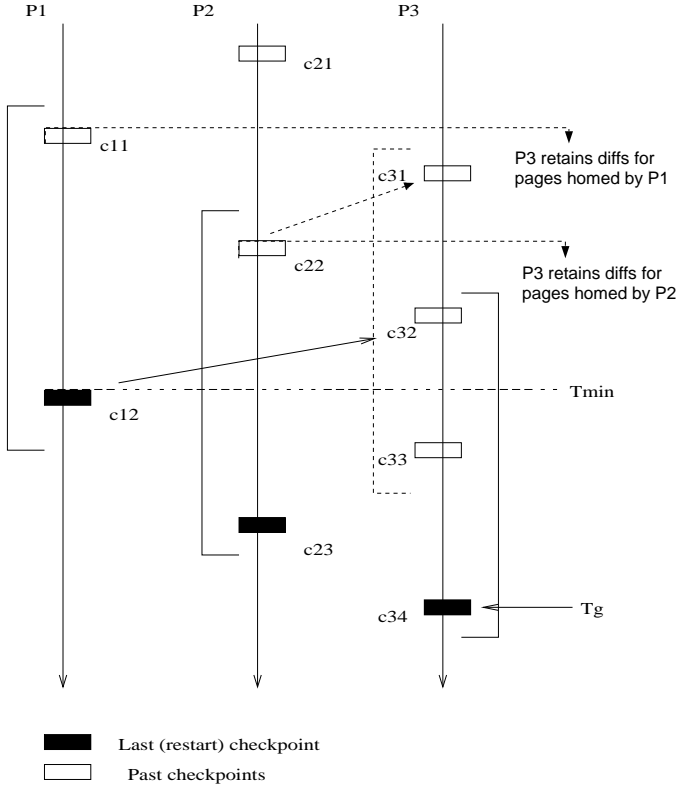


Figure 2: *Determining the checkpoint window for CGC and the diff log bounds for LLT by process P_3 at the time it takes c_{34} .*

was taken. In effect, the window advances by including the new c_{34} and dropping the unnecessary c_{31} .

As a writer of a page homed by another process, according to part 2 of Rule 3, P_3 must retain only diffs it has logged for pages that a recovering process would potentially need from c_{11} and c_{22} , respectively. As noted before, trimming of diff logs can be performed lazily, when P_3 learns the page version timestamps of page copies in c_{11} and c_{22} .

4.4.4 Using Approximate Information in Computing Trimming Bounds

Note that, in practice, each process P_i maintains, for any P_j , its last known value \hat{T}_{ckp}^j of P_j 's checkpoint timestamp T_{ckp}^j . Our algorithms are lazy in nature and can tolerate stale or inconsistent \hat{T}_{ckp}^j values across nodes at the expense of non-optimal efficiency, but preserving correctness of recovery.

Rules 1, 2, and 3 prove ideal bounds for how each process can dynamically trim all unnecessary entries from its logs. For this purpose, a process P_i needs from all other processes P_j : (i) the checkpoint timestamp T_{ckp}^j of the restart checkpoint for trimming its synchronization logs and for its CGC, and (ii) $p_0.v[\hat{i}]$ (if P_i is a writer of page p homed by P_j) for its LLT.

| 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 1: *Applications used and their characteristics.*

| Application and problem size | HLRC traffic (MB) | CGC traffic (MB) | % overh. |
|------------------------------|-------------------|------------------|----------|
| 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 2: *Message traffic overhead of CGC and LLT in a HLRC DSM system.*

One approach to distributing this information is to propagate it lazily, for example piggybacked on protocol messages: a process P_i would only have to send to another process P_j a vector T_{ckp}^i and one per-page integer, $p_0.v[j]$. Piggybacking, or other form of lazy propagation, may make this information stale at P_j . The corresponding values that P_j uses for trimming ($\hat{T}_{ckp}^i[j]$, $\hat{T}_{ckp}^i[i]$, and $\hat{p}_0.v[j]$) may become obsolete, depending on the sharing/communication pattern. The logs of write notices and the release logs will be trimmed less efficiently if $T_{ckp}^i[j]$ and $T_{ckp}^i[i]$ are stale (Rules 1, 2). For the page checkpoints, the bound \hat{T}_{min} may be less than T_{min} of Rule 3 because of stale $\hat{T}_{ckp}^i < T_{ckp}^i$, 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 $p_0.v[i]$.

The policy of propagating the T_{ckp}^i of the last checkpoint from a process P_i to other processes 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 results presented in Section 5 use piggybacking on protocol messages, and achieve good efficiency for LLT and CGC with low message overhead.

5 Experimental Results

To evaluate our design, we have extended the HLRC protocol to include: (i) LLT, (ii) a checkpointing policy to decide when to checkpoint at each node, and (iii) a simulation of CGC¹.

¹The prototype we use in our experiments implements logging, but does not yet implement full checkpointing and recovery. Nevertheless, it is complete enough to make measurements on a real protocol with real-world applications.

| <i>Application and problem size</i> | <i>Ckp. policy</i> | <i>Ckp. taken</i> | <i>Exe time with FT support (s)</i> | <i>% increase over base</i> | <i>Time logging (s)</i> | <i>Time disk write (s)</i> | <i>% overh. over base</i> |
|-------------------------------------|--------------------|-------------------|-------------------------------------|-----------------------------|-------------------------|----------------------------|---------------------------|
| Barnes 256 k | OF L = 1.0 | 6 - 10 | 2,677 | 61 | 16.2 | 96.8 | 6.8 |
| Water-Nsq. 19,683 | OF L = 0.1 | 9 | 1,644 | 0.6 | 0.9 | 5.3 | 0.4 |
| Water-Sp. 256 k | OF L = 0.1 | 5 | 2,737 | 7 | 13 | 79.4 | 3.6 |

Table 3: Performance of the independent checkpointing scheme with CGC and LLT in a HLRC DSM system.

| <i>Application and problem size</i> | W_{max} | <i>Max log disk (MB)</i> | <i>Total disk traffic (MB)</i> | <i>Logs created (MB)</i> | <i>Saved logs (MB)</i> | <i>% saved</i> | <i>Discarded logs (MB)</i> | <i>% disc.</i> |
|-------------------------------------|-----------|--------------------------|--------------------------------|--------------------------|------------------------|----------------|----------------------------|----------------|
| 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 |

Table 4: Overall efficiency of CGC and LLT in an HLRC DSM system. Disk traffic is generated by checkpointing homed pages and saving logs after in-memory trimming.

Our goals are to: (i) assess the efficiency 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 diff logs and the number of past checkpoints from which a home node must retain pages, and (ii) evaluate the impact of checkpointing and logging on the application performance.

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. (Recall that LLT and CGC mechanisms are totally decoupled from checkpointing operations and can be run at any point during execution.) We consider only the diff logs for trimming, as they consume the largest amount of space.

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 [3], using user-level virtual memory-mapped communication [10].

We have selected three applications from the SPLASH-2 benchmark suite [34], Barnes, Water-Nsquared, and Water-Spatial, to drive our prototype system. Barnes simulates the interaction of a system of bodies in three dimensions over a number of time-steps. Water-Nsquared simulates a system of water molecules in liquid state using an $O(n^2)$ brute force method with a cutoff radius. Water-Spatial solves the same problem in a 3-d setting.

We choose these applications in particular 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 present the worst-case scenario for LLT). 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. All results we present are for one run of each application. Data in the

tables is averaged over all nodes.

Table 1 shows the shared memory footprint of each application and the execution times with the base protocol (without fault tolerance support).

5.1 Checkpointing Policy

Our policy for deciding when a node should checkpoint is to enforce a limit L on the size of the volatile log. Since we examine applications with different memory requirements, we express this limit as a fraction L of the shared memory footprint. The checkpointing decision is independently made by each node, and it is transparent to the application: when the size of the log at a node exceeds L , that node takes a checkpoint.

The parameter L of the log-overflow policy is sensitive to the volume of updates. The second column in Table 3 shows the value of L we chose for each application, with the volatile log overflow policy (denoted by OF). In general, if L is too low it may increase checkpointing frequency and cause too much interference with the application. However, if L is too large it may result in too widely spaced checkpoints and cause a larger amount of logs to be retained (and saved to stable storage), for the *same* maximum checkpoint window size. Note that this latter effect is *not* related to a particular checkpointing policy or to our decision to trim logs only at checkpoints, nor is it inherent to LLT: the increased volume of retained logs simply reflects the larger amount of work that would be lost in a crash and thus needs more state to be retained for recovery.

Note that for Barnes $L = 1$, as opposed to 0.1 for the other applications. The reason for the larger value of L is that Barnes has the largest average volume of logs generated per byte of shared memory space. Increasing L avoids unrealistically frequent checkpoints, at the expense of a larger amount of memory used for logging. In the case of Barnes the shared footprint is small enough so that, even with a large value of L , the

log can still fit in the available physical memory.

5.2 Performance with Fault-Tolerant HLRC

Table 2 shows the message traffic overhead of the LLT/CGC control data piggybacked on protocol messages, compared to the base protocol traffic. The added messaging overhead is negligible.

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: (i) the parameter L of the log-overflow checkpointing policy (as a fraction of the shared memory size), (ii) number of checkpoints taken, (iii) execution time, (iv) percentage of its increase over the base execution time, (v) overhead of logging, (vi) overhead of writing to stable storage, and (vii) 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. Note that this is the direct overhead introduced by fault tolerance support for the DSM system. Other components of the checkpointing overhead, like checkpointing private data of a process, are unavoidable and therefore present in any fault-tolerant system. Furthermore, a particular implementation can add to the checkpointing overhead. For example, for transparent recovery of a system based on user-level communication, the communication state kept in user space and shared with the kernel can be saved in the checkpoint and restored upon restart. While user-level communication improves the overall communication performance of an application, its transparent recovery would also incur the overhead of saving this state in the checkpoint. An alternative, simpler implementation, could just bring the communication system to a predefined state after restart, at the expense of losing messages, a situation that must be dealt with while recovering the higher level DSM state. We do not explore these tradeoffs here, choosing to only assess the overhead of checkpointing the DSM space.

From Table 3 we see 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%. 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 added.

Figure 3 compares the averaged breakdown of execution time when running with base HLRC (left bar), and with fault-tolerant HLRC (right bar). The graphs are

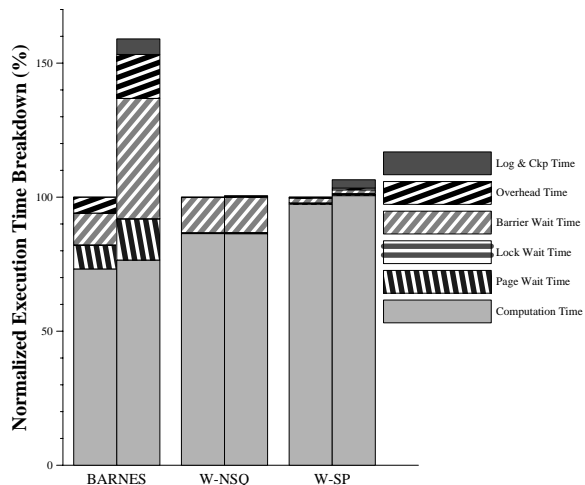


Figure 3: *The normalized execution time breakdown: the left bar represents the base protocol; the right bar represents the failure-free execution of HLRC with checkpointing and logging.*

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% to 28% of the execution time. The reason is the imbalance in distribution of homed pages and updates (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, the unbalanced load will spread checkpoints unevenly over time. Since Barnes uses many barriers, and checkpoints are unevenly distributed in time across processes, it is more likely that processes checkpoint in *different* intervals delimited by consecutive barriers. Therefore, the penalty taken by only one or a small set of processes checkpointing in some interval adds to the barrier waiting time of all other processes. For the other applications, which have less barriers and have regular access and update patterns, this effect is not visible.

5.3 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: (i) maximum size of the checkpoint window (W_{max}), (ii) maximum amount of stable storage consumed by diff logs, (iii) total amount of checkpointed pages and diff logs written to stable storage, (iv) size

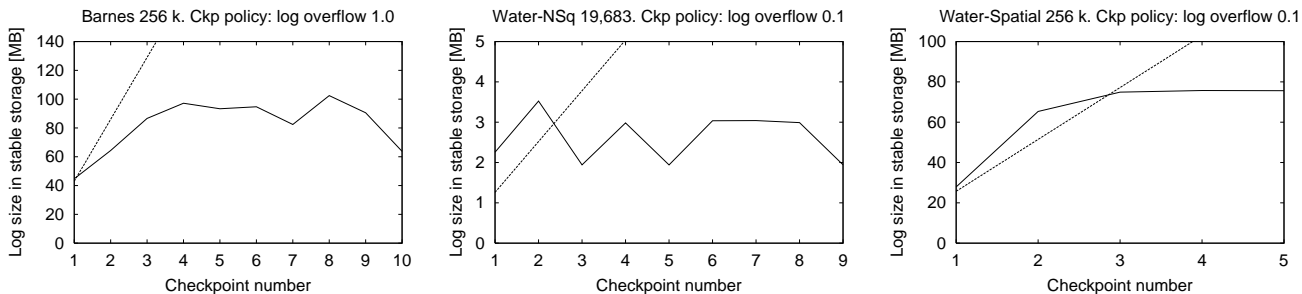


Figure 4: *Dynamics of log size in stable storage for three applications (logs are sampled at checkpoint time). Discrete points are connected to exhibit the trend under LLT control. Straight dotted lines show the unbounded growth the log would have in the absence of LLT.*

of volatile logs created during execution, (v) size of volatile logs saved in stable storage, (vi) percentage of saved logs from the volatile logs created, (vii) size of logs discarded as a result of LLT, and (viii) 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).

Figure 4 plots the variation of log size on stable storage against checkpoints, 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, with a slope of L bytes per checkpoint. The actual slope of any log increase can be less than L due to LLT, and could be higher due to our inexact evaluation of the overflow condition. 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, it can be seen that the actual log size suffers from a minor start-up effect: with the first checkpoint, more than L bytes of log might be saved. However, as it can be seen with Water-Nsquare and Water-Spatial, this effect is quickly canceled out by the effective trimming: log size increases by less than L bytes per checkpoint or even decreases, and within three checkpoints the total log size falls under the theoretical unbounded increase without LLT.

Table 4 shows that about 80 % of the created logs are discarded during the run by Barnes and Water-Nsquare, as a result of trimming. Water-Spatial ends

with over 40 % of the created logs retained, only because of the small number of iterations and the few checkpoints taken. If we double the number of iterations, the number of checkpoints also doubles and the percentage of discarded logs grows to 80 %.

Water-Spatial presents an interesting case, as it is very regular, and the log-size overflow policy forces checkpoints in every iteration. The regularity effect can be observed in Figure 4 in the gradual build-up of the saved log in the first two checkpoints, retaining the necessary state for recovery of the last iteration. After this point, every iteration adds and discards about the same amount to the saved logs. This is about 37 MB, well over the actual threshold imposed by the policy, (as can be seen from the slope steeper than L between the first two checkpoints), because of the imprecision mentioned above. 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 coordination.

5.4 Summary

While we do not make definitive claims based on results for only three applications, we believe that they 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 3 shows that the total expected time for saving checkpointed pages and volatile logs to disk is negligible compared to the base execution time.

- *LLT is effective at limiting log sizes.* Figure 4 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.
- *The maximum size of the checkpoint window is consistently small (three checkpoints), implying that CGC is efficient at controlling the total size of checkpointed information on stable storage.*
- *There exist negative side effects of independent checkpointing, exhibited by applications with intense global synchronization.* Depending on the local checkpointing policy used, local checkpointing overhead may add in the worst case to the execution time of all processes due to global synchronization in the application. A combination of checkpointing policies is perhaps better suited for applications with a large number of barriers, where the overhead of taking a coordinated checkpoint can be amortized as part of the barrier operations. Ultimately, the best choice in this case can be made by the programmer. The system can export an API to allow an application to specify places (for example at a barrier) where checkpoints can be taken by all processes, overriding any built-in policy. While checkpointing will not be entirely transparent, it will enable other useful application-level optimizations on the checkpoint size by memory exclusion. Alternatively, a two-level hierarchical scheme like that used in [33, 9] can interleave coordinated with independent checkpoints.

6 Conclusions

In this paper, we have presented the design of a fault-tolerant software DSM system based on independent checkpointing and logging. Independent checkpointing is particularly well suited to very large LAN-based clusters as well as meta-clusters connected by WANs. In order to make independent checkpointing practical, the system must efficiently control the size of the logs and checkpoints without recourse to global coordination. We have described the minimal state of the HLRC DSM protocol [16] that must be checkpointed and logged to support recovery from single-fault failures, and presented the rules developed in [31] for log trimming and checkpoint garbage collection.

We have implemented our proposed algorithms in an HLRC DSM system (not including recovery), and evaluated their performance using three update-intensive applications from the SPLASH-2 benchmark suite. The 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 pages from more than three checkpoints and logs on stable storage never grow beyond twice the applications' shared memory footprints. The message overhead of our scheme is negligible, and the logging and checkpointing overhead of the fault-tolerant protocol is less than 7 % of the execution times with the base protocol. The impact on overall performance is under 7 %, except for one case where performance degrades by about 60 %, due to interference of the checkpointing policy used with the irregularity and the intense global synchronization in the application.

7 Acknowledgements

We thank Murali Rangarajan for insights into protocol and application behavior, and for all the help with the experimental part. We also thank Ricardo Bianchini for comments on an early version of the paper. We are grateful to Xiang Yu from Princeton University, who was of great help to us in the implementation of the fault tolerance support in VMMC.

This project is sponsored by Rutgers Sciences Council Projects Program, USENIX, Microsoft, and NSF under CISE-9986046.

References

- [1] T.E. Anderson, D.E. Culler, D.A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, v. 15, no. 1, pp. 54-64, February 1995.
- [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] 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.
- [5] G. Cao, M. Singhal. On Coordinated Checkpointing in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, v. 9, no. 12, pp. 1,213-1,225, December 1998.
- [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] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, J. Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing (PP97)*, March, 1997.
- [9] 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.
- [10] C. Dubnicki, L. Iftode, E.W. Felten, K. Li. Software Support for Virtual Memory-Mapped Communication. *Proc. 10th International Parallel Processing Symposium*, April 1996.
- [11] 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.
- [12] E. N. Elnozahy, D. B. Johnson. The Performance of Consistent Checkpointing. *Proc. 11th Symposium Reliable Distributed Systems*, pp. 86-95, October 1992.
- [13] 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.
- [14] 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.
- [15] L. Iftode, J. P. Singh. Shared Virtual Memory: Progress and Challenges. *Proceedings of the IEEE*, v. 83, no. 3, March 1999.
- [16] L. Iftode. Home-Based Shared Virtual Memory. *Ph.D. Thesis*, Princeton University, June 1998.
- [17] 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.
- [18] G. Janakiraman, Y. Tamir. Coordinated Checkpointing Rollback Error Recovery for Distributed Shared Memory Multicomputers. *Proc. 13th Symposium on Reliable Distributed Systems*, October 1994.
- [19] D. B. Johnson, W. Zwaenepoel. Sender-based Message Logging. *Proc. 17th International Fault-Tolerant Computing Symposium (FTCS)*, pp. 14-19, June 1987.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] G. G. Richard III, M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. *Proc. 12th Symposium on Reliable Distributed Systems*, pp. 86-95, October 1993.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] 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.
- [30] R. Strom, S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, v. 3, no. 3, pp. 204-226, August 1985.
- [31] F. Sultan, T. Nguyen, L. Iftode. Limited-size Logging for Fault-Tolerant Distributed Shared Memory with Independent Checkpointing. *Dept. of Computer Science Technical Report DCS-TR-409*, Rutgers University, February 2000.
- [32] 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.
- [33] N. H. Vaidya. A Case for Two-Level Distributed Recovery Schemes. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 64-73, May 1995.
- [34] 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.

- [35] 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.