Integrating Coherency and Recoverability in Distributed Systems

Michael J. Feeley, Jeffrey S. Chase, Vivek R. Narasayya, and Henry M. Levy

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
{feeley,chase,nara,levy}@cs.washington.edu

Abstract

We propose a technique for maintaining coherency of a *transactional* distributed shared memory, used by applications accessing a shared persistent store. Our goal is to improve support for fine-grained distributed data sharing in collaborative design applications, such as CAD systems and software development environments. In contrast, traditional research in distributed shared memory has focused on supporting parallel programs; in this paper, we show how distributed programs can benefit from this shared-memory abstraction as well.

Our approach, called *log-based coherency*, integrates coherency support with a standard mechanism for ensuring recoverability of persistent data. In our system, transaction logs are the basis of both recoverability and coherency. We have prototyped log-based coherency as a set of extensions to RVM [Satyanarayanan et al. 94], a runtime package supporting recoverable virtual memory. Our prototype adds coherency support to RVM in a simple way that does not require changes to existing RVM applications. We report on our prototype and its performance, and discuss its relationship to other DSM systems.

1 Introduction

Existing distributed shared memory (DSM) systems support *parallel programming* on distributed-memory multicomputers and workstation networks. Examples of such systems include IVY [Li & Hudak 89], Munin [Carter et al. 91], TreadMarks [Keleher et al. 94], and Midway [Zekauskas et al. 94]. These DSM systems maintain the illusion of a single shared memory by synchronizing data access and moving data between nodes when required, transparently to the application. DSM is useful in this context, because it simplifies programming of these distributed-parallel programs.

Parallel programs are not the only applications that can benefit from the concept of distributed shared memory; DSM can be applied to other application domains as well. Our goal is to support coherent virtual memory for programs that perform *transactional* updates to their shared memory space. While this style of programming is not ordinarily seen in parallel programs, it is standard for applications using a *persistent store*, a system that supports storage and retrieval of virtual memory data structures in disk files.

Our work explores the interactions between coherency and transactions. We describe *log-based coherency*, a simple technique for maintaining consistency of a transactional distributed virtual memory. The key idea behind log-based coherency is that log records used to support atomic transactions are also used as the basic mechanism for updating cached copies of the data on peer nodes. This unification of mechanisms allows us to add DSM support to systems that support persistence, without modifying existing persistent programs and without adding significant software overhead.

1.1 Persistent Stores

Persistent storage systems have evolved to meet the data management needs of design applications, including electronic and mechanical CAD systems and software development environments. These application environments consist of collections of programs that operate on persistent data structures representing design artifacts and derived information. Client programs navigate through the stored data by following pointers directly in virtual memory, with the sys-

This work is supported in part by the National Science Foundation (Grants No. CDA-9123308 and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer. Both Feeley and Chase have been supported by Intel Foundation Graduate Fellowships. Chase's present address is: Department of Computer Science, Duke University, Durham, NC 27706 (chase@cs.duke.edu).

tem moving data between memory and the persistent store as needed. A number of persistent stores have been built; some are research systems [Cockshott et al. 84, Moss 90, Carey et al. 94] and others are commercial object-oriented database (OODB) products (e.g., [Butterworth et al. 92, Lamb et al. 91, O. Deux 92]), augmented with database features such as query processors, schema languages, and indexing facilities.

To prevent failures from corrupting persistent structures, updates to a persistent store are grouped together and applied "all or nothing" at *commit points* designated by the program during its execution. For our purposes, a *transaction* is a period of execution ending in a commit point. The key property of transactions is that each commit point atomically enters a set of updates that together transform the store from one durable consistent state to another. Failures may cause some uncommitted updates to be lost, but the last committed state can always be recovered. Database systems typically combine these requirements of atomicity and durability with additional assumptions about how concurrent transactions are synchronized, but this is not essential to our notion of a transaction.

Our work explores techniques for extending a persistent store to allow network access. For example, a persistent store that supports design applications can be extended to allow a group of collaborating designers to run CAD tools at their workstations, accessing the shared store through the network. In the simplest configuration, updates are written atomically to a centralized server that maintains the authoritative copy of the data in the store. Clients fetch data from the server in bulk, cache it locally, and operate directly on the cached image in virtual memory. In the database community this architecture is sometimes called a *client/server database*. We refer to it as a *cached persistent store* since the ideas generalize to persistent virtual memory systems that do not provide full database features. Note that cached persistent stores are distinct from transactional systems for reliable distributed programming (e.g., Argus [Liskov 88] and Camelot [Eppinger et al. 91]) in that the database itself is not distributed; each transaction updates a cached *image* of a centralized database in virtual memory on a single node. In particular, there is no need for two-phase commit, since each transaction commits or aborts within a single process.

1.2 Combining Coherency and Recoverability

A key problem for cached persistent stores is maintaining consistency of the data cached in memory on each client. While these systems are similar to distributed file systems, the caches should be viewed as a distributed shared memory, since each client process accesses the cache *directly* in its virtual address space. There are other important similarities with DSM systems. Fine-grained sharing is common in collaborative design environments, where changes made by one engineer must be quickly integrated into structures that are readable by the others. The caches of different clients will overlap considerably when clients access the same artifacts, more so as physical memories grow and data remains in client memory for longer periods. For these reasons, DSM techniques such as fine-grained client-client transfers are appropriate for maintaining coherency for cached persistent stores. In the past, distributed persistent systems have supported coherency in a manner similar to most distributed file systems: by reading and writing shared data through the server, usually in fixed-size blocks, and invalidating cached blocks when another client acquires the file token or lock.

In addition, there is commonality between the key implementation aspects for coherency and recoverability. To implement either property, the system must capture the updates made by the application, and propagate those updates — either to durable memory (e.g., disk) or to other memories in the network — in such a way that all clients see a consistent view of the data at all times. The key performance factors are the same: (1) the cost of capturing updates, and (2) the precision with which those updates are captured, which determines the amount of traffic to the disk or network. This synergy can be exploited when both properties are supported together, and optimizations that improve the performance of one may also improve the performance of the other.

This paper develops a combined approach that integrates coherency support with a mechanism for ensuring recoverability of persistent data. It differs from other DSM systems in that it accommodates a transactional programming model and exploits the capturing of updates for a transaction log. Section 2 outlines our approach and its variations, discusses some of the design issues, and sets our work in context with other systems that support distributed database access and distributed shared memory. Section 3 describes our prototype implementation, and Section 4 presents some performance results. In Section 5 we discuss additional related work, and we conclude in Section 6.

2 Log-Based Coherency for a Cached Persistent Store

Log-based coherency is an extension to *write-ahead redo logging*, a common mechanism for implementing atomic and durable transactions. This mechanism works by recording new values of data items modified by a transaction, and writing them to a log on durable memory when the transaction commits. The system ensures that commits are atomic by writing the log *before* writing the updated objects back to the permanent database file. In the event of failure, a recovery procedure restores the database to a consistent state by replaying the committed log records into the permanent database file. Many recoverable systems use write-ahead redo logging; for example, it is fundamental to the *pin/update/log* commit protocol used by Camelot [Eppinger et al. 91].

When multiple clients are accessing the store through a network, the log records are written to a logically centralized storage service that also holds the permanent database file. A client failure aborts all uncommitted transactions executing in that client. A server failure may abort uncommitted transactions in all clients and initiate the recovery procedure to bring the permanent database file to a consistent state, reflecting the committed updates made by all clients. (Note that the storage service could be transparently replicated to reduce the probability of a server failure.)

The key to log-based coherency is that the redo log generated by each client holds exactly the information needed to maintain consistency of distributed memory. The system need only transmit the committed log tails to peer nodes that are sharing the modified objects; the recipients apply the log records they receive to update their cached data images. There is no extra runtime cost for collecting the information needed to maintain coherency, since it is already collected to support recoverability. Thus, a common implementation technique for a recoverable store is extended to support coherency in a straightforward way. In fact, our approach can be viewed as transaction logging to remote memory instead of (or in addition to) the disk.

As an example, suppose that several nodes are sharing a persistent object, X; each has a copy of X cached in its primary memory. When one of the nodes executes a transaction modifying X, the updates are performed and logged locally; other nodes are neither updated nor invalidated at that time. When the transaction is complete, the generated log record contains exactly the information needed to transition to the current state of X from its initial state. Thus, to bring the other nodes up to date, it is sufficient to propagate this portion of the log to the peer nodes.

Within this broad framework, systems that use log-based coherency could vary in a number of details, including:

- when log records are transmitted to peer nodes,
- which peers receive the log records,
- how updates are applied to a recipient's cache,
- how updates are synchronized with client threads executing locally, and, in a recipient's cache.

In our prototype, described in Section 3, we have made choices for ease of implementation and for the workloads we expect. However, our approach is flexible enough to accommodate variations. The following subsections discuss underlying concepts, the design choices we made, and some alternative possibilities.

2.1 The Role of Synchronization

In DSM systems, the method and timing of update propagation is closely tied to synchronization events. The first DSM systems (Monads [Rosenberg & Abramson 85] and IVY [Li & Hudak 89]) used virtual page protections and reference traps to capture updates and synchronize access to shared pages. In these systems, page-grain locking and page-grain coherency led to performance problems caused by false sharing. Newer high-performance DSM systems based on *release consistency* [Gharachorloo et al. 90] (e.g., Munin), *lazy release consistency* [Keleher et al. 92] (TreadMarks) and *entry consistency* (Midway) have reduced this problem by supplying synchronization primitives that function independently of the coherency protocol, and drive the propagation of updates. These systems reduce false sharing by allowing concurrent accesses to a shared page. However, they assume a fully synchronized (*properly labeled*) application program; that is, the application must acquire and release the correct locks at the correct times. For all of these systems, the coherency algorithm will fail for programs that make synchronization errors.

Log-based coherency uses a similar approach resting on similar assumptions. Our prototype supplies standard mutex primitives that are acquired as a transaction executes and released at transaction commit (we currently assume that transactions use strict two-phase locking). The system propagates updates only after a writer has released all relevant locks at commit, and it ensures that all relevant updates are applied locally before a reader is permitted to acquire a lock. Our system differs from release consistency, and is similar to entry consistency (Midway), in that coherency operations initiated by a lock operation are restricted to the data under the scope of that lock. The store is partitioned into segments, each under the control of a separate lock. Segments can be large or small, presenting an obvious tradeoff between synchronization overhead and false lock conflicts. We expect locks to be relatively coarse-grained; most commercial persistent stores support coarse-grained locking as natural for collaborative design environments. Regardless of the locking grain, the updates made while a lock is held often involve only a small number of the bytes controlled by that lock. In fact, we expect that most updates will be fine-grained (e.g., "change this **and** to an **or**"), but that read operations will consume large amounts of data for input to functions such as design analysis or graphical display. For this reason, log-based coherency separates the coherency grain from the synchronization grain: the updates sent to peer nodes are determined not by the locks acquired, but by the values logged at transaction commit.

Transaction log records include information about the locks acquired during the transaction. Each lock has a unique lock number and a sequence number (timestamp) that is incremented on each **acquire**. When a transaction acquires a lock, its log record is tagged with the sequence number and lock number of that lock. The synchronization primitives contain embedded calls to logging routines to generate these tags. The tags are used to determine which nodes must receive the log records and when they must be applied. For example, updates for a segment need only be sent to nodes that have previously acquired the lock for the segment. The sequence numbers are also used to ensure that all relevant data has been rendered coherent before an **acquire** can succeed, and to preserve the global ordering of updates from multiple nodes during recovery, as described in Section 3.4.

We believe that alternative synchronization models can be implemented with a modest effort, by replacing the synchronization primitives and the calls they make to underlying logging and coherency routines. Several relaxed models have been proposed by researchers arguing that strict serializability is inappropriate for design transactions (e.g., [Hornick & Zdonik 87]). We are exploring a read/write model that permits readers to operate on a previous consistent version of the data while an update is in progress elsewhere; readers use an **accept** primitive to explicitly signal their willingness to move forward to a newer consistent version. In this scheme, pending log records must be buffered in the recipient until they can be applied. We have used a similar *version-based consistency* model in the past for a range of parallel applications [Feeley & Levy 92].

2.2 Propagating Log Records

Our prototype uses a simple eager policy for propagating updates. At each commit point, after the log records have been written to the storage server, the in-memory copies of the log records for each segment are eagerly propagated to all clients that have recently acquired the locks for the modified segments. We use eager updates because they are simple (i.e., no buffering of log records), they are tolerant of client failures, and they reduce the latency of data access on a client. However, eager updates may increase network traffic and cannot scale to large numbers of clients, particularly on networks with point-to-point links that do not support broadcast or multicast.

We believe that alternative policies could be implemented transparently to the application by replacing the synchronization primitives and their embedded coherency code. For example, the system could propagate segment updates lazily, using an embedded call in the **acquire** primitive to fetch and apply pending log records. Segment updates could be fetched from the server, where all log records are cached in memory for a time, or passed with the lock by the last writer. In Midway, for example, the **acquire** primitive retrieves current versions of any stale objects in the requester's cache from the current lock holder. The acquire request includes a timestamp for the last time the lock was held by the requester; the lock holder determines which modified objects to return by comparing the timestamp against a modification timestamp for each object under the lock's scope.

For log-based coherency, lazy update propagation raises the question of how to determine when pending log records are no longer needed by peer nodes and can be discarded. One solution is to pass information about how many log records to hold for each segment along with the lock token, as each node acquires the lock in turn. Each node holds all log records up to and including the oldest records needed by the most out-of-date peer. Acquiring a lock brings new records, as well as an opportunity to discard records held locally. Note that discarding pending update records is not a concern with Midway's lazy propagation scheme, because no log records or captured updates are held; nodes do not save intermediate versions of objects that are modified repeatedly. In our case, these log records must be held in order to support the nonserializable read/write model described above. That is, a reader may acquire a previous version even if uncommitted writes are present in the writer's cache; the reader's cache must be updated to reflect the previous committed version.

2.3 Summary

In this section we presented an overview of log-based coherency. Built-in synchronization primitives are fundamental to our approach. Alternative coherency protocols — rules for propagating, applying, and releasing log records — can be realized by embedding calls to logging routines in the synchronization primitives. The locking routines can collect and maintain information about which peers must receive a given set of updates, and when.

3 Prototype Implementation

To experiment with our approach, we prototyped a simple implementation of log-based coherency by adding distribution support to CMU's Recoverable Virtual Memory (RVM) package [Satyanarayanan et al. 94]. RVM is a logging facility that supports transactional update and recovery of virtual-memory resident data structures. RVM is designed to be a lightweight and portable package for use with small databases that easily fit in physical memory: an RVM client copies the entire database into virtual memory when it starts up. This avoids the need to pin modified pages, but for large databases it causes double paging and unnecessary pageouts of clean pages by the virtual memory system. This limits RVM's usefulness for the collaborative design environments that interest us; however, it is an expedient vehicle for experimenting with log-based coherency.

In keeping with its minimalist philosophy, RVM does not support or rely upon any particular synchronization scheme. Though updates are transactional, multithreaded updates may or may not be serializable. In a similar spirit, our RVM-based prototype separates the synchronization aspects of coherency and recoverability from the mechanisms for collecting, propagating, and applying redo log records. Our intent is to accommodate various policies for propagating updates as part of a synchronization scheme "plugged in" to RVM.

We use RVM as a client/server distributed database by placing the transaction log files and the database file on a central NFS server. Clients maintain caches of the central database in their local virtual memories by reading the entire database into memory at startup (as in centralized RVM), in this case using the NFS protocol. As write transactions execute, RVM produces redo log records that are written to the NFS server. Log-based coherency is implemented by sending the log tails to other nodes that have the region mapped, where they are applied to each recipient's cache. Each node recording such a transaction produces a separate log. We added an RVM utility that merges these into a single log for recovery (see Section 3.4).

Our application interface is summarized in Table 1. The left-hand column describes the procedures the application uses to initialize, begin, and commit a transaction; to acquire a segment lock; and to describe the data that is modified by the transaction. The right-hand column shows the RVM calls made by each of these procedures. We added a new procedure, called `rvm_setlockid_transaction`, to the standard RVM interface. This procedure is called by the **acquire** primitive, as described in Section 3.3.

3.1 Capturing Updates

Both DSM and recoverable systems must determine which bytes are modified by the application using either VM write faults, a write barrier inserted by the compiler, or explicit calls from the application. Like the RVM system, our prototype assumes explicit runtime calls to a `set_range` procedure in the runtime package. A call to this function indicates an intent to update a particular range of bytes. We expect that this range corresponds to an object, and that the call is made by code generated explicitly by the language compiler (such as the ML compiler that has been used with RVM [O'Toole et al. 93]). In contrast, most DSM systems use virtual page access faults to captue updates, though software-based write detection is also used in Midway [Zekauskas et al. 94]. This issue is discussed in more detail in Section 4.

RVM coalesces modified ranges that are adjacent or overlapping in order to avoid writing redundant bytes to the disk log. Overlapping ranges, however, are unlikely when calls to `set_range` are generated by a compiler, as is anticipated. To improve common-case performance, we modified `set_range` to coalesce only when there is an exact match with a previously added range. Thus, objects that are modified multiple times during a transaction are still coalesced but with a simpler and more efficient implementation; this reduces `set_range` overhead by a factor of five. The remaining per-update overhead is dominated by searching the binary tree that stores modified ranges (in order by their address). As a second optimization, we avoid this search in the special case where a sequence of `set_range` calls is ordered by address.

3.2 Propagating Log Records

In response to `set_range` calls, the RVM runtime library builds a data structure that describes the modifications made by a transaction. When the transaction commits, this data structure is used to build I/O vectors for the Unix `writev` system call. The `writev` causes the new values of all modified objects to be copied from virtual memory to a system buffer for writing to disk. RVM thus avoids building an object log in virtual memory that contains copies of the modified objects.

We modified the RVM commit procedure to broadcast the same new-value information that is written to disk. Coherency data is broadcast using TCP/IP by issuing a `writev` system call for each node that has the current region mapped. We use the OSF/1 PThreads facility [Mueller 93] to create *receiver* threads for each communication channel that connects a node to its peers. These threads block in `readv` system calls waiting for coherency messages and applying the updates

Log-Based Coherency Operation	RVM Routine Called
Trans.Init()	tid=rvm_malloc_tid()
Trans.Begin(rvm_mode)	rvm_begin_transaction(tid, rvm_mode)
Trans.Commit(rvm_mode)	rvm_end_transaction(tid, rvm_mode)
Trans.Acquire(lock)	rvm_setlockid_transaction(tid, lock.lockId, lock.sequenceNum)
Trans.SetRange(addr, size)	rvm_set_range(addr, size)

Table 1: *Log-based coherency interface.*

when they arrive.

The format of the coherency records data differs from the data sent to disk in two respects. First, some records that are needed only for recovery and log trimming are not included in the broadcast data; i.e., only new-value range records are needed for coherency. Second, the header information for each range record is compressed from 104 bytes to between 4 and 24 bytes. The standard RVM header contains fields that are not needed for coherency; our header contains only the range's type, address, and size. The header is further compressed when ranges are small (less than 4 Kbytes) or close together (fewer than 256 Kbytes apart) by using smaller fields and by replacing the range's address with its offset from the preceding range; our modified `set_range` orders modified ranges by their address.

3.3 Synchronization

We added distributed locks that provide mutually-exclusive access to non-overlapping portions of an RVM region. Locks are acquired inside of transactions in a two-phase manner; all locks are automatically released when a transaction commits. Applications must acquire a lock before reading or writing the data protected by that lock.

The lock implementation is token based with a centralized lock manager and a distributed waiter queue, an approach used in many distributed systems including TreadMarks. At all times, exactly one node owns the lock token. The lock can be acquired on that node without remote communication; nodes hold onto the token until they are requested to pass it to another node. Acquire operations at other nodes send a message to the lock manager to request the lock. The node number of the lock manager is determined from the lock identifier number. The manager maintains a distributed queue of nodes waiting to acquire the lock. It adds the requester to the tail of the queue and forwards its request to the previous tail which responds either by sending the lock token to the requester, if the lock is available, or by queueing the request until the lock is released.

Each lock has a unique identifier and a sequence number that is incremented on each **acquire**. The current sequence number is passed along with the lock token when ownership of the lock changes. The **acquire** primitive calls RVM to associate the lock with the current transaction. We added a new procedure to the RVM interface, `rvm_setlockid_transaction(tid, lockId, sequenceNum)`, for this purpose. Because locks follow a strict two-phase protocol, each lock is acquired at most once during a transaction.

3.4 Preserving Ordering

While synchronization is mostly independent of coherency and recovery, there is an important interaction. When a transaction commits, the lock information provided to RVM is used to generate *lock records* that are inserted in the log entry for the transaction. Lock records contain the lock identifier and sequence number. These lock records are used by both the coherency and recovery code to preserve the ordering of updates generated by different nodes.

For coherency, the lock records ensure that log records applied by a receiver thread are properly interleaved with those sent by other nodes. The lock records included with a received update are used to set the receiver's local sequence number for the lock. If necessary, receiver threads hold log records until the updates for the immediately preceding sequence number have been applied. Also, the sequence number must match the sequence number passed with the lock token before the lock can be acquired on that node. An application that tries to acquire the lock prematurely will wait on a condition variable until signaled by the receiver thread that applies the latest update. This interlock between coherency and synchronization is necessary because updates are broadcast asynchronously; the lock token could arrive at a node before all necessary coherency updates have been received and applied. For example, if the token passes in order through nodes A, B, and C, C might receive the token from B before A's updates arrive at C. The protocol ensures that, for updates made to data protected by the same lock, (1) none of B's updates are applied at C until C has applied A's updates and (2) C is not allowed to acquire the lock until B's updates have been applied.

For recovery, lock records are used to merge the individual RVM redo logs produced by each node. When applications share a segment, their logs may record interleaving updates to the same data. Thus, before any of these logs can be used by the standard RVM recovery procedure, they must be merged into a single log. We built a new RVM utility to do this. Our merge utility reads input logs from head to tail; transaction records from different logs are compared by comparing their lock records. Our current merge utility exploits the fact that our current synchronization model guarantees strictly serializable transactions. When merging records from different logs, it is sufficient to order transactions so that if two transactions acquired the same lock, the transaction with the earlier sequence number for that lock is ordered first.

3.5 Distributed Log Trimming

The current RVM log-trimming algorithm has an unfortunate interaction with our distribution implementation. In the current version of RVM, log records are trimmed from the head of the log using the standard recovery procedure to compute a new checkpoint. This operation, which runs asynchronously with normal transactions, is triggered when the number of records in the log reaches a high-water point. In our system, it is no longer possible to use the log generated by a single node to compute a new checkpoint; instead, log records from all nodes must first be merged. Our current prototype performs log trimming offline using the recovery procedure described above. Online trimming could be implemented using the merging procedure by coordinating checkpointing; one node would checkpoint at a time, broadcasting to other nodes when done to inform them of their new log head.

An improved log-trimming scheme for RVM is described in [Satyanarayanan et al. 94]. In this scheme, nodes checkpoint a page at a time by writing the current version of a page to the checkpoint file. Log records for updates made to a page before it was checkpointed can be discarded. This checkpointing scheme could be more easily incorporated into our prototype, because it does not require logs to be merged.

4 Performance

This section presents performance measurements of our prototype. While most DSM systems have used parallel programs for evaluation, our application domain — collaborative design applications accessing a distributed persistent store — requires a different benchmark. For this reason, we have chosen to use OO7 [Carey et al. 93], a standard object-oriented database benchmark. OO7 consists of a number of different traversals, updates, and queries of a synthetic object-oriented database. The database and the traversal tests are intended to be suggestive of typical engineering database applications.

We modified OO7 to run with RVM in standard virtual memory (i.e., no OODB) and integrated it with our log-based coherency prototype. We report timings and related overheads for several OO7 traversals using our prototype. These experiments were conducted using two Digital 3000-400 Alpha APX (133 Mhz, 74 SPECints) workstations with 8-Kbyte pages and separate 512-Kbyte direct-mapped instruction and data caches [Dig 92]. The machines are connected by a 100 Mbit/s AN1 network, an experimental switch-based network capable of sending message packets of up to 64 Kbytes [Rodeheffer & Schroeder 91]. Elapsed time measurements were taken using the Alpha cycle counter. For these experiments, we disabled RVM disk logging so that we could isolate the costs associated with coherency from the synchronous disk writes needed to support recovery. This is important in part because optimizations using non-volatile RAM can be used to eliminate synchronous disk writes from the commit critical path [Hagmann 86].

The figures also depict estimated lower bounds for alternative DSM implementations that use page access faults to capture updates. For log-based coherency, updates are captured by calls to the RVM `set_range` procedure. These calls are coded explicitly in our OO7 benchmarks, although other RVM applications have used compiler-generated `set_range` calls [O'Toole et al. 93]. In contrast, most DSM systems use page access faults to capture updates without involvement from the compiler or application. Early page-locking DSM systems, such as Monads and IVY, use page access faults to grant a writer exclusive access to a page while updates are in progress, then transmit the entire contents of each modified page to other nodes accessing the page. In newer multiple-writer "copy/compare" DSM systems, such as Munin and TreadMarks, updates are also detected a page at a time, but multiple nodes are permitted to make concurrent non-conflicting updates to any given page. In these systems, the first store to a unmodified page on each node delivers a write-access fault to the coherency software, which makes a copy of the page before enabling write access. Updates are collected by later comparing the modified page with its copy. The copy/compare technique could improve performance for some OODBs that use page-grained locking and updates today.

Our lower-bound estimates for page-locking (labeled *Page*) and multiple-writer (labeled *Cpy/Cmp*) DSM systems are computed from the measurements listed in Table 2. We measured the cost of using the OSF/1

Operation	**Cost** (μsec/page)	**Throughput** (MBytes/s)
page copy (cold cache)	171.9	43
page copy (warm cache)	57.8	135
page compare (cold cache)	281.0	28
page compare (warm cache)	147.3	53
page send (TCP/IP)	677.0	12 (96.8 Mbit/s)
handle signal and change protection	360.1	

Table 2: *Operation costs (per page) on Alpha/AN1.*

`mprotect` system call to change page protection, by storing to a read-only page to generate a protection fault, delivering the signal to a user-level procedure, calling `mprotect` again to enable writing, and returning from the signal handler. We also measured the time to copy and compare pages (we use the cold cache times in the figures). For page-grain DSM (Page), updated pages are not copied or scanned, so we assume no collection overhead. However, network I/O overhead for Page is greater because entire pages are transmitted instead of just the modified bytes. This time is estimated from the measured TCP throughput given in Table 2. Communication overhead for Cpy/Cmp is assumed to be the same as the measured times for log-based coherency (labeled *Log*), since both send only the modified bytes. Both Log and Cpy/Cmp also incur overhead at the receiver to *apply* the updates to the cache; however, this cost is too small to be clearly distinguished in any of the graphs below.

4.1 The OO7 Benchmark

The OO7 database is composed of a *design library* and an *assembly hierarchy*. The design library, which makes up the bulk of the database, is a set of 500 composite parts. A composite part corresponds to a design primitive, such as a register cell in a VLSI CAD application or a procedure in a CASE application. Each composite part is itself made up of a graph of 20 atomic parts. Composite and atomic part objects are each roughly 200 bytes long. Each atomic part contains an index field; a part index is maintained using a self-balancing tree.

Objects in the assembly hierarchy correspond to higher-level design constructs in the application. The assembly hierarchy is a tree of assembly objects with 729 leaf nodes. Each leaf node, called a base assembly, points to three composite parts that are chosen at random from the design library when the database is constructed.

There are two traversals in OO7 that update the database, T2 and T3. Both traverse the assembly hierarchy to visit all of the composite parts pointed to by each base assembly; thus, a total of 2187 composite parts are visited. When a composite part is visited, its atomic-part graph is traversed and updates are performed. There are three variants of the traversals (A, B, and C) that update different numbers of atomic parts. In A, one atomic part per composite part is modified; in B, every atomic part is modified; and in C, every atomic part is modified four times. An atomic part is updated by changing an eight-byte field. The difference between T2 and T3 is that T3 updates the atomic part's index field. Each time this field is changed, the part index is updated by deleting the index entry for the old value and adding an entry for the new value. This results in an average of seven index updates for each atomic-part update.

We added a third update traversal, called T12. T12 differs from the other update traversals in that it performs a sparse traversal of the database. It is similar to read-only traversal T6; for each composite part it visits only one atomic part. In T12, a higher percentage of overall running time is related to updating objects. This highlights the performance costs associated with coherency overhead.

In our RVM-based OO7 benchmark, the database elements are heap-allocated C++ objects and a threaded AVL-balanced tree is used for the part index. The atomic parts associated with a particular composite part tend to be clustered on the same page while atomic parts from different composite parts are usually on different pages. We ran each of the OO7 update traversals under our prototype. Each test consists of a single transaction (and a single segment lock) in which one node performs the traversal and another receives the log tail and installs the updates, bringing its copy of the database up to date.

4.2 Results

Figures 1, 2, and 3 show the results of running the various OO7 traversals, along with the associated coherency overhead. Write overhead consists of *detecting* and *collecting* updates and the *network I/O* costs of transmitting those updates to the other node. Table 3 summarizes the characteristics of these traversals, listing the number of updates performed by each, the

Traversal	Updates	Bytes Updated	Message Bytes	Pages Updated
T12-A	2,187	4,000	6,000	500
T12-C	8,748	4,000	6,000	500
T2-A	2,187	4,000	6,000	500
T2-B	43,740	80,000	120,000	618
T2-C	174,960	80,000	120,000	618
T3-A	16,924	31,300	39,000	552
T3-B	248,632	114,650	163,300	667
T3-C	1,502,708	115,100	163,800	670

Table 3: *Summary of OO7 update-traversal characteristics*

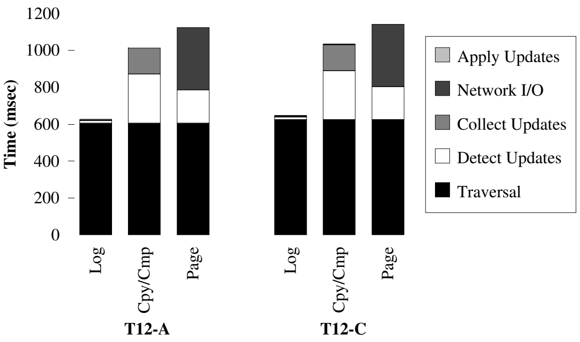

Figure 1: *OO7 Sparse-update traversals T12-A and T12-C.*

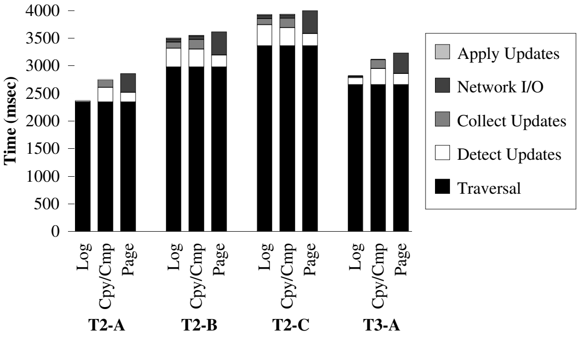

Figure 2: *OO7 Full-update traversals T2-A, T2-B, and T2-C, and index-update traversal T3-A.*

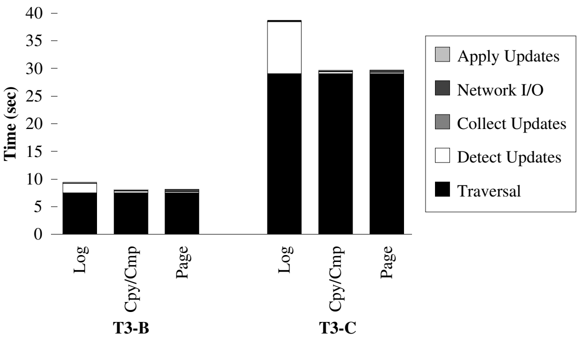

Figure 3: *OO7 Index-update traversals T3-B and T3-C.*

number of unique bytes updated, the number of bytes sent over the network, and the number of pages updated. (The difference between the number of bytes updated and the number transmitted is due to range-message overhead; each range is preceded by a header that describes the address of the range and its size.)

Based on the Alpha-OSF/1 measurements, our analysis shows that for the anticipated application workload, in which updates affect a large number of pages, software write detection (e.g., compiler-generated `set_range` calls) performs better than any form of hardware-based write detection. Our approach performs significantly better for traversals T12-A, T12-C, T2-A, and T3-A because they perform relatively few updates per page. Traversals T2-B and T2-C perform 71 and 283 update per page respectively; for these traversals, our approach performs about as well as Cpy/Cmp. However, the index-update traversals T3-B and T3-C perform significantly more updates per page, 373 and 2,243; as a result our approach performs poorly. This shows that when there are many updates per page, a page-based systems such as TreadMarks is preferred to a software-based approach such as log-based coherency.

4.3 Analysis

There is a performance tradeoff between the three approaches evaluated above that is a function of (1) the number of modified bytes per page and (2) the number of individual updates per page. The overhead of Log and Cpy/Cmp depends on the number of modified bytes, Cpy/Cmp and Page depend on the number of modified pages, and Log alone depends on the number of updates per page. Which approach works best depends on the workload. As we have seen, log-based coherency is preferred when there are few updates per page; similarly, Page performs best when most of a page is modified. Figures 4–7 below show where the breakeven points occur.

Figure 4 shows coherency overhead as a function of the number of modified bytes per page. This compares

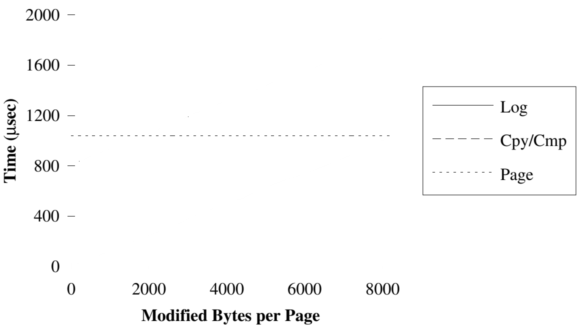

Figure 4: *Comparison of overhead as the number of modified bytes per page increases. For log-based coherency, per-update overhead is not included.*

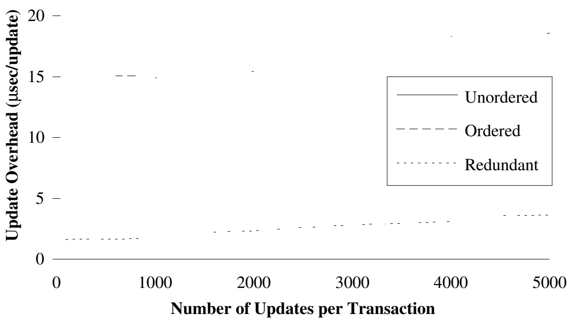

Figure 5: *The overhead associated with a single update for log-based coherency as the number of updates per transaction increases.*

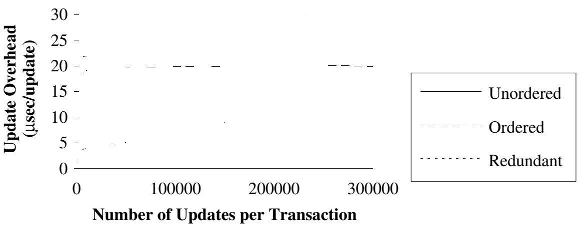

Figure 6: *Log-based coherency update overhead up to 300,000 updates per transaction.*

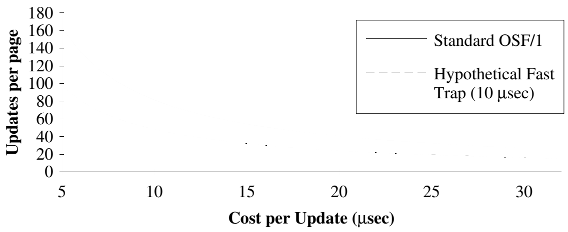

Figure 7: *Breakeven point for log-based coherency. For different update overheads, the number of updates per page at which log-based coherency performs better than Cpy/Cmp.*

the per-byte overhead of log-based coherency with the total overhead of copy/compare and page approaches. When more than 1037 bytes are modified per page, Page outperforms Cpy/Cmp. When there are few updates per page, Log outperforms the alternatives no matter how many bytes are modified. However, this graph does not include the per-update overhead associated with Log; this is shown in Figures 5–7.

Figures 5 and 6 show the log-based coherency overhead associated with detecting and collecting a single update as the number of updates per transaction increases. This is a measurement of the performance of RVM operations `set_range` and `rvm_commit`. The middle line is the cost for an update in an ordered sequence of `set_range` calls (taking advantage of the optimization described in Section 3.1). The lower line is the cost of detecting an update to a range that was modified previously in the same transaction.

Figure 7 shows the breakeven point at which log-based coherency and Cpy/Cmp have equivalent performance. For a given average per-update cost on the x-axis, the y-axis shows the maximum number of updates per page possible before Cpy/Cmp outperforms log-based coherency. For example, using Figures 5 and 7, we can determine that if there are 1000 updates per transaction, log-based coherency performs better when there are 45 or fewer updates per page (55 if the updates are ordered).

Recent work [Thekkath & Levy 94] has shown an order-of-magnitude reduction in exception-handling cost, which would make hardware-based write detection more attractive. Figure 7 shows how the performance tradeoff would be affected if signal handling overhead were 10 μsec instead of the 340 μsec measured for Alpha-OSF/1.

4.3.1 Increasing the Number of Nodes

Another important performance concern for log-based coherency is the effect of increasing the number of nodes using a segment to be kept coherent. In our prototype the *network I/O* overhead of the writer increases linearly with the number of peer nodes, because the writer node issues separate `writev` system calls for each peer. Since this overhead is relatively small, our approach will scale to a moderate number of nodes. Systems with a very large number of clients will perform better with multicast hardware or lazy coherency.

4.3.2 Impact of Coherency on Recoverability

Finally, we wanted to determine the impact of our modifications on the performance of standard RVM. Figure 8 shows four measurements of the coherency and recoverability overheads for the T12-A bench-

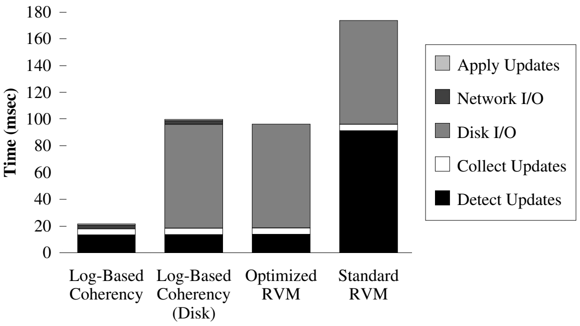

Figure 8: *Comparison of log-based coherency, disk logging, and standard RVM. Optimized RVM is standard RVM with our optimizations to* `set_range`.

mark. The first column is the overhead for log-based coherency presented in the previous section. The next column measures the overhead when disk logging is enabled; from this we see that the only additional overhead is due to writing the log tail to disk. The third and fourth columns were taken using standard RVM, without our log-based coherency modifications; the third column is standard RVM with our optimized `set_range` modification. This shows that the overhead added to RVM by log-based coherency is directly related to sending the modified bytes to peer nodes. This validates our assertion that there is a high degree of overlap between the mechanisms for recoverability and coherency.

5 Related Work

In the context of RVM, transaction logs have been used in a similar fashion to propagate updates between data spaces in a system with concurrent replicating garbage collection [O'Toole et al. 93]. Our contribution is to use this idea for maintaining coherency of client database caches. Log propagation has been used to maintain the consistency of replicas in replicated database systems. Replicated systems differ from log-based coherency in that the replicas are used to ensure availability of the data. In replicated systems, log records flow in one direction: from clients to servers (or from a client to a server, and then on to other servers). Servers cooperate to ensure that log records are applied in a consistent order at all locations. In our system, the replicas are caches that keep only enough data to meet the local client's needs; update propagation may be delayed until a client requests the new data.

Harp [Liskov et al. 92] file servers log received updates to peer servers in order to remove stable storage writes from the commit path. Similarly, Naughton and Li have used log propagation to keep a hot standby of a main-memory database [Li & Naughton 88]. The standby keeps a complete copy of the database in its memory and receives updates from the primary in the form of log records sent at commit points. Their purpose is to allow checkpointing to take place in the standby, off-line, without interfering with clients executing on the primary copy of the database. Delis and Roussopoulos conducted a simulation study of client-server relational databases using a log-based approach for updating client caches [Delis & Roussopoulos 92]. Updates are centralized at the server; the server maintains a recovery log and a separate update log. Before a client accesses a data item in its cache, it first contacts the server to retrieve log records generated since the cached copy of the item was last updated. While several of these systems send log records from clients to one or more servers, log-based coherency sends log records from client to client.

Several groups have integrated coherency and recoverability by adding checkpointing to page-based DSM, without using transactions. This allows non-transactional applications such as parallel programs to be made recoverable. The main issue is to coordinate individual node checkpoints to attain a consistent global checkpoint, using a combination of dependency tracking, message logging, and replication. The first such system is due to Wu and Fuchs [Wu & Fuchs 90]. Stumm and Zhou describe a system that tolerates the failure of a single node by ensuring that every page resides in the caches of at least two nodes [Stumm & Zhou 90]. Richard and Singhal use page logging [Richard & Singhal 93]; a copy of a page is written to a local volatile log each time it is acquire for reading or writing. A node writes its volatile log to disk before transferring a modified page to another node.

Janssens and Fuchs added checkpointing to relaxed-consistency DSM [Janssens & Fuchs 93]. Instead of requiring checkpointing or other recoverability actions each time an application gains access to a page, their system checkpoints only when a node releases or acquires a lock.

Neves et al. added checkpointing to an entry-consistent DSM system similar to Midway, using object-grain locking [Neves et al. 94]. Their system tolerates single node failures by keeping old versions of modified objects in the volatile memory of the nodes that modify them. Each node checkpoints independently; a failed node recovers by replaying its execution starting with its most recent checkpoint. Information recorded at other nodes is sufficient for them to supply the recovering node with the same version of objects as it saw during normal execution. The impact on failure-free execution is minimized by piggy-backing recoverability information on coherency messages.

6 Conclusion

The key points of this paper are: (1) DSM techniques such as fine-grained client-client transfers are appropriate for maintaining cache coherency for distributed persistent stores, (2) there is a commonality between the implementation techniques for recoverability and coherency, and (3) this synergy can be exploited when both properties are supported together.

We have presented a new DSM approach called log-based coherency that uses recoverability mechanisms from persistent object systems as the basis for maintaining coherency of distributed objects. Our work extends previous work on DSM to exploit the notion of commit points in which a group of related updates become visible atomically. We extend work on persistent stores to support the fine-grained sharing made possible for parallel applications by DSM systems. In particular, log-based coherency separates the coherency grain from the synchronization grain. This is important for collaborative-design applications, where large data regions are shared among engineers at different nodes, but where updates are sparse and infrequent. With log-based coherency, locking overhead can be reduced by using coarse-grain locks without increasing coherency overhead; i.e., coarse-grain locks can support fine-grain sharing.

While log-based coherency provides an alternative to other DSM systems, the approaches are not mutually exclusive. Our measurements show that application behavior determines the best approach; e.g., if updates are highly clustered within a page, standard DSM techniques will perform better, while for sparse updates, the log-based approach will perform better. Therefore, adaptive hybrid approaches maybe be possible where application behavior can be predicted.

References

[Butterworth et al. 92] Butterworth, P., Otis, A., and Stein, J. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1992.

[Carey et al. 93] Carey, M. J., Dewitt, D. J., and Naughton, J. F. The OO7 benchmark. *1993 ACM SIGMOD. International Conference on Management of Data*, 22(2):12–21, May 1993.

[Carey et al. 94] Carey, M. J., Dewitt, D. J., and Franklin, M. J. Shoring up persistent applications. *1994 ACM SIGMOD. International Conference on Management of Data*, May 1994.

[Carter et al. 91] Carter, J., Bennet, J., and Zwaenepoel, W. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems and Principles*, pages 152–164, October 1991.

[Cockshott et al. 84] Cockshott, W., Atkinson, M., Chisholm, K., Bailey, P., and Morrison, R. Persistent object management system. *Software Practice and Experience*, 14(1), January 1984.

[Delis & Roussopoulos 92] Delis, A. and Roussopoulos, N. Performance and scalability of client-server database architectures. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 610–623, August 1992.

[Dig 92] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Handbook*, 1992.

[Eppinger et al. 91] Eppinger, J., Mummert, L., and Spector, A. *Camelot and Avalon*. Morgan Kaufmann, 1991.

[Feeley & Levy 92] Feeley, M. J. and Levy, H. M. Distributed shared memory with versioned objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1992.

[Gharachorloo et al. 90] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News*, pages 15–26. ACM, June 1990.

[Hagmann 86] Hagmann, R. B. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–843, September 1986.

[Hornick & Zdonik 87] Hornick, M. F. and Zdonik, S. B. A shared, segmented memory system for an object-oriented database. *A ACM Transactions on Office Informations Systems*, 5(1), January 1987.

[Hosking & Moss 93] Hosking, A. L. and Moss, J. E. B. Protection traps and alternatives for memory managemnt of an object-oriented language. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[Janssens & Fuchs 93] Janssens, B. and Fuchs, W. K. Relaxing consistency in recoverable distributed shared memory. In *Proceedings of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 155–163, June 1993.

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

[Keleher et al. 94] Keleher, P., Cox, A. L., Dwarkadas, S., and Zwaenepoel, W. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winder 1994 USENIX Conference*, pages 115–132, January 1994.

[Lamb et al. 91] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[Li & Hudak 89] Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li & Naughton 88] Li, L. and Naughton, J. F. Multiprocessor main memory transaction processing. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pages 177–187, December 1988.

[Liskov 88] Liskov, B. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Liskov et al. 92] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., and Shrira, L. Replication in the Harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 226–238, October 1992.

[Moss 90] Moss, J. E. B. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[Mueller 93] Mueller, F. A library implementation of POSIX threads under Unix. In *Proceedings of the Winter 1993 USENIX Conference*, pages 29–41, January 1993.

[Neves et al. 94] Neves, N., Castro, M., and Guedes, P. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, August 1994.

[O. Deux 92] O. Deux. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1992.

[O'Toole et al. 93] O'Toole, J., Nettles, S., and Gifford, D. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 161–174, December 1993.

[Richard & Singhal 93] Richard, III, G. G. and Singhal, M. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 58–67, October 1993.

[Rodeheffer & Schroeder 91] Rodeheffer, T. and Schroeder, M. D. Automatic reconfiguration in autonet. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 183–197, October 1991.

[Rosenberg & Abramson 85] Rosenberg, J. and Abramson, D. A. MONADS-PC: A capability-based workstation to support software engineering. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, 1985.

[Satyanarayanan et al. 94] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and J.Kistler., J. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(4):33–57, February 1994.

[Stumm & Zhou 90] Stumm, M. and Zhou, S. Fault tolerant distributed shared memory algorithms. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, December 1990.

[Thekkath & Levy 94] Thekkath, C. and Levy, H. Hardware and software support for efficient exception handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[Wu & Fuchs 90] Wu, K.-L. and Fuchs, W. K. Recoverable disributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[Zekauskas et al. 94] Zekauskas, M. J., Sawdon, W. A., and Bershad, B. N. Software write detection for distributed shared memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.