Using Peer Support to Reduce Fault-Tolerant Overhead in Distributed Shared Memories

Galen C. Hunt *
Michael L. Scott †

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

{gchunt,scott}@cs.rochester.edu

Abstract

We present a peer logging system for reducing performance overhead in fault-tolerant distributed shared memory systems. Our system provides fault-tolerant shared memory using individual checkpointing and rollback. Peer logging logs DSM modification messages to remote nodes instead of to local disks. We present results for implementations of our fault-tolerant technique using simulations of both TreadMarks, a software-only DSM, and Cashmere, a DSM using memory mapped hardware. We compare simulations with no fault tolerance to simulations with local disk logging and peer logging. We present results showing that fault-tolerant Treadmarks can be achieved with an average of 17% overhead for peer logging. We also present results showing that while almost any DSM protocol can be made fault tolerant, systems with localized DSM page meta-data have much lower overheads.

Keywords: Distributed shared memory, fault tolerance, peer logging.

*Galen Hunt was supported by a research fellowship from Microsoft Corporation.

†This work was supported in part by NSF Institutional Infrastructure grant no. CDA-8822724, and ONR research grant no.N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology – High Performance Computing, Software Science and Technology Program, ARPA Order no. 8930).

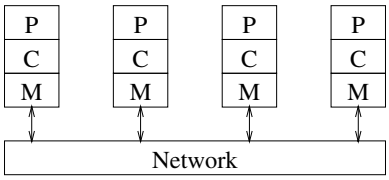

Figure 1: A network-based distributed parallel architecture includes processors (P), caches (C) and memory (M).

1 Introduction

While modern distributed shared memory (DSM) systems hide the details of explicit communication management, they cannot hide the increased risk of system failure. For any distributed system the mean time to failure (MTTF) is inversely proportional to the number of nodes in the system. Large DSM applications need fault tolerance to ensure their completion in volatile distributed environments.

A number of fault-tolerant techniques have been suggested for DSM systems. Experimental results, however, have been limited to systems using sequential consistency models. We present in this paper a self-reliant fault-tolerant protocol that is sufficiently general to be applied to almost any consistency model. To demonstrate its generality, we present experimental results for two DSMs: Cashmere and TreadMarks.

Traditional fault-tolerant systems have logged data needed for system recovery to local disks. The protocol presented in this paper can use either local disks or peer-based network logging. In a peer-based system, each node has a designated peer. Recovery information is replicated across the network to the peer. Our results show that peer logging reduces log latency and in some cases drastically reduces fault-tolerant overhead.

An important design decision in any DSM system is the placement of page meta-data. Depending on the DSM protocol, page meta-data includes information such as page location, which processes hold valid copies of pages, and when each process last saw a valid copy of a page. Meta-data placement and management becomes especially important when planning for recovery from failures. As will be shown later, systems that maintain localized meta-data provide much lower fault tolerance overhead.

The remainder of the paper is organized as follows: In Section 2, we give a brief overview of the architecture and operating models of modern DSM systems. Section 3 examines fault tolerance issues as they relate to DSM. Section 4 describes our fault-tolerant DSM protocol. Our simulations and experimental methodology are presented in Section 5. In Section 6 we examine the experimental data. We discuss related work in Section 7. Finally, in Section 8 we summarize and discuss future work.

2 Distributed Shared Memory

Distributed shared memory (DSM) systems extend the shared memory programming model to loosely coupled distributed processors. In a common distributed system each processor has its own local memory; see Figure 1. The processors communicate by passing messages through a network. The DSM system is responsible for providing consistency between local copies of shared memory. The DSM system converts shared memory modifications to network messages. DSM systems have been implemented both in hardware [15] and in software [2, 3, 10, 16].

Software DSM systems are particularly interesting because they can be used on either specialized networks or on commodity networks such as Ethernet [17] and ATM [14]. Using a software DSM, a network of workstations can be turned into a parallel processing environment. Programs designed for tightly-coupled,

bus-based architectures can be run with little modification. Programmer productivity is increased through the use of a familiar model: shared memory.

The DSM system is responsible for providing each node with a consistent view of shared memory. The DSM system maintains consistency using a model, which describes how the program interacts with shared memory, and a protocol, which describes how shared memory on one processor interacts with shared memory on another. Common consistency models/protocols include:

- **Sequential consistency** [13] requires that a read from any shared object will reflect the most recent write by any processor to that object. Object modifications must be propagated immediately to all sharing processors.
- **Release consistency** [6] relaxes consistency guarantees based on the insight that programs use synchronization objects to guard access to data objects. In release consistency, writes need to become visible at remote processors only when a synchronization release become visible. Release consistency allows optimizations such as collecting all writes and transferring them with the released synchronization object in a single message.
- **Lazy release consistency** [9] reduces network bandwidth by lazily copying only objects used by acquiring processors. When a processor acquires a synchronization object it sends the releasing processor a vector time stamp. The time stamp records the intervals when the acquiring processor last received data from each of the other processors. The releasing processor returns a message containing a list of exactly the data modified during the intervals that are in the releaser's logical past, but not in the logical past of the acquirer.
- **Entry consistency** [2] reduces the number and size of messages (compared to release consistency) by binding data objects to synchronization objects. When a processor acquires a synchronization object, it receives modifications only for the related data objects.

DSM systems can collect data object modifications using virtual memory hardware, instrumented store operations, or labeled access functions. In VM hardware based systems, sub-page modification granularity can be achieved by twinning and diffing or by ongoing write-through to the home node. Before modifying a page, a processor copies the page creating a **twin**. A **diff** is created by enumerating the differences between the modified page and twin made before the modifications.

DSM protocols can be classified as either update or invalidate. In an invalidate protocol, the acquiring processor receives invalidation notices for modified objects. The acquiring processor will request a modification only when it attempts to access an invalidated variable. In an update protocol, the acquiring processor receives the actual modifications. Invalidate protocols wins if the number of modifications needed by the acquiring processor is a small subset of the modifications made before a release.

3 Fault Tolerance

Error-free processes fail when the node on which they run fails. In a single uniprocessor, to recover from failure, the failed process must be restarted either on the same node, in the case of transient failure, or on another node. In a shared memory system, recovery must also include the DSM.

Richard and Singhal [19] describe a fault-tolerant checkpointing and logging method for sequential consistency DSM systems. They use a volatile in-memory log and a stable disk log. Before reading a shared page, the processor makes a copy of the page in the volatile log. When another process requests a page, the logged page is moved to stable storage before satisfying the remote request. The volatile log is cleared when

pages are moved to stable log. The stable log is reset after each checkpoint. Because a page can be invalidated by another process at any time, it must be logged every time it is read. To start recovery, the program loads the lastest checkpoint and begins replay. Whenever the recovering process needs a remote page, it retrieves the page from the stable log. By using logged pages, the process sees exactly the same data that it saw before the failure.

Suri et al [23] improve on the work of Richard and Singhal. For sequential consistency, they log the contents of the page only on the first read rather than logging a page each time it is read. When the page is invalidated, they log the number of times the page was read before being invalidated. On recovery the page is invalidated after it has been read the specified number of times. They also present a fault-tolerant lazy release consistency protocol. Lazy release consistency DSM systems invalidate pages only as result of an acquire on a synchronization object. Their LRC protocol logs the page the first time it is read and the list of invalidated pages at an acquire.

Neves et al [18] describe fault-tolerant protocol for entry consistency DSM systems using only a volatile log. Before releasing an object, a process copies it into the volatile log. If the acquiring process fails at some time in the future, it can recover by asking the releasing process to provide it with a new copy of the logged object. The log entries needed by a single process are distributed among the prior object owners. After a checkpoint, log entries are garbage collected by sending messages to all nodes in the system. A log entry can be discarded only when all processes that subsequently requested the object have been checkpointed. If a second node fails before all processes have checkpointed, recovery cannot be guaranteed.

An alternative to checkpointing is replication. Kermarrec et al [11] describe a fault-tolerant sequential consistency DSM system. Their system takes advantage of the page duplication inherent in DSM systems. Periodically a global checkpoint is created by making sure that duplicates exist for all shared pages. All pages are then marked read only. As write faults take place, a new writable page is created. Modifications and future readers use this page. When the next checkpoint is made, the written page is duplicated and the pages from the last checkpoint are discarded. In the case of a single node failure, the entire system must role back to the last checkpoint.

The protocol we present in the next section is similar to the **self reliant** work of Richard and Singhal, and of Suri *et al*. Each process of a DSM program is responsible for its own fault tolerance. The process must preserve any information necessary to guarantee that replay after a failure is externally indistinguishable from the original execution. The process is self reliant both before failure and after a failure. A process must make any preparation for failure without changing the state of other processes in the system and it must be able to recover from a failure without causing the rollback of any other processes.

Our protocol differs from earlier self-reliant protocols in that we use peer logging to reduce fault-tolerant overhead. For most DSM systems, peer logging reduces logging time and shortens the critical path of distributed applications. Our peer logging differs from the work of Neves et al and Kermarrec et al in that we support isolated node recovery.

4 Peer Logging

Log latency is the primary performance cost of fault tolerance. A disk log operation has finished only when the log entry and any disk meta-data have been safely flushed to disk. While disk logs are reliable, cheap, and non-volatile, they also have large latencies. Even the fastest disks have latencies on the order of a few million CPU cycles.

Disk latency becomes a crucial issue when it lengthens the critical path of program execution. For most DSM programs, the critical path follows access to a synchronization object or a data page. A process acquires the object, modifies some data and passes the object to another process. For reliable fault tolerance, the object

cannot be passed until the acquire has been logged. If log latency is large, compared to the time the object is held, the program's critical path will grow.

We reduce log latency by using a remote peer as a log provider. Data is logged by sending an asynchronous message to the log provider. Once the message has been sent, the requesting process can continue. The log provider starts an asynchronous disk operation to create a non-volatile copy of the log and then returns a acknowledgement to the requester. The requesting process considers the log operation finished when it receives the acknowledgement from the log provider. Spooling to disk is not required for correctness of the protocol because a valid backup of the data exists in the memory of the log provider. We spool log entries to disk to reduce memory usage on the logging peer. In the case of failure, the process is restored by reading log entries over the network from the peer.

The log provider's process state is not changed by receipt of a log request. Conceptually, logging and computation are two separate processes. Logging state need not be preserved because it is not externally visible. Only the log entries are externally visible, and then only to the process that made the log and only in the case of failure. The log provider can operate as either a secondary task on a peer that is running the same DSM program or as a log "server" that only manages log entries and does not participate in meaningful computation.

Our protocol is divided into a failure preparation phase and a failure recovery phase. Failure preparation is responsible for guaranteeing that in the case of a failure the failure recovery phase can restore the process to its pre-failed state. Failure preparation consists of the following:

1. The process makes periodic checkpoints. The checkpoints must contain sufficient information to completely recover the process state at the point of the checkpoint. The checkpoint should include all private data and local copies of shared memory pages.

2. Whenever the process receives information from another process, it logs the information asynchronously to reliable storage. Incoming messages include page invalidation and write notices, responses to requests for copies of remote pages caused by local page faults, and acquire and release operations on synchronization objects. In the case of failure, the logged messages will be used to ensure that recovery matches the path of the original execution.

3. Before sending any messages that reflect internal state, the process must wait for any relevant outstanding log operations to finish. This wait ensures that state becomes externally visible only after the data necessary to recreate it has been safely logged. For example, in the case of a remote request to satisfy a page fault, the processor must ensure that any pages read prior to modifying the specified page have been logged.

4. After a periodic checkpoint has been committed, the prior checkpoint and all intervening log entries are discarded. The frequency of checkpoints can be adjusted to meet application requirements based on the size of available log space, checkpoint cost and desired recovery time.

5. In the case that the log provider fails, the process must find another log provider and create a checkpoint before fulfilling any network requests. By creating a new checkpoint, the process removes any need for the missing log entries. A window of vulnerability does exist between the time the log provider fails and the point when the new checkpoint has been committed to a new log provider. This window of vulnerability can be reduced by duplicating log providers. It is assumed that in most cases, the time to locate a new log provider and create a new checkpoint is sufficiently small to provide a desired level of fault tolerance.

The second phase of our fault-tolerant protocol is failure recovery. After a failure has been detected, the process on the failed node must be recovered before execution of the DSM program can continue. The processor can be recovered either on the original or on another node. Our protocol does not require an explicit mechanism for suspending other processes in the DSM system. The other processes will continue forward execution until they reach a point at which they require information from the failed process. At that point they stall until the failed process has recovered. During recovery, requests from remote processes are queued for processing after recovery completes. Failure recovery consists of the following:

1. The last checkpoint for the failed process is restored. The process can be restored either on the same physical node on which it was originally running or on another node. The restored checkpoint contains all data available locally to the process when the checkpoint was taken.

2. After restoring the checkpoint, execution of the failed process rolls forward. As the replay continues, the process will need data that was originally supplied via network messages. At each message receive point, such as a request for a remote page or a synchronization object, the message is retrieved from the log provider. The logged message is identical to the original message so execution follows a path exactly the same as that of the process before failure.

3. Process recovery completes when all of the log entries have been replayed.

5 Experimental Methodology

Our experimental results were measured using execution driven simulations. Our simulator consists of two parts, a MINT [24, 25] front end and a memory system back end. MINT runs native parallel applications using the MIPS II instruction set. Memory references are passed to the memory system back end which determines which operations continue and which wait. This information is fed back to the MINT front end.

Table 2 details basic hardware parameters used in the simulation. To avoid the complexity of simulating operation system software the simulation uses constant numbers for disk scheduling, network message handling and VM interrupt handlers. We believe that our parameters accurately reflect optimistic, but achievable numbers. As with any system, your actual mileage may vary.

Our test application suite consists of four programs: `appbt`, `em3d`, `sor` and `water`. `Appbt` is from the NAS parallel benchmark suite [1]. `Em3d` is from the Split-C [4] project at UC Berkeley. `Sor` is a local computation kernel. `Water` is part of the SPLASH suite [21].

`Appbt` calculates approximate solutions to Navier-Stokes differential equations modeling heat dissipation. `Appbt` uses linear arrays to represent higher dimensional data structures. Each processor is assigned a separate square area. Communication between processors occurs only on boundaries between the areas. We simulate 5 time steps over 4096 grid points.

`Em3d` models the propagation of electro-magnetic waves through objects in three dimensions. A preprocessing step models the problem as computation on an irregular bipartite graph containing nodes representing electric and magnetic field values. Computation consists of step-wise integration of field forces alternating between electric and magnetic field nodes. A node's value is calculated based on the value of its neighbors. For our simulations, we simulate 65536 field nodes. Each node is connected to 5 neighbors with 5% of all neighbors located on a remote process. Each process is statically assigned a $1/n$ fraction of the nodes for the duration of the program. We calculate the field for 10 time steps.

`Sor` computes the steady state temperature of a metal sheet using a banded parallelization of red-black successive overrelaxation on a 1024x1024 grid. Sharing occurs between processing neighbors. We simulate 10 time steps.

Hardware Parameter	Simulated Value
CPU Speed	200 MHz
Cache Size	128K
Cache Line Size	64 bytes
Cache Miss Latency	20 cycles
Memory Bandwidth	381 MB/s
Interrupt Handler	400 cycles
Page Size	4096 bytes
TLB Size	128 entries
TLB Miss	100 cycles
Disk Scheduling	1000 cycles
Disk Seek	10ms
Disk Bandwidth	15MB/s
Mem. Chan. Latency	200 cycles
Mem. Chan. Packet Size	64 bits
Mem. Chan. Bandwidth	50MB/s
ATM Packet Start Latency	2000 cycles
ATM Packet Size	48 bytes
ATM Packet Overhead	5 bytes
ATM Bandwidth	155Mbps
Ethernet Protocol Latency	2000 cycles
Ethernet Packet Overhead	54 bytes
Ethernet Bandwidth	10Mbps

Figure 2: Simulated hardware parameters.

`Water` is an N-body dynamics simulation of water molecules in a cubic box. The simulation evaluates forces and potentials using Newtonian equations of motion. To avoid computing all the $n^2/2$ pairwise interaction among molecules, a spherical cutoff range is set at a radius of half the box length. `Water` uses static scheduling to increase data locality. During any time step, a single process will share data with at most half of the other processes. We simulate 256 molecules for 3 time steps.

Our simulations consist of two DSM protocols, TreadMarks and Cashmere. TreadMarks [10] represents the state of the art in software-only DSM systems. It uses a multiple-writer, lazy release consistency model to reduce network bandwidth and minimize the impact of network latency. Shared memory accesses are detected using the virtual memory protection mechanisms. Updates collected using twinning and diffing. Page invalidations are forwarded with synchronization objects and diffs are forwarded on read faults. Cashmere [12] takes advantage of memory mapped hardware to remove the need for diffs. Modifications to shared memory are written through the network to a master copy of each page. To ease management, a common protocol page directory is maintained in distributed, globally modifiable memory. Access to the global directory is protected with locks. Cashmere provides software coherence with performance close to hardware-only systems.

Our TreadMarks simulation logs synchronization object operations, page invalidation notices, and received diffs. The Cashmere simulation logs directory entries on all directory operations, write notices on acquires, incoming master pages on page faults and synchronization object operations.

For each simulation we collect data varying several parameters. For a base case, the coherence protocols have no fault tolerance support. We simulate each protocol using both disk and peer logging. Our disk logging simulations assume a log-based file system. Any single log entry can be committed to disk with a single disk

App.	Protocol	Processors		
		4	16	32
appbt	Cashmere	929	820	772
appbt	TreadMarks	18	80	189
em3d	Cashmere	3784	1813	1389
em3d	TreadMarks	14	15	18
sor	Cashmere	394	403	87
sor	TreadMarks	0.19	1.21	3.21
water	Cashmere	279	473	437
water	TreadMarks	40	316	653

Figure 3: Data (in megabytes) logged by application and consistency protocol.

seek. Subsequent log entries that reach the disk controller prior to the termination of the first operation need not pay the seek latency. Our peer logging simulations use the same network mechanisms as the coherence protocol.

In addition to log provider, we also vary the type of network from 10Mb/s Ethernet to 155Mb/s ATM and 50MB/s Memory Channel [7]. The Ethernet numbers assume a 51 byte UDP packet overhead. The ATM numbers use a standard packet containing 48 data bytes and 5 bytes of overhead. The Memory Channel hardware is memory mapped and provides write-through of 64-bit values. Remote reads are not supported. Our Cashmere simulation assumes a network reminiscent of the Memory Channel but supporting remote reads as well as writes. For ATM and Memory Channel, we model contention at sending and receiving nodes, but assume adequate switching resources to alleviate the need to consider switching contention. The Ethernet model simulates a standard bus architecture network where all nodes must compete for bandwidth.

We measure performance using 4, 16 and 32 processors. Again the desire is to simulate reasonable systems and not to measure performance in extreme conditions.

Our simulations do not include checkpointing overhead or any measurement of recovery costs. While checkpointing cost is an important consideration in overall system performance, it does not directly affect DSM costs. On the other hand, logging overhead directly increases DSM costs. The number of log entries for a particular execution remains constant when checkpointing costs are varied.

6 Results

Table 3 contains information about the amount of data logged by each simulation. Log size remains constant when the log provider is changed from disk to peer. Cashmere logs far more pages than TreadMarks. When logging incoming pages, TreadMarks writes only the modifications made to the page since the last time the process read the page. Cashmere writes the entire page since it has no way of knowing which parts of the page have been modified. Even when a page is found locally on a node, Cashmere must log the page if any other nodes have touched the page. The lack of diffs forces Cashmere to log pages even when the changes may not necessarily affect a process's computation. Additionally, Cashmere maintains a large amount of page meta-state in the centralized directory. Because the meta-state information affects local process state, Cashmere must log every access to the shared page directory.

The normalized execution times for TreadMarks and Cashmere on the Memory Channel with 32 processors are shown in Figure 4. At its best, fault-tolerant Cashmere is 3.59 times slower than non-fault-tolerant Cashmere. The effects of the large number of log entries can easily be seen in vast performance differences between Cashmere with disk logging and Cashmere with peer logging. Peer logging on the Memory Channel

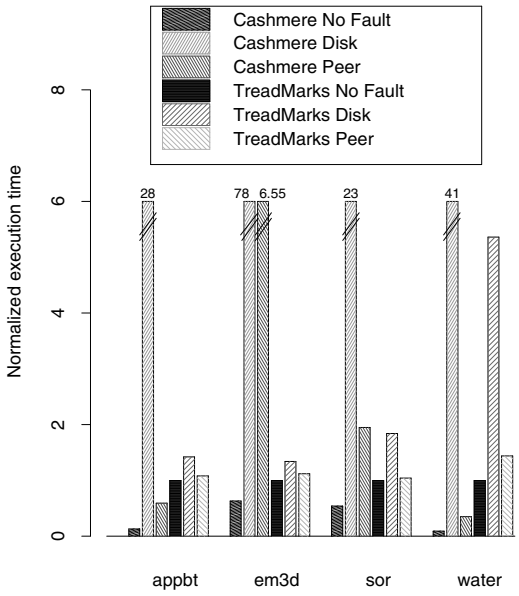

Figure 4: Execution times for Cashmere and TreadMarks on Memory Channel with 32 processors normalized to execution time on same system without fault tolerance.

has a much lower latency than disk logging. The burst bandwidth of the network is also much higher than disk bandwidth.

For most applications, fault tolerance on TreadMarks can be acheived with only minimal overhead. Figure 5 contains the normalized execution times for the test applications running on TreadMarks with 32 processors for each of the three networks. Execution times are normalized to the same network configuration without fault tolerance. For the ATM and Memory Channel networks, only one execution shows lower overhead for disk logging than peer logging. On TreadMarks, disk logging has on average a 67% overhead versus 17% for for peer logging. The important factor again is that logging to a peer has a lower latency than logging to a local disk. For Ethernet, disk logging always provides lower overhead. The reason is that at low bandwidths, peer logging must contend with coherence protocol traffic for bandwidth. While the local disk does have a higher latency, its bandwidth is larger than Ethernet.

Peer logging overhead as a percentage of overall execution time remains almost constant when varying the number of processors. Of twenty four possible network and protocol configurations, none showed more than a 20% change in normalized execution time. Only one showed more than an 11% change. Logging overhead is affected most by varying the number of pages logged and the latencies between page accesses and synchronization events. Page access latencies remain fairly constant regardless of the number of processors. With the exception of network bandwidth for Ethernet, logging resources scale with the number of processors.

While reducing overhead is important, speedup and absolute execution time tell the real story. Figure 6 contains speedup plots for `sor`. Shown are data for Cashmere on Memory Channel, TreadMarks on Memory Channel and TreadMarks on ATM. The Cashmere on Memory Channel numbers with superlinear speedup

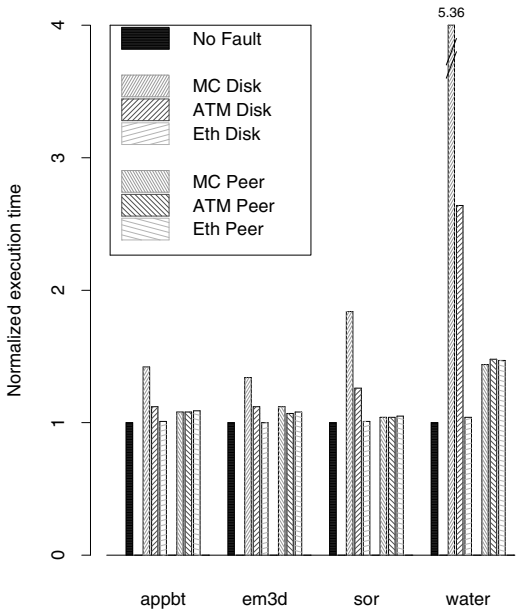

Figure 5: Execution times for TreadMarks on Memory Channel, ATM, and Ethernet with 32 processors normalized to execution time on same system without fault tolerance.

are a result of cache effects. The numbers for TreadMarks on Ethernet are not worth showing. The highest speedup obtained by TreadMarks on Ethernet is .86 with 32 processors and no fault tolerance. For the three networks in Figure 6, fault-tolerant peer logging shows better speedup than disk logging.

7 Related Work

Fault-tolerant DSM systems are related to work on transactional virtual memory systems. Transactional VM systems, including Camelot [22], PLinda [8], RVM [20] and LDSM [5], add transaction semantics to persistent virtual memory. A transaction consists of a start operation, a number of modifications to shared memory and either a commit or an abort operation. Transactions have three properties: first, atomicity: all updates made by the transaction either commit completely becoming globally visible or abort completely; second, isolation: changes made by the transaction are only visible after it commits; and third, automatic cleanup: uncommitted transactions in failed processes are aborted automatically.

Transactional memory systems tend to differ from DSM in the number and frequency of modifications. Transactional VMs generally assume that modifications to shared memory are rare and persistent, whereas DSMs assume that modifications are frequent and transient. In each case, system performance is optimized to meet modification assumptions. Our protocol does not require explicit use of transaction semantics. All DSM operations inherently commit. By using asynchronous disk I/O our DSM and the user applications are persistent as side effect of the fault-tolerance mechanism.

8 Conclusion and Future Work

We presented an efficient protocol for adding fault tolerance to distributed shared memories using peer logging. Our protocol uses independent checkpointing and isolated rollback for recovery from failures.

Our experimental results for two DSM protocols, Cashmere and TreadMarks, demonstrate that fault tolerance can be achieved with low impact on program execution times. In the case of TreadMarks, average overhead was 17%. A lazy release consistency DSM system using an update protocol might show even lower overhead.

We demonstrated the advantages of peer logging over local disk logging. In one extreme example for Cashmere, peer logging reduced program execution time by two orders of magnitude. In all cases, peer logging application have better speedup than disk logging applications. Our implementation of network logging is unique in that logging and recovery only involve a single process and its log provider. Peer logging is non-intrusive; it does not alter computation state on the log provider.

Our protocol is sufficiently general to be applied to almost any DSM system. Our work demonstrates that DSM systems with localized page meta-data have much lower fault-tolerant overhead ratios. We are exploring ways to relax Cashmere's global meta-data dependence to bring fault-tolerance overhead closer to that measured for TreadMark because Cashmere without fault tolerance is significantly faster than TreadMarks given a network with direct memory access.

References

[1] D. Bailey, E. Barszcz, L. Dagum, and H. Simon. NAS Parallel Benchmarking Results. In *Proceedings Supercomputing '92*, Minneapolis, MN, November 1992.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON '93*.

[3] J. B. Carter. Design of the Munin Distributed Shared Memory System. In *Journal of Parallel and Distributed Computing*, September 1995.

[4] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings Supercomputing '93*, pages 262–273, Portland, OR, November 1993.

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

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, Seattle, WA, May 1990.

[7] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2), February 1996.

[8] K. Jeong and D. Shasha. Persistent Linda 2: A Transactional/Checkpointing Approach to Fault Tolerant Linda. In *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems*, October 1994.

[9] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[10] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter '94 Technical Conference*, pages 115–131, San Francisco, CA, January 1994.

[11] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability. In *Proceedings of the Twenty-fifth International Symposium on Fault-Tolerant Computing*, pages 289–298, Los Alamitos, CA, June 1995. INRIA.

[12] L. I. Kontothanassis and M. L. Scott. High Performance Software Coherence for Current and Future Architectures. *Journal of Parallel and Distributed Computing*, 29(2):179–195, November 1995.

[13] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[14] J. Lane. ATM Knits Voice, Data on Any Net. *IEEE Spectrum*, 31(2):42–45, February 1994.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[16] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989. Originally presented at the *Fifth ACM Symposium on Principles of Distributed Computing*, August 1986.

[17] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–403, July 1976.

[18] N. Neves, M. Castro, and P. Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, Los Angeles, CA, August 1994.

[19] G. G. Richard III and M. Singhal. Using logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, Princeton, NJ, October 1993.

[20] M. Satyanarayanan, H. M. Mashburn, P. Kumar, D. C. Steele, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1):33–57, Carnegie Mellon University, February 1994.

[21] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, March 1992.

[22] A. Z. Spector, J. J. Bloch, D. S. Daniels, R. P. Draves, J. L. Eppinger, S. G. Menees, and D. S. Thompson. The Camelot Project. CMU-CS-86-166, Computer Science Department, Carnegie-Mellon University, November 1986.

[23] G. Suri, B. Janssens, and W. K. Fuchs. Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995.

[24] J. E. Veenstra. MINT Tutorial and User Manual. TR 452, Computer Science Department, University of Rochester, July 1993.

[25] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201–207, Durham, NC, January – February 1994.

A Execution Times

These tables contain the simulated execution times for `appbt`, `em3d`, `sor`, and `water`. The uniprocessor times include no DSM logging.

appbt		Exec. Time (secs.)		
System	**Logging**	**P=4**	**P=16**	**P=32**
Uniprocessor				5.2691
Cashmere MC	None	1.7491	0.6595	0.4921
Cashmere MC	Peer	12.8166	3.7409	2.2020
Cashmere MC	Disk	441.3210	168.2547	105.7270
Treadmarks MC	None	2.3469	2.1290	3.7278
Treadmarks MC	Peer	2.8243	2.4602	4.0235
Treadmarks MC	Disk	3.3022	3.6996	5.3022
TreadMarks ATM	None	6.4400	5.6766	10.4176
TreadMarks ATM	Peer	7.4303	6.5257	11.2237
TreadMarks ATM	Disk	7.3189	7.0735	11.6816
TreadMarks Eth	None	56.2128	67.9023	139.2775
TreadMarks Eth	Peer	69.6557	81.2708	151.6853
TreadMarks Eth	Disk	56.9925	69.8680	140.4964

em3d		Exec. Time (secs.)		
Uniprocessor				23.0623
System	**Logging**	**P=4**	**P=16**	**P=32**
Cashmere MC	None	3.4500	0.4832	0.2165
Cashmere MC	Peer	46.3837	5.6227	2.2558
Cashmere MC	Disk	463.3579	62.9992	26.9437
TreadMarks MC	None	3.9440	0.6467	0.3443
TreadMarks MC	Peer	4.2577	0.7270	0.3854
TreadMarks MC	Disk	4.1271	0.7027	0.4603
TreadMarks ATM	None	15.4463	2.2541	1.1230
TreadMarks ATM	Peer	16.1535	2.4203	1.1981
TreadMarks ATM	Disk	15.7011	2.3091	1.1528
TreadMark Eth	None	118.8242	19.0492	10.8414
TreadMark Eth	Peer	126.1312	20.8999	11.7438
TreadMark Eth	Disk	118.9804	19.1391	10.8716

sor		Exec. Time (secs.)		
Uniprocessor				2.2646
System	**Logging**	**P=4**	**P=16**	**P=32**
Cashmere MC	None	0.5763	0.1520	0.0486
Cashmere MC	Peer	5.1025	1.3113	0.1744
Cashmere MC	Disk	46.3621	12.2262	2.0423
TreadMarks MC	None	0.6199	0.1758	0.0895
TreadMarks MC	Peer	0.6229	0.1796	0.0930
TreadMarks MC	Disk	0.6826	0.2385	0.1645
TreadMarks ATM	None	2.4388	0.6633	0.2450
TreadMarks ATM	Peer	2.4452	0.6718	0.2540
TreadMarks ATM	Disk	2.4989	0.7232	0.3091
TreadMark Eth	None	20.9576	5.9028	2.6204
TreadMark Eth	Peer	21.0572	5.9931	2.7554
TreadMark Eth	Disk	20.9933	5.9418	2.6452

water		Exec. Time (secs.)		
Uniprocessor				0.2031
System	**Logging**	**P=4**	**P=16**	**P=32**
Cashmere MC	None	1.7061	0.5619	0.3292
Cashmere MC	Peer	5.7591	2.2856	1.2593
Cashmere MC	Disk	723.1721	321.4573	148.1354
TreadMarks MC	None	2.8372	2.8658	3.5782
TreadMarks MC	Peer	4.0776	4.3544	5.1441
TreadMarks MC	Disk	12.9234	20.2351	19.1909
TreadMarks ATM	None	5.8282	7.6960	9.7414
TreadMarks ATM	Peer	8.5775	12.0173	14.4167
TreadMarks ATM	Disk	19.0361	26.2900	25.6712
TreadMark Eth	None	56.0234	100.1866	135.7335
TreadMark Eth	Peer	85.2908	158.1562	199.9762
TreadMark Eth	Disk	84.3400	117.1506	141.3701

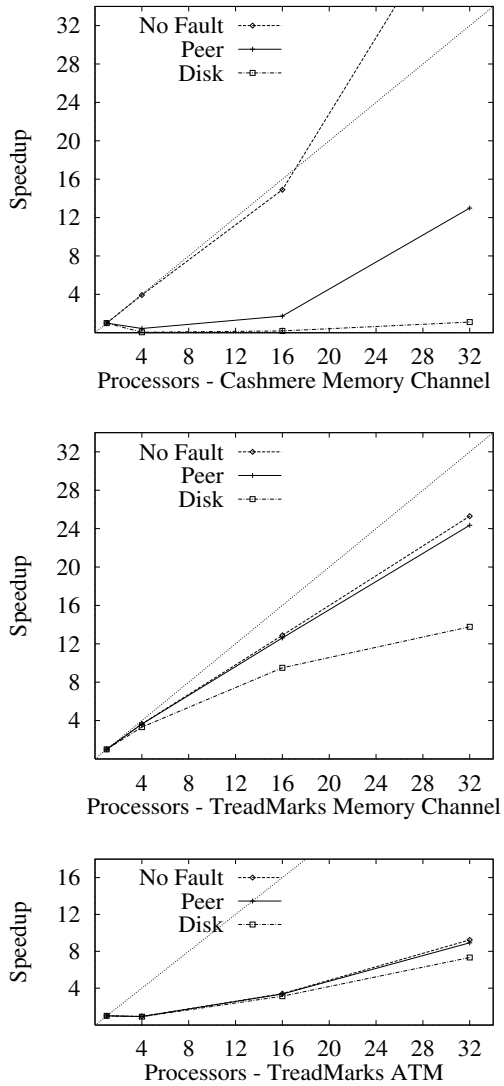

Figure 6: `Sor` speedups without fault tolerance, with peer logging and with disk logging.