Effective Information Sharing Using Update Logs

by

James William O'Toole Jr.

B.S., University of Maryland at College Park (1984)
S.M., Massachusetts Institute of Technology (1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1996

Signature of Author .....
Department of Electrical Engineering and
Computer Science
, 1996

Certified by .....
David K. Gifford
Professor of Computer Science
Thesis Supervisor

Accepted by .....
Chairman, Departmental Committee on Graduate Students

Effective Information Sharing Using Update Logs

by

James William O'Toole Jr.

Submitted to the Department of Electrical Engineering and
Computer Science
on February 2, 1996, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Information storage systems enable computational activities to preserve valuable information that can be shared by multiple users and for multiple purposes. Automatic indexing and garbage collection are useful features of storage systems because they can make information sharing more convenient and effective. Automatic indexing enables users and programmers to locate and access shared information without substantial prior agreement about information structure and index design, as is typically necessary in traditional file systems and databases. Automatic garbage collection enables users and programmers to share information without requiring complete agreement on ownership and deallocation protocols for the shared objects.

My thesis demonstrates, via two working systems, that update logs can be used to provide automatic garbage collection and indexing services in information storage systems. The garbage collection system uses log-based replicating garbage collection, a new algorithm that enables concurrent compacting garbage collection of persistent storage. In this garbage collector design, an update log is used to record client operations that require subsequent action by the collector. The update log decouples the operations of application programs from the operations of the garbage collector. The indexing file system uses an update log to decouple indexing operations from file modification operations. Both the indexing system and the garbage collection system use update logs in novel ways to implement convergent consistency properties that promote convenient and efficient sharing of information.

Thesis Supervisor: David K. Gifford
Title: Professor of Computer Science

Acknowledgments

Dave Gifford, my advisor, provided me with support and encouragement to pursue unorthodox ideas, guidance (sometimes heeded, sometimes not), and a place to grow.

Butler Lampson, Barbara Liskov, and Frans Kaashoek provided research advice and critical judgement.

Andrew Myers, Mark Sheldon, Frans Kaashoek, Barbara Liskov, and Butler Lampson carefully read and commented on drafts of the dissertation.

Scott Nettles, Liuba Shrira, Dawson Engler, and Frans Kaashoek made joint research fun and exciting.

Jonathan Rees, Gerry Sussman, Franklyn Turbak, Bill Rozas, Olin Shivers, Phil Greenspun, Mark Sheldon, Brian LaMacchia helped make the fourth floor a great place for new ideas. Bob Taylor, Chuck Thacker, Mike Schroeder, Mike Burrows, Mark Manasse, John Ellis, Butler Lampson, and many others at Digital's Systems Research Center created an exciting and stimulating research environment for me during the summer of 1990 when the idea of replicating garbage collection was first conceived.

Mark Sheldon, Ron Weiss, Brian Reistad, Pierre Jouvelot, Andrzej Duda, Jeanne Darling, and Rebecca Bisbee helped make the Programming Systems Research Group a fun place to be.

Franklyn Turbak, Jonathan Rees, Dave Gifford, and Gerry Sussman helped me learn to speak by their good example and excellent advice.

Gerry Sussman, Peter Szolovits, and Jonathan Rees gave me forthright advice in difficult times, and Mark Weiser, Sam Bowen, and my father, James W. O'Toole, generously taught me how to think, judge, and act, long before I arrived at MIT; their wisdom was invaluable.

Rich Lethin, Mark S. Day, Andrew Myers, Ulana Legedza, Kavita Bala, and Patrick Sobalvarro added strong bonds of friendship to graduate school and generally made life better.

Trevor Darrell and Andrew Myers made joint living in Cambridge fun and exciting.

Eric C. Liebeler III, Mei-lin Wong, Andy Berlin, Kathy Yelick, Dennis J. Arnow, James S. Person III, and Nathan Glasser welcomed me to MIT and injected valuable doses of humor, joy, irreverence and sanity into my life.

Jim, Edwina, Frank, Regina and Margaret E. O'Toole provided me with the unconditional love and patience that a little kid needs. Michelle Nicholasen provided me with the love and companionship that a big kid needs.

Thanks.

Contents

Chapter 1

My Thesis

Update logs can be used to provide automatic indexing and garbage collection services in information storage systems.

1.1 Useful Services for Information Sharing

Information storage systems enable computational activities to preserve valuable information to be shared by multiple users and for multiple applications. Many computational activities depend on storage systems to preserve valuable information that is both a work product and a subsequent work input. Traditional information storage systems have provided this service primarily by supporting the reliable storage of raw data files. The choice of raw data files as the interface between the storage system and its clients provides a degree of security because the integrity of the data is independent of its structure. Nevertheless, by providing more information about the structure of data, clients can enable storage systems to provide better performance or additional services.

Computer systems are now supporting a great variety of human activity, and there is consequently an explosion of information resources available in existing storage systems. One of the greatest potential benefits of increased use of computer systems is that they can enable people to more easily communicate, investigate new ideas, and rapidly acquire

new knowledge. However, when the information stored in these systems cannot be located and shared conveniently, much of this potential is lost. Storage services that provide more effective and convenient ways to share information are therefore valuable.

I believe that in the future it will be necessary and practical to implement storage systems with interfaces that provide more convenient ways to share information than current storage systems provide. In the course of my research in this direction, I implemented the Semantic File System and the Log-Based Replicating Garbage Collector. The Semantic File System is a file storage system that provides content-based indexing of files in a portable and extensible way. I built the Semantic File System because I expected that the integration of extensible indexing and content-based access primitives into the storage interface would help users of the file system locate and share data more easily. The Log-Based Replicating Garbage Collector is part of a heap-based storage system that provides transaction and concurrent garbage collection services. I built the Replicating Garbage Collector because I believe that the use of automatic garbage collection in a programming environment makes it easier for programmers to design applications that share data. I expected its interactive performance to be superior to that of other garbage collectors and that this advantage would be important in actively used storage systems of the future.

1.2 Semantic File Systems

Semantic File Systems were developed as an approach to information storage that would permit users to share information more effectively. A primary design goal of the first semantic file system was to provide a transition path from existing file systems, while presenting a more effective storage abstraction than traditional tree structured file systems. A semantic file system is an information storage system that provides flexible associative access to the system's contents by automatically extracting attributes from files with file-type-specific *transducers*. Associative access is provided by a conservative

extension to existing tree-structured file system protocols, and by protocols that are designed specifically for content-based access. Automatic indexing is performed when files or directories are created or updated.

The automatic indexing of files and directories is called "semantic" because user programmable transducers use information about the semantics of updated file system objects to extract the properties for indexing. Through the use of specialized transducers, a semantic file system "understands" the documents, programs, object code, mail, images, name service databases, bibliographies, and other files contained by the system. For example, the transducer for a C program could extract the names of the procedures that the program exports or imports, procedure types, and the files included by the program. A semantic file system can be extended easily by users through the addition of specialized transducers.

Associative access is designed to make it easier for users to share information by helping them discover and locate programs, documents, and other relevant objects. For example, files can be located based upon transducer-generated attributes such as author, exported or imported procedures, words contained, type, and title.

A semantic file system provides both a user interface and an application programming interface to its associative access facilities. User interfaces based upon browsers [26, 73] have proven to be effective for query-based access to information, and browsers could also be provided by semantic file system implementations. Application programming interfaces that permit remote access include specialized protocols for information retrieval [46], and remote procedure call based interfaces [21].

It is also possible to export the facilities of a semantic file system without introducing any new interfaces. This can be accomplished by extending the naming semantics of files and directories to support associative access. A benefit of this approach is that all existing applications, including user interfaces, immediately inherit the benefits of associative access.

A semantic file system integrates associative access into a tree structured file system

through the concept of a *virtual directory.* Virtual directory names are interpreted as queries and thus provide flexible associative access to files and directories in a manner compatible with existing software.

Semantic file systems may be useful to both individuals and groups. Individuals can use the query facility of a semantic file system to locate files and to provide alternative views of data. Groups of users should find semantic file systems an effective way to learn about shared files and to keep themselves up to date about the status of group projects. As workgroups increasingly use file servers as shared library resources, automatic indexing technology for file systems will become even more useful.

Because semantic file systems are compatible with existing tree structured file systems, implementations of semantic file systems can be fully compatible with existing network file system protocols such as NFS [62, 70] and AFS [28]. NFS compatibility permits existing client machines to use the indexing and associative access features of a semantic file system without modification. Files stored in a semantic file system via NFS will be indexed automatically and query result sets will appear as virtual directories in the NFS name space.

I built a semantic file system and ran some experiments to explore its practicality and its usefulness. These experiments suggest that semantic file systems can be used to find information more quickly than is possible using ordinary file systems. Experience with the implementation demonstrates that the practicality of the system depends substantially on the use of an update log, which enables incremental indexing of modified file system objects and decouples the execution of transducers from the actual file system update operations.

1.3 Log-Based Replicating Garbage Collection

Operating systems, persistent programming environments, and object repositories must store dynamically-allocated persistent data structures. Unfortunately, using explicit deal-

location to manage these data structures can easily cause catastrophic system failures due to the effects of dangling pointers and storage leaks. These problems can discourage users and programmers from sharing dynamically allocated data structures.

Permanent data storage management should meet the much higher safety standard achieved by tracing garbage collection. Modern garbage collectors are very efficient and can often be competitive with explicit deallocation [78]. However, existing garbage collectors are not practical for use in systems applications. Existing implementations either stop the client while garbage collecting or do not maintain a heap image that survives system failure.

I have developed a new algorithm, Log-Based Replicating Garbage Collection, and used it in the design and implementation of a concurrent compacting garbage collector for a persistent heap. Clients read and write the heap while the collector concurrently replicates objects to create a new stable heap. Clients are free to modify objects that have already been copied because the modifications are recorded in an update log. The log is used by the collector to ensure that the new stable heap contains all pertinent data before it is used to replace the old stable heap. The update log is processed concurrently, reducing garbage collection interruptions imposed on the clients to brief synchronization pauses.

There are several important advantages that derive from using log-based replicating garbage collection in a transactional heap system:

- The garbage collector is simple and easy to implement.
- Transaction processing and garbage collection operations are decoupled so that collector activity need not affect transaction throughput or commit latencies.
- A single log is shared by the transaction manager and the garbage collector.

In practice, the log-based replicating collection provides clients more responsive access to the heap than does a stop-and-copy collector. Concurrent replicating collection eliminates lengthy interruptions caused by garbage collection. The implementation supports

transactions, recovers from system failures, and provides good performance.

1.4 Using Update Logs

In building the Semantic File System and the Replicating Garbage Collector, I found that using an update log simplified the design of both systems. Both the indexing system and the garbage collection system use update logs in novel ways to implement convergent consistency properties that promote convenient and efficient sharing of information. As I report in Chapter 2, the use of an update log in the Semantic File System makes possible the implementation of incremental indexing, although obtaining an accurate update log for the file system is difficult with the existing NFS protocol.

In the case of the replicating garbage collector, the use of an update log simplified the design and implementation of the system. The update log allows the collector to use a novel invariant in which the client program uses only From-space. This simplification is probably largely responsible for the ease with which Scott Nettles and I were able to build a concurrent garbage collector for a persistent heap, whereas previous attempts using alternative designs were quite complex and not completely implemented.

Both the Semantic File System and the Replicating Garbage Collector use logs to ensure that the system converges to a consistent state. These systems do not provide absolute consistency. The semantic file system, as implemented, responds to queries without waiting for deferred indexing operations to complete. I believe that this is a desirable property of the system because the expense of providing absolute consistency would make the associative access features useless. In the garbage collector, absolute immediate consistency would require objects to be garbage collected as soon as they become unreachable from the root. Again, this is not necessary for the garbage collection system to be useful; it is sufficient to ensure that the system is converging towards having unreachable objects collected.

1.5 Contributions of This Work

The work described here provides a concrete demonstration that there are practical ways to support information sharing in storage interfaces. I have demonstrated, via two working systems, that update logs can be used to provide automatic garbage collection and indexing services in information storage systems. Automatic indexing and garbage collection are useful features of storage systems because they can make information sharing more convenient and effective. Automatic indexing enables users and programmers to locate and access shared information without substantial prior agreement about information structure and index design, as is typically necessary in traditional file systems and databases. Automatic garbage collection enables users and programmers to share information without agreement on ownership protocols for the shared objects.

The implementation of these services are novel. No previous work on indexing file systems provided the combination of compatibility with existing network protocols and extensibility by user-programmable transducers. Also, no previous work generated an update log by monitoring file system traffic and used that update log in order to implement automatic indexing. No previous garbage collector implementation provided concurrent compacting garbage collection of a persistent heap. In addition, replicating garbage collection is the only technique to use a complete update log and a design in which all client operations use the from-space heap.

1.6 Who Did What

Essentially all the work reported in this thesis was performed jointly to a substantial degree. Replicating Garbage Collection was first conceived by me and Scott Nettles in the summer of 1990 at Digital's Systems Research Center. The Semantic File System project was first conceived by Dave Gifford and Mark Sheldon, who were exploring ideas for providing intelligent information discovery facilities in traditional storage systems in mid-1990. As in any group project, it is nearly impossible to assign credit for particular

ideas. I will try to identify some contributions to the best of my recollection.

Replicating garbage collection, although extensively debated by Scott Nettles and myself during the summer of 1990, was little more than a wild idea, a design memo, and some plans for how to try the method within Standard ML of New Jersey. Scott Nettles returned to CMU, and I to MIT, with the best intentions to work on other topics for Ph.D. research.

However, Scott later suggested the idea to David Pierce and Nicholas Haines, who began work on modifying SML/NJ to use an incremental but non-concurrent variant of replicating garbage collection. At some point, Scott completed that first implementation, and together we wrote a short paper [41].

At a meeting in France in September 1992, I identified a flaw in the design relating to interactions with generational collection and we agreed to reimplement the garbage collector and improve its performance. I helped rewrite portions of the implementation, but I believe Scott handled all aspects of benchmarks and performance testing at CMU. We met in Cambridge at MIT for shoulder-to-shoulder writing sessions and remote-control benchmarking. These efforts in late 1992 produced a much stronger presentation of incremental replicating garbage collection [40].

Scott and I discussed the possibility of applying replicating garbage collection to object caches, persistent storage, Scheme48, and how, in general, replicating gc would take over the world of automatic storage management. Scott suggested that by using portions of his Venari transaction system, we could extend replicating gc into a persistent heap very easily. Together, we designed the stable replicating garbage collector. While Scott implemented a simplified transaction manager, I ported our existing replicating gc implementation to SunOS and Irix, restructured it to add concurrency, and tested it on an SGI multiprocessor. Dave Gifford joined Scott and me in writing a paper describing the persistent heap design and implementation [49], and I've used most of that paper to report on replicating garbage collection in Chapter 3 of this dissertation.

In the course of that effort, I produced simple arguments for the correctness of our

implementation with respect to concurrency, heap integrity, and fault tolerance. Scott and I each reimplemented the log-processing module at least once, and made essentially all implementation decisions jointly. We conceived a large number of additional optimizations, one of which we reported together [39], and many of which Scott subsequently explored in his Ph.D. dissertation [38]. Later, Scott modified the concurrent garbage collector's convergence strategy to improve pause time performance, and we collaborated on a paper explaining some implementation details of the concurrent collector [52].

In the case of the Semantic File System, I built the first prototype of the Semantic File System in early 1991 using a previously existing user-level NFS server and other hodge podge software, in an attempt to rapidly implement a search interface defined by Dave Gifford and Mark Sheldon. Later, Pierre Jouvelot and Mark Sheldon formalized the virtual directory specifications. At the same time, I redefined the transducer interfaces and reimplemented portions of the indexing and server systems to obtain performance that was tolerable for daily use in our research group.

Dave, Mark, Pierre and I co-authored a paper describing SFS [22], and I've used that paper with only minor changes to report on SFS in Chapter 2 of this dissertation. Subsequent to that effort, I made some improvements to the implementation, which was being used by Mark Sheldon, Ron Weiss, Andrzej Duda, Dave Gifford, and myself to operate a search distribution server [66]. Mark Sheldon continued to investigate ideas that he originated in the SFS project by exploring search distribution systems and developing an architecture for network-based information discovery [65].

1.7 Previous Work

Associative access to on-line information was pioneered in early bibliographic retrieval systems where it was found to be of great value in locating information in large databases [61]. The utility of associative access motivated its subsequent application to file and document management. The previous research includes work on personal computer indexing

systems, information retrieval systems, distributed file systems, new naming models for file systems, and wide-area naming systems:

- Personal computer indexing systems such as On Location [72], Magellan [13], and the Digital Librarian [45, 44] provide window-based file system browsers that permit word-based associative access to file system contents. Magellan and the Digital Librarian permit searches based upon boolean combinations of words, while On Location is limited to conjunctions of words. All three systems rank matching files using a relevance score. These systems all create indexes to reduce search time. On Location automatically indexes files in the background, while Magellan and the Digital Librarian require users to explicitly create indexes. Both On Location and the Digital Librarian permit users to add appropriate keyword generation programs [11, 45] to index new types of files. However, Magellan, On Location, and the Digital Librarian are limited to a list of words for file description.

- Information retrieval systems such as Basis [26], Verity [73], and Boss DMS [36] extend the semantics of personal computer indexing systems by adding field specific queries. Fields that can be queried include document category, author, type, title, identifier, status, date, and text contents. Many of these document relationships and attributes can be stored in relational database systems that provide a general query language and support application program access. The WAIS system permits information at remote sites to be queried, but relies upon the user to choose an appropriate remote host from a directory of services [27, 68]. Distributed information retrieval systems [21, 15] perform query routing based upon database content labels to ensure that all relevant hosts are contacted in response to a query.

- Distributed file systems [71, 28] provide remote access to files with tree structured names. These systems have enabled file sharing among groups of people and over wide geographic areas. Existing UNIX tools such as `grep` and `find` [23] are often used to perform associative searches in distributed file systems.

- New naming models for file systems include the Portable Common Tool Environment (PCTE) [20], the Property List DIRectory system (PLDIR) [37], Virtual Systems [43] and Sun's Network Software Environment (NSE) [69]. PCTE provides an entity-relationship database that models the attributes of objects including files. PCTE has been implemented as a compatible extension to UNIX. However, PCTE users must use specialized tools to query the PCTE database, and thus do not receive the benefits of associative access via a file system interface. The Property List DIRectory system implements a file system model designed around file properties and offers a Unix front-end user interface. Similarly, Virtual Systems permit users to hand-craft customized views of services, files, and directories. However, neither system provides automatic attribute extraction (although [37] alludes to it as a possible extension) or attribute-based access to their contents. NSE is a network transparent software development tool that allows different views of a file system hierarchy called *environments* to be defined. Unlike virtual directories, these views must be explicitly created before being accessed.

- Wide-area naming systems such as X.500 [10], Profile [56], and the Networked Resource Discovery Project [64] provide attribute-based access to a wide variety of objects, but they are not integrated into a file system nor do they provide automatic attribute-based access to the contents of a file system.

1.7.1 File System Extensions

Previous research supports the view that overloading file system semantics can improve system uniformity and utility when compared with the alternative of creating a new interface that is incompatible with existing applications. Examples of this approach include:

- Devices in UNIX appear as special files [59] in the `/dev` directory, enabling them to be used as ordinary files from UNIX applications.

- UNIX System III named pipes [60, p. 159f] appear as special files, enabling programs to rendezvous using file system operations.

- File systems appear as special directories in Automount daemon directories [7, 54, 55], enabling the binding of a name to a file system to be computed at the time of reference.

- Processes appear as special directories in Killian's process file system [29], enabling process observation and control via file operations.

- Services appear as special directories in Plan 9 [57], enabling service access in a distributed system through file system operations in the service's name space.

- Arbitrary semantics can be associated with files and directories using Watchdogs [6], Pseudo Devices [75], and Filters [43], enabling file system extensions such as terminal drivers, network protocols, X servers, file access control, file compression, mail notification, user specific directory views, heterogeneous file access, and service access.

- The ATTIC system [8] uses a modified NFS server to provide transparent access to automatically compressed files.

1.7.2 Garbage Collection

The basic literature on uniprocessor garbage collection techniques is surveyed by Wilson [76]. Discussions of persistent heaps and language support for transactions appear in work on Persistent Algol [4] and Argus [34]. Earlier descriptions of replicating garbage collection appear in work by O'Toole and Nettles [52, 49, 40, 41], and an instance of the basic technique is also described by Huelsbergen and Larus [25].

There is one earlier working implementation of a concurrent collector for a persistent heap. Almes [1] designed and implemented a mark-and-sweep collector for use in the Hydra OS for C.mmp. The collector is based on Dijkstra's concurrent mark-and-sweep

algorithm [18]. There are two key differences between this work and our own. First, it cannot relocate objects and therefore offers no opportunities for heap compaction or clustering of objects for fast access. Second, although it works in the context of persistent data, it is not designed for use in a system with transaction semantics. Because of the rather unusual environment in which it operated, we cannot make any performance comparison between it and our work.

There is recent work ongoing in the EOS system [24]. The EOS design proposes to combine a marking process and a compaction process. An implementation of EOS is underway but details are not available at this time.

Brian Oki designed the Hybrid Log storage organization for Argus [35]. Oki [47] described two possible housekeeping (compaction) methods for the Hybrid Log, one based on a copying collection of the old log to produce a new log, the other based on a copying collection of the stable state. I believe that Oki's intended "snapshot" method is essentially a so-called fuzzy dump technique, and therefore also very similar to the log-based replicating garbage collection algorithm. I believe Oki's design may require additional synchronization with the active client application, because the actions of the housekeeping process depend on the locking state of the individual objects. However, this dependence is probably an artifact of the presence of two-phase commit operations and the explicit treatment of mutex objects, both complications that are not present in my replicating collector design. Another difference with Oki's work is that his storage organization relies on unique object identifiers. It seems clear that this design decision, which may be necessary in any distributed object storage design, pervasively affects other aspects of garbage collector and transaction manager design. Although it would be interesting to compare the hybrid log housekeeping system with replicating garbage collection in spite of these differences, I believe the snapshot-based concurrent housekeeping algorithm was not implemented [48].

There is a long history of incremental and concurrent copying collectors dating back to Baker [5]. These collectors require the client to access the to-space version of an object

during collections and sometimes force objects to be copied so that the client may access them. The technique of Ellis, Li, and Appel [3] enforces this restriction by using virtual memory protection traps to detect certain client accesses and perform required collector work. In contrast, our technique does not constrain the order in which objects are copied nor does it require any special operating system support. I believe that the ability to freely choose the traversal order may be especially important in systems that need to optimize disk access costs.

There are two earlier designs of concurrent copying garbage collectors for persistent heaps, both based on the Ellis, Li, and Appel algorithm. Detlefs [17] described how to apply this algorithm in the transactional environment of Avalon/C++ [16], while Kolodner [31] worked in the context of Argus. In Detlefs's design, the programmer must explicitly manage object persistence at the time of allocation; Kolodner supports orthogonal persistence.

Neither of these designs was completely implemented, probably because of the complexity of using the to-space invariant in a transactional setting (see Section 3.4.7).

Chapter 2

A Semantic File System

A semantic file system is an information storage system that provides flexible associative access to the system's contents by automatically extracting attributes from files with file-type-specific *transducers*. Associative access is provided by a conservative extension to existing tree-structured file system protocols, and by protocols that are designed specifically for content-based access. Automatic indexing is performed when files or directories are created or updated.

The automatic indexing of files and directories is called "semantic" because user-programmable transducers use information about the semantics of updated file system objects to extract the properties for indexing. Through the use of specialized transducers, a semantic file system "understands" the documents, programs, object code, mail, images, name service databases, bibliographies, and other files contained by the system. For example, the transducer for a C program could extract the names of the procedures that the program exports or imports, procedure types, and the files included by the program. A semantic file system can be extended easily by users through the addition of specialized transducers.

A semantic file system integrates associative access into a tree structured file system through the concept of a *virtual directory*. Virtual directory names are interpreted as queries and thus provide flexible associative access to files and directories in a manner

compatible with existing software.

For example, in the following session with a semantic file system the user first locates within a library all of the files that export the procedure `lookup_fault`, and then further restricts this set of files to those that have the extension `c`:

```
% cd /sfs/exports:/lookup_fault
% ls -F
virtdir_query.c@        virtdir_query.o@
% cd ext:/c
% ls -F
virtdir_query.c@
%
```

Because semantic file systems are compatible with existing tree structured file systems, implementations of semantic file systems can be fully compatible with existing network file system protocols such as NFS [62, 70] and AFS [28]. NFS compatibility permits existing client machines to use the indexing and associative access features of a semantic file system without modification.

This chapter describes the interface and semantics chosen for an experimental semantic file system implementation, discusses the design and implementation, and presents experimental results concerning the operation of the system.

2.1 File System Interface

Semantic file systems can implement a wide variety of semantics. In this section I present the semantics that has been implemented.

Files stored in a semantic file system are interpreted by file-type-specific transducers to produce a set of descriptive attributes that enable later retrieval of the files. An *attribute* is a *field-value* pair, where a *field* describes a property of a file (such as its author, or the words in its text), and a *value* is a string or an integer. A given file can have many attributes that have the same field name. For example, a text file would have

as many `text:` attributes as it has unique words. By convention, field names end with a colon.

A user extensible *transducer table* is used to determine the transducer that should be used to interpret a given file type. One way of implementing a transducer table is to permit users to store subtree specific transducers in the subtree's parent directory, and to look for an appropriate transducer at indexing time by searching up the directory hierarchy.

To generalize the unit of associative access beyond whole files and accommodate files (such as mail files) that contain multiple objects, I will refer to the unit of associative access as an *entity*. An entity can consist of an entire file, an object within a file, or a directory. Directories are assigned attributes by directory transducers.

A transducer is a filter that takes a file as input and outputs the file's entities and their corresponding attributes. A simple transducer could treat an input file as a single entity and use the file's unique words as attributes. A complex transducer might perform type reconstruction on an input file, identify each procedure as an independent entity, and use attributes to record their reconstructed types. Figure 2-1 shows examples of an object file transducer, a mail file transducer, and a TEX file transducer.

The semantics of a semantic file system can be readily extended because users can write new transducers. Transducers are free to use new field names to describe special attributes. For example, a CAD file transducer could introduce a `drawing:` field to describe a drawing identifier.

The associative access interface to a semantic file system is based upon queries that describe desired attributes of entities. A *query* is a description of desired attributes that permits a high degree of selectivity in locating entities of interest. The result of a query is a set of files and/or directories that contain the entities described. Queries are boolean combinations of attributes, where each attribute describes the desired value of a field. It is also possible to ask for all of the values of a given field in a query result set. The values of a field can be useful when narrowing a query to eliminate entities that are not

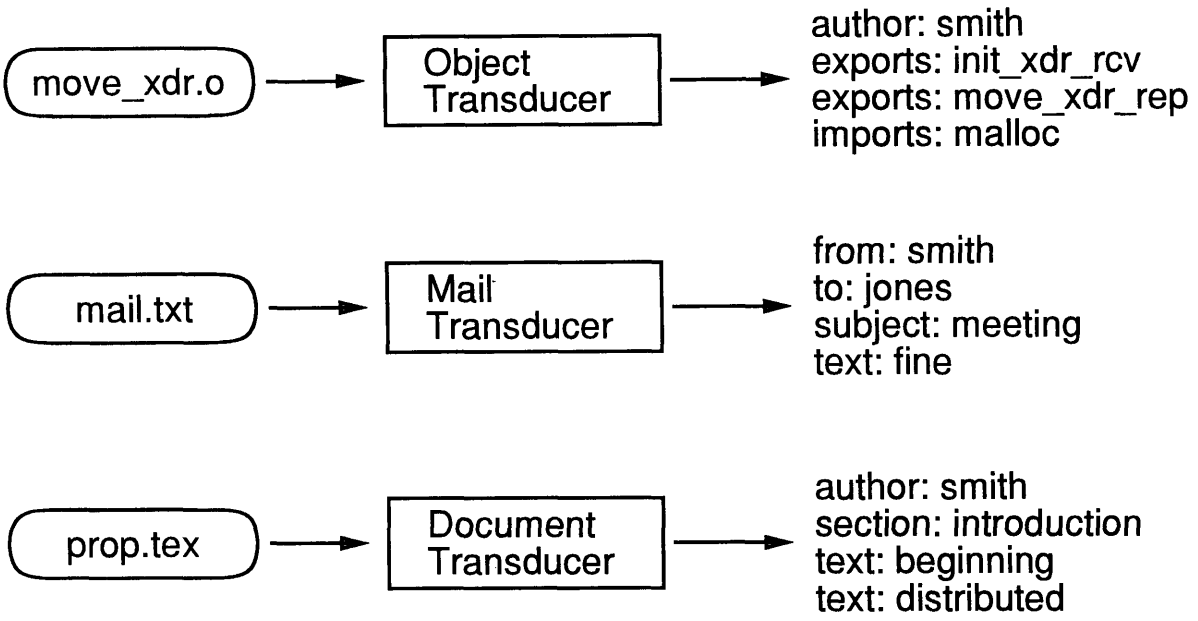

Figure 2-1: Sample Transducer Output

of interest.

A semantic file system is *query consistent* when it guarantees query results that correspond to its current contents. If updates cease to the contents of a semantic file system it will eventually be query consistent. This property is known as convergent consistency. The rate at which a given implementation converges is administratively determined by balancing the user benefits of fast convergence against the higher processing cost of indexing rapidly changing entities multiple times. It is of course possible to guarantee that a semantic file system is always query consistent with appropriate use of atomic actions.

In the remainder of this section I will describe how conjunctive queries are mapped into tree-structured path names. This is only one of the possible interfaces to the query capabilities of a semantic file system. It is also possible to map disjunction and negation into tree-structured names, but they have not been implemented in this prototype.

Queries are performed in a semantic file system through the use of virtual directories to describe a desired view of file system contents. A virtual directory is computed on demand by a semantic file system. From the point of view of a client program, a virtual

directory is indistinguishable from an ordinary directory, except that it is not writeable. However, unlike ordinary directories, virtual directories do not have to be explicitly created to be accessed.

The query facilities of a semantic file system appear as virtual directories at each level of the directory tree. A *field virtual directory* is named by a field and has one entry for each possible value of its corresponding field. Thus in `/sfs`, the virtual directory `/sfs/owner:` corresponds to the `owner:` field. The field virtual directory `/sfs/owner:` would have one entry for each owner that has written a file in `/sfs`. For example:

```
% ls -F /sfs/owner:
jones/          root/           smith/
%
```

The entries in a field virtual directory are value virtual directories. A *value virtual directory* has one entry for each entity described by a field-value pair. Thus the value virtual directory `/sfs/owner:/smith` contains entries for files in `/sfs` that are owned by Smith. Each entry is a symbolic link to the file. For example:

```
% ls -F /sfs/owner:/smith
bio.txt@        paper.tex@      prop.tex@
%
```

When an entity is smaller than an entire file, a view of the file can be presented by extending file naming semantics to include view specifications.

To permit the conjunction of attributes in a query, value virtual directories contain field virtual directories. For example:

```
% ls -F /sfs/owner:/smith/text:/resume
bio.txt@
%
```

A pleasant property of virtual directories is their synergistic interaction with existing file system facilities. For example, when a symbolic link names a virtual directory the link describes a computed view of a file system. It is also possible to use file save programs,

such as `tar`, on virtual directories to save a computed subset of a file system. It would able be possible to generalize virtual directories to present views of file systems with respect to a certain time in the past.

A semantic file system can be overlaid on top of an ordinary file system, allowing all file system operations to go through the SFS server. The overlaid approach has the advantage that it provides the power of a semantic file system to a user at all times without the need to refer to a distinguished directory for query processing. It also allows the server to do indexing in response to file system mutation operations. Alternatively, a semantic file system may create virtual directories that contain links to the files in the underlying file system. This means that subsequent client operations bypass the semantic file system server.

When an overlaid approach is used, field virtual directories must be invisible to preserve the proper operation of tree traversal applications. A directory is *invisible* when it is not returned by directory enumeration requests, but can be accessed via explicit lookup. If field virtual directories were visible, the set of trees under `/sfs` in the above example would be infinite. Unfortunately making directories invisible causes the UNIX command `pwd` to fail when the current path includes an invisible directory, because the `pwd` command searches upward towards the root directory to reconstruct the name of the current directory.

The distinguished `field:` virtual directory makes field virtual directories visible. This permits users to enumerate possible search fields. The `field:` directory is itself invisible. For example:

```
% ls -F /sfs/field:
author:/     exports:/   owner:/     text:/
category:/   ext:/       priority:/  title:/
date:/       imports:/   subject:/   type:/
dir:/        name:/
% ls -F /sfs/field:/text:/semantic/owner:/jones
mail.txt@        paper.tex@     prop.tex@
%
```

The syntax of semantic file system path names is:

```
<sfs-path>    ::=  /<pn> | <pn>
<pn>          ::=  <name> | <attribute>
                   | <field-name> | <name>/<pn>
                   | <attribute>/<pn>
<attribute>   ::=  field: | <field-name>/<value>
<field-name>  ::=  <string>:
<value>       ::=  <string>
<name>        ::=  <string>
```

The semantics of semantic file system path names is:

- The initial universe of entities is defined by the path name prefix before the first virtual directory name.
- The contents of a field virtual directory is a set of value virtual directories, one for each value that the field describes in the universe.
- The contents of a value virtual directory is a set of entries, one for each entity in the universe that has the attribute described by the name of the value virtual directory and its parent field virtual directory. The contents of a value virtual directory defines the universe of entities for its subdirectories. In the absence of name conflicts, the name of an entry in a value virtual directory is its original entry name. Entry name conflicts are resolved by assigning nonce names to entries.
- The contents of a `field:` virtual directory is the set of fields in use.

I have chosen this attribute-based query semantics for virtual directories, but there are many other possibilities. For example:

- The virtual directory syntax can be extended to support a richer query language. Disjunctive queries, textual patterns, and other text-based search operators could be added to the query engine.

- Users could assign attributes to file system entities in addition to the attributes that are automatically assigned by transducers.

- Transducers could be created for audio and video files. This could permit access by time, frame number, or content.

- The entities indexed by a semantic file system could include a wide variety of object types, including I/O devices and file servers. Wide-area naming systems such as X.500 [10] could be presented in terms of virtual directories.

2.2 An NFS Implementation

I have built a semantic file system that implements the NFS [62, 71] protocol as its external interface. To use the search facilities of the semantic file system, an Internet client can simply mount the file system at a desired point and begin using virtual directory names. My NFS server computes the contents of virtual directories as necessary in response to NFS `lookup` and `readdir` requests.

A block diagram of my implementation is shown in Figure 2-2. The dashed lines in the figure describe process boundaries. The major processes are:

- The *client* process is responsible for generating file system requests using normal NFS style path names.

- The *file server process* is responsible for creating virtual directories in response to path name based queries. The SFS Server module implements a user level NFS server and is responsible for implementing the NFS interface to the system. The SFS Server uses *directory faults* to request computation of needed entries by the Virtual Directory module. A faulting mechanism is used because the SFS Server caches virtual directory results, and will only fault when needed information is requested the first time or is no longer cached. The Virtual Directory module

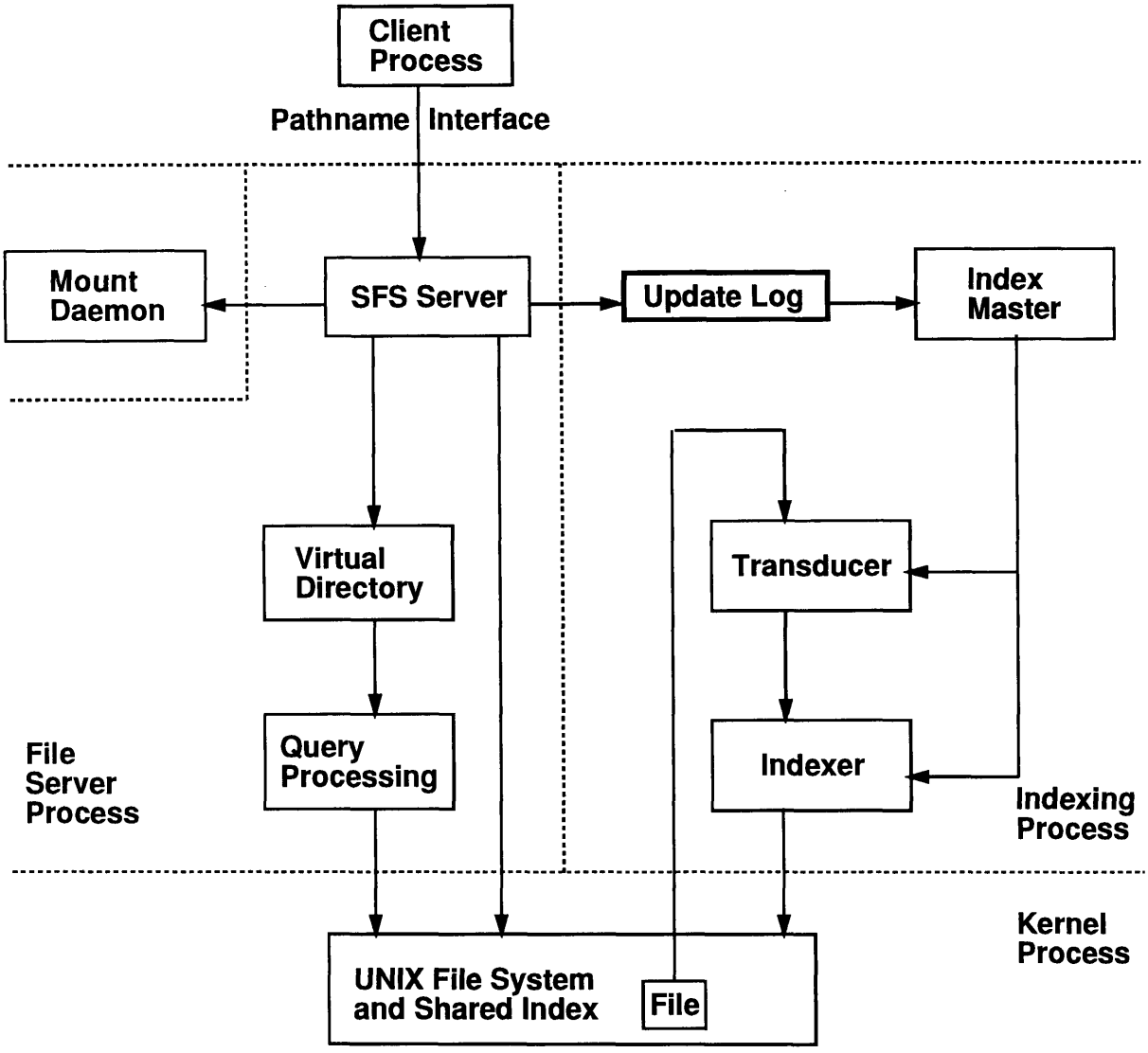

Figure 2-2: SFS Block Diagram

in turn calls the Query Processing module to actually compute the contents of a virtual directory.

The file server process records file system modification events in an update log. The update log is filtered to eliminate duplicate file modification events.

- The *indexing process* is responsible for keeping the index of file system contents up-to-date. The Index Master module examines the modification log generated by the file server process every two minutes. The indexing process responds to a file system modification event by choosing an appropriate transducer for the modified object. An appropriate transducer is selected by determination of the type of the object (e.g. C source file, object file, directory). If no special transducer is found a default transducer is used. The output of the transducer is fed to the Indexer module that inserts the computed attributes into the index. Indexing and retrieval are based upon Peter Weinberger's BTree package [74] and an adapted version of the `refer` [33] software to maintain the mappings between attributes and objects.

- The *mount daemon* is contacted to determine the root file handle of the underlying UNIX file system. The file server process exports its NFS service using the same root file handle on a distinct port number.

- The *kernel* implements a standard file system that is used to store the shared index. The file server process could be integrated into the kernel by a VFS based implementation [30] of an semantic file system. I implemented the prototype system using a user level NFS server to simplify development.

Instead of computing all of the virtual directories that are present in a path name, the implementation only computes a virtual directory if it is enumerated by a client `readdir` request or a `lookup` is performed on one of its entries. This optimization allows the SFS Server to postpone query processing in the hope that further attribute specifications will reduce the amount of work necessary for computation of the result set. This optimization is implemented as follows:

- The SFS Server responds to a `lookup` request on a virtual directory with a `lookup_not_found` fault to the Virtual Directory module. The Virtual Directory module checks to make sure that the virtual directory name is syntactically well formed according to the grammar in Section 3. If the name is well formed, the directory fault is immediately satisfied by calling the `create_dir` procedure in the SFS Server. This procedure creates a placeholder directory that is used to satisfy the client's original `lookup` request.

- The SFS Server responds to a `readdir` request on a virtual directory or a `lookup` on one of its entries with a `fill_directory` fault to the Virtual Directory module. The Virtual Directory module collects all of the attribute specifications in the virtual directory path name and passes them to the Query Processing module. The Query Processing module uses simple heuristics to reorder the processing of attributes to optimize query performance. The matching entries are then materialized in the placeholder directory by the Virtual Directory module that calls the `create_link` procedure in the SFS Server for each matching file or directory.

The transducers that are presently supported by the prototype semantic file system implementation include:

- A transducer that describes New York Times articles with `type:`, `priority:`, `date:`, `category:`, `subject:`, `title:`, `author:`, and `text:` attributes.

- A transducer that describes object files with `exports:` and `imports:` attributes for procedures and global variables.

- A transducer that describes C, Pascal, and Scheme source files with `exports:` and `imports:` attributes for procedures.

- A transducer that describes mail files with `from:`, `to:`, `subject:`, and `text:` attributes.

- A transducer that describes text files with `text:` attributes. The text file transducer is the default transducer for ASCII files.

In addition to the specialized attributes listed above, all files and directories are further described by `owner`, `group`, `dir`, `name`, and `ext` attributes.

At present, only publicly readable files are indexed, because indexing protected files would expose the contents of private files through the query system. Enhancements to the prototype implementation that could be explored further include: 1) support for multi-host queries using query routing, 2) an enhanced query language, 3) better support for file deletion and renaming, and 4) integration of views for entities smaller than files. The present implementation deals with deletions by keeping a table of deleted entities and removing them from the results of query processing. Entities are permanently removed from the database when a full reindexing of the system is performed. I have considered performing file and directory renames without reindexing the underlying files, but this is not possible in the current design because the user-programmable transducers are permitted to alter their behavior depending on the name of the file being indexed.

2.3 Experience and Performance

I ran some preliminary experiments using the semantic file system implementation to explore whether semantic file systems present a more effective storage abstraction than do traditional tree structured file systems for information sharing and command level programming. All of the experimental data I report are from my research group's file server using a semantic file system. The server is a Microvax-3 running UNIX version 4.3bsd. The server indexes all of its publicly readable files and directories.

To compact the indexes the prototype system reconstructs a full index of the file system contents every week. On 23 July 1991, full indexing of our user file system processed 68 MBytes in 7,771 files (Table 2.3).[1] Indexing the resulting 1 million attributes

[1]The 162 MBytes in publicly readable files that were not processed were in files for which transducers

Total file system size	326	MBytes
Amount publicly readable	230	MBytes
Amount with known transducer	68	MBytes
Number of distinct attributes	173,075	
Number of attributes indexed	1,042,832	

Type	Number of Files	KBytes
Object	871	8,503
Source	2,755	17,991
Text	1,871	20,638
Other	2,274	21,187
Total	7,771	68,319

Table 2.1: User File System Statistics for 23 July 1991

took 1 hour and 36 minutes (Table 2.2). This works out to an indexing rate of 712 KBytes/minute.

File system mutation operations trigger incremental indexing. In update tests simulating typical user editing and compiling, incremental indexing is normally completed in less than 5 minutes. In these tests, only 2 megabytes of modified file data were reindexed. Incremental indexing is slower than full indexing in the prototype system because the incremental indexer does not make good use of real memory for caching. The full indexer uses 10 megabytes of real memory for caching; the incremental indexer uses less than 1 megabyte.

The indexing operations of the prototype are I/O bound. The CPU is 60% idle during indexing. Measurements show that transducers generate approximately 30 disk transfers per second, thereby saturating the disk. Indexing the resulting attributes also saturates the disk. Although the transducers and the indexer use different disk drives, the transducer-indexer pipeline does not allow I/O operations to proceed in parallel on the two disks. Thus, the indexing throughput could be roughly doubled by improving

have not yet been written: executable files, PostScript files, DVI files, tar files, image data, etc.

[2]in parallel with Transduce

Part of index	Size in KBytes
Index Tables	6,621
Index Trees	3,398
Total	10,019

Phase	Time (hh:mm)
Directory Enumeration	0:07
Determine File Types	0:01
Transduce Directory	0:01
Transduce Object	0:08
Transduce Source	0:23
Transduce Text	0:23
Transduce Other	0:24
Build Index Tables[2]	1:22
Build Index Trees	0:06
Total	1:36

Table 2.2: User FS Indexing Statistics on 23 July 1991

the pipeline's structure.

I expect this indexing strategy to scale to larger file systems because indexing is limited by the update rate to a file system rather than its total storage capacity. Incremental processing of updates will require additional read bandwidth approximately equal to the write traffic that actually occurs. Past studies of Unix file system activity [53] indicate that update rates are low, and that most new data is deleted or overwritten quickly; thus, delaying slightly the processing of updates might reduce the additional bandwidth required by indexing.

To determine the increased latency of overlaid NFS operations introduced by interposing my SFS server between the client and the native file system, I used the `nhfsstone` benchmark [32] at low loads. The delays observed from an unmodified client machine were smaller than the variation in latencies of the native NFS operations. Preliminary measurements show that `lookup` operations are delayed by 2 ms on average, and operations that generate update notifications incur a larger delay.

Some anecdotal evidence supports the claim that a semantic file system is more effective than traditional file systems for information sharing:

- The typical response time for the first `ls` command on a virtual directory is approximately 2 seconds. This response time reflects a substantial time savings over linear search through the entire file system with existing tools. In addition, subsequent `ls` commands respond immediately with cached results.

  I also tested how the number of attributes in a virtual directory name altered the observed performance of the `ls` command on a virtual directory. Attributes were added one at a time to arrive at the final path name:

  ```
  /sfs/text:/virtual/
       text:/directory/
       text:/semantic/
       ext:/tex/
       owner:/gifford
  ```

  The two properties of a query that affect its response time are the number of attributes in the query and the number of objects in the result set. The effect of an increase in either of these factors is additional disk accesses. Figure 2-3 illustrates the interplay of these factors. Each point on the response time graph is the average of three experiments. In a separate experiment I measured an average response time of 5.4 seconds when the result set grew to 545 entities.

- My research group began to use the semantic file system as soon as it was operable for a variety of everyday tasks. Group members found the virtual directory interface to be easy to use. (We were immediately able to use the GNU Emacs directory editor DIRED [67] to submit queries and browse the results. No code modification was required.) Several group members reexamined their file protections in view of the ease with which other users could locate interesting files in the system.

- Users outside the research group have successfully used the query interface to locate information, including newspaper articles, in the file system.

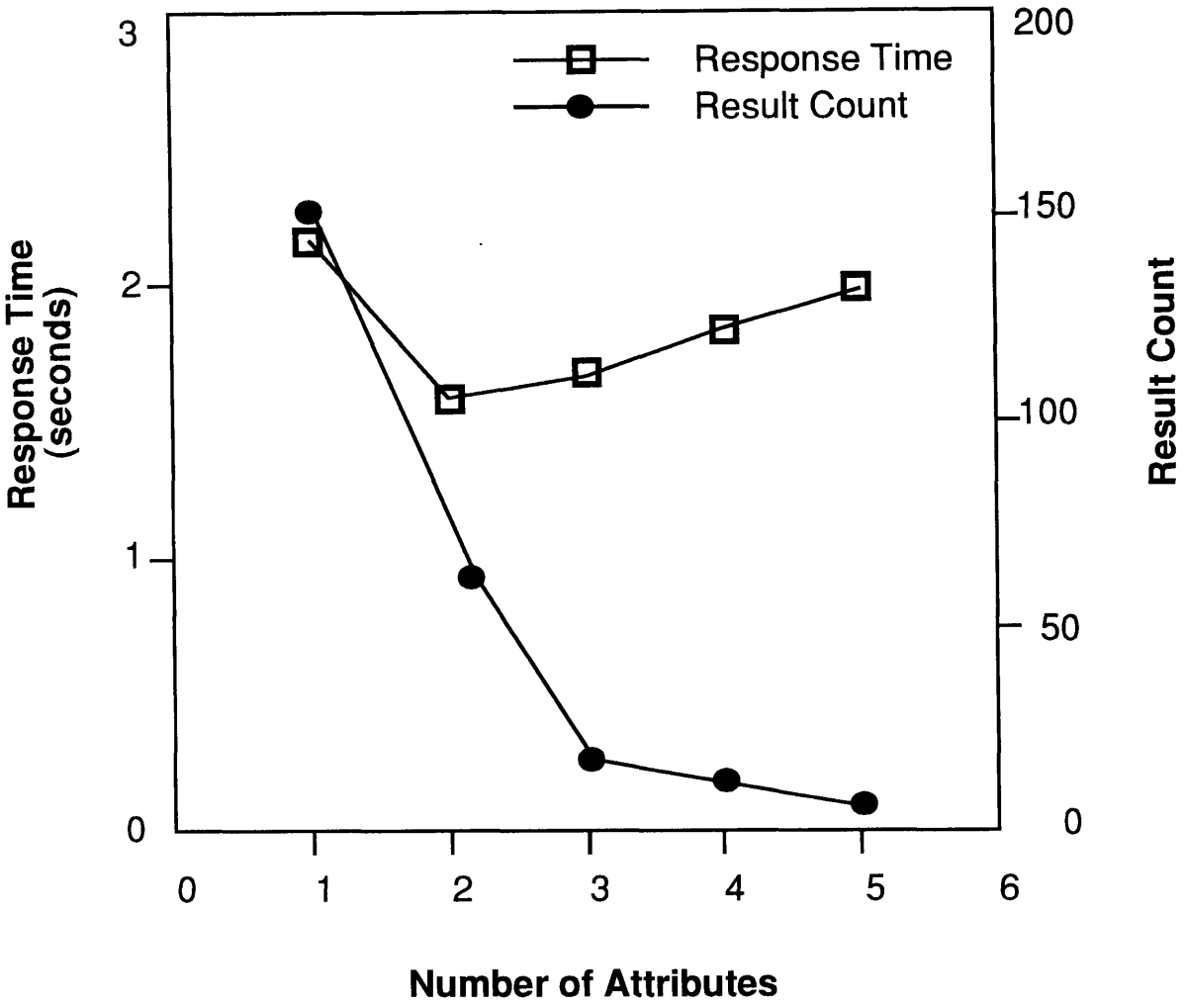

Figure 2-3: Plot of Number of Attributes vs. Response Time and Number of Results

- Users outside our research group have failed to find files for which no transducer had yet been installed.

The following anecdotal evidence supports the claim that a semantic file system is more effective than traditional file systems for command level programming:

- The UNIX shell pathname expansion facilities integrate well with virtual directories. For example, it is possible to query the file system for all `dvi` files owned by a particular user, and to print those whose names begin with a certain sequence of characters.

- Symbolic links have proven to be an effective way to describe file system views. The result of using such a symbolic link as a directory is a dynamically computed set of files.

Chapter 3

Log-Based Replicating Garbage Collection

The use of dynamic storage allocation has increased as programs have become larger and more complex. Greater use of object oriented and functional programming techniques further exacerbates this trend. These trends also make automatic management of dynamic storage or garbage collection increasingly necessary. Garbage collection simplifies the programmers task and increases the robustness and safety of programs that use it.

The traditional objections to garbage collection are primarily performance related. It has often been considered too expensive for use in practical applications. Recent studies by Zorn [78] of applications that make heavy use of dynamic storage suggest that in fact explicit storage management may be as costly as garbage collection. However, many garbage collectors interrupt the execution of the client application, causing pauses that are unacceptable to users and system designers.

Concurrent garbage collection addresses the problem of pause times by allowing the collector and the client application to execute in parallel or in an interleaved manner on a uniprocessor. This chapter presents the design and implementation of a log-based replicating garbage collection system that provides concurrent and compacting collection of a persistent heap. The first section introduces the basic idea of replicating garbage

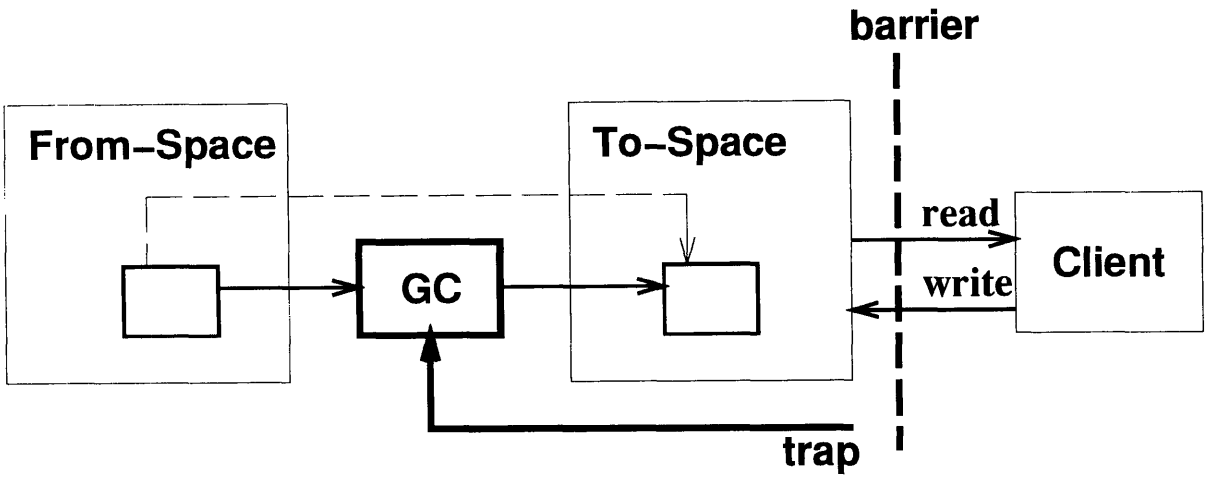

Figure 3-1: A Read Barrier Protecting Tospace

collection and compares it to the major alternative technique. Later sections describe the extension of replicating garbage collection to a persistent heap interface, its design, and a prototype implementation. The final two sections examine the performance of the prototype implementation and discuss a few possible extensions.

3.1 Replicating Garbage Collection

Concurrent garbage collectors permit the client to execute while the garbage collection is in progress. The operations of the client and the collector may be interleaved in any order, yet the effects of the garbage collector must not be observable by the client. In many previous concurrent garbage collection designs based on read-barrier methods the interactions between the client and the collector involved complex and expensive synchronization requirements. Replicating garbage collection requires that the collector replicate live objects without modifying the original objects. Interactions with the client are minimized, making this design attractive for use in a concurrent collector.

3.1.1 The Read-Barrier Method

The standard technique used by copying garbage collectors to copy an object destroys the original object by overwriting it with a forwarding pointer. Therefore, concurrent collectors using this technique must ensure that the client uses only the relocated copy of an object. I call this requirement the *to-space invariant.* Enforcing this to-space invariant typically requires a "read-barrier", as shown in figure 3-1. The implementation of read-barriers has consequently been the focus of much effort in incremental garbage collection work.

3.1.2 The Client Uses From-Space

The primary way in which replicating collection differs from the standard approach is that the copying of objects is performed non-destructively. Conceptually, whenever the collector replicates an object it stores a relocation record in a relocation map, as shown in Figure 3-2. In general the client may access the original object or the relocated objects and is oblivious to the existence or contents of the relocation map. In the implementation presented here the client accesses only the original object. I call this the *from-space invariant.*

3.1.3 Mutations are Logged

After the collector has replicated an object, the original object may be modified by the client. In this case, the same modification must also be made to the replica before the client can safely use the replica. Therefore, our algorithm requires the client to record all mutations in a "mutation log", as shown in Figure 3-2. The collector uses the log to ensure that all replicas are in a consistent state when the collection terminates. The collector does this by reading the log entries and applying the mutations to the replicas.

The cost of logging and of processing the log varies depending on the application and the logging technique. Mutation logging works best when mutations are infrequent or

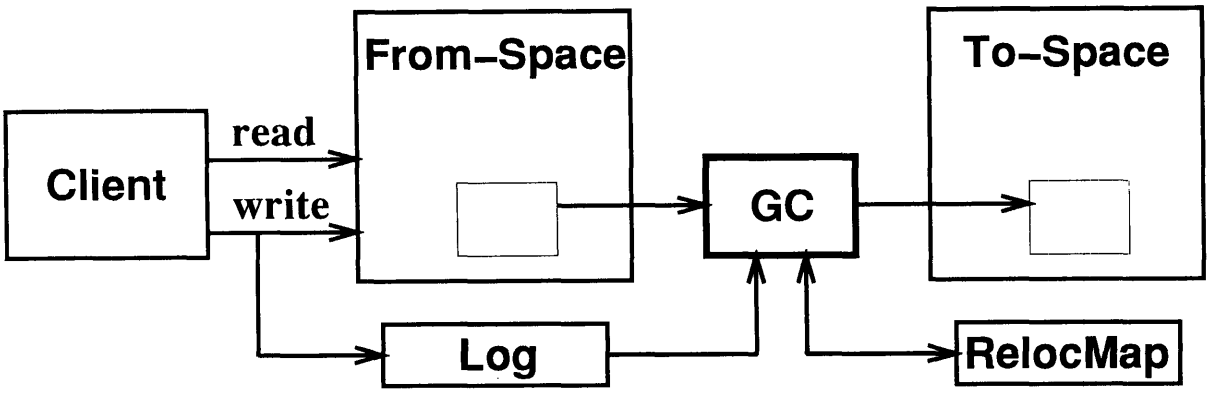

Figure 3-2: Replication and The Mutation Log

can be recorded without client cooperation. Mutation logging is also attractive whenever a log is already required for other reasons, such as in generational collectors, distributed applications, and transactional storage systems [42, 49].

3.1.4 The Collector Invariant

The invariant maintained by the collector is that the client can only access from-space objects and that all to-space replicas are up-to-date with respect to their original from-space objects unless a corresponding mutation is described in the mutation log.

3.1.5 The Completion Condition

While the collector executes, it endeavors to replicate all the objects that are accessible to the client. The collector creates replicas of the objects pointed to by the client's roots. The collector also scans replicas in to-space to find pointers to from-space objects and replace them with pointers to corresponding replicas in to-space.

The collector has completed a collection when the mutation log is empty, the client roots have been scanned, and all of the objects in to-space have been scanned. When these conditions have been met, the invariant ensures that all objects reachable from the roots have been replicated in to-space and are up-to-date. The replicas contain only to-space pointers because to-space has been scanned. When the collector has established

this completion condition, it halts the client, atomically verifies the completion condition, updates the client's roots to point at the corresponding to-space replicas, discards the from-space, and renames to-space as from-space.

3.1.6 Client Interactions

Although the garbage collector executes concurrently with the client, the from-space invariant ensures that there is no low-level interaction between the collector and client. The client executes machine instructions that read and write the objects that reside in from-space. The collector reads the objects in from-space and writes the objects in to-space. Conceptually, the relocation map shown in Figure 3-2 is used only by the collector.

The collector does interact with the client via the mutation log and the client's roots. The collector must occasionally obtain an up-to-date copy of the client's roots in order to continue building the to-space replica. Also, the collector reads the mutation log, which is being written by the client. These interactions may be asynchronous and do not require the client to be halted.

However, when the collector has established the completion condition, it must halt the client in order to atomically verify the completion condition and update the client's roots. After the roots have been updated, the client can resume execution. The duration of this pause in the client's execution depends on the synchronization delay due to interacting with the client thread and also on the size of the root set. In a generational collector the root set may include the set of cross-generational pointers that point from older objects to newer objects.

3.1.7 Optimization Opportunities

The from-space invariant used by this algorithm is very weak, in the sense that the collector never needs to work on any particular task in order to allow the application to

execute. The collector only needs to replicate all the live data soon enough to terminate and reuse the memory in from-space before the application runs out of memory.

In the algorithm of Appel, Ellis, and Li[3], the application may frequently be blocked waiting for the collector to copy the objects that it must use. I believe that the flexibility of this invariant offers potentially important optimization opportunities to any replicating-style implementation. For example, the collector can copy objects in essentially any desired order.

This freedom in copying order could be used to increase locality of reference or to change the representation of objects stored in a cache[50]. Another way that copying order freedom can be exploited is by concentrating early replication work on objects reachable from particular roots. Particular roots may be more likely to change than others, so copying them later could reduce the amount of latent garbage copied by the collector.

Also, if no mutable objects have been replicated then the collector need not apply mutations to replicas. The collector could choose to concentrate early replication effort on only immutable objects, and thereby delay the need to process the log until the last possible moment. The actual copying of an object can be delayed until the object is scanned using an optimization suggested by Ellis[19]. The collector could replicate mutable objects into a segregated portion of the to-space, and delay copying and scanning mutable objects as long as possible. Mutation log entries created before the first mutable object was actually copied could be discarded.

3.2 A Transactional Heap Interface

The persistent heap interface provides basic heap operations, transaction operations, and two distinguished roots. The complete interface is shown in Figure 3-3. The basic heap operations are read, write and allocate. The transaction operations are commit and abort.

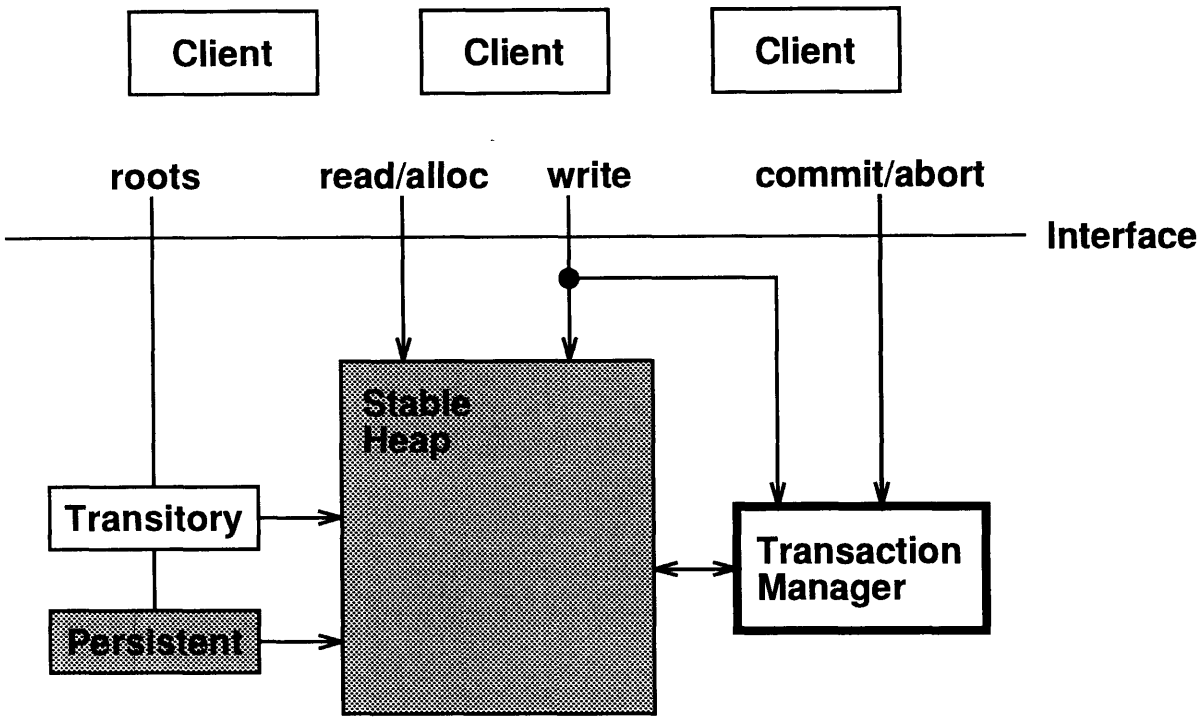

Figure 3-3: The Transactional Heap Interface

The transitory and persistent roots are distinguished and available to the clients. No deallocation operation is exposed to the clients. Instead, objects that the clients can access by dereferencing pointers starting from either root are considered live and will be preserved by the system. A garbage collector will identify unreachable objects and recycle their storage.

The heap interface provides *orthogonal persistence* [4]: objects that are reachable via the persistent root are guaranteed to survive system failures. In Figure 3-3, the stability of the persistent root and the stable heap to which it points are indicated by their gray background. In contrast, objects reachable only via the transitory root are not available after a failure; the transitory root is reset when the system recovers from a failure. Clients can use the transitory root to maintain access to temporary objects without requiring that the system ensure the persistence of those objects.

This interface also includes the ability for multiple clients to perform transactions on the heap by using the commit and abort operations. The transaction manager is respon-

sible for tracking modifications to objects and implementing the transaction semantics implied by commit and abort. Support for multiple clients, multithreaded clients, and nested transactions can be provided by appropriate choice of the transaction manager.

The exact transaction semantics supported by the transaction manager are not my primary concern here. In considering the design of the system, I will assume that the complexities of multiple clients and multiple transactions are completely hidden from the rest of the system by the transaction manager. I describe the system as if it contains only a single client that performs a linear sequence of top-level transactions. Every modification to an object is part of a transaction and each transaction can be either committed or aborted. The heap must be restored to the most recently committed state if a system failure occurs.

3.3 Garbage Collection Design

The heap design shows that the transactional heap interface can be implemented efficiently. The interface specifies a high standard of programming safety and data stability. Programming safety is provided by using garbage collection and orthogonal persistence. Data stability is ensured through the use of transactions and stable storage. Good performance comes from caching stable data in volatile memory and from the use of concurrent replicating collection. Using an extra processor to perform concurrent replicating collection improves throughput and provides the client with low-latency access to the heap.

I will present the design of the storage manager as a series of three refinements to a basic design. Each refinement improves either the functionality or the performance of the basic design:

- *Basic Design: Replicating Collection of Stable Heaps.* The basic design uses a replicating collector on stable spaces. Clients operate on *stable from-space*, and the collector concurrently copies from-space into *stable to-space*. In this design each write is individually committed and the client must access stable storage for every

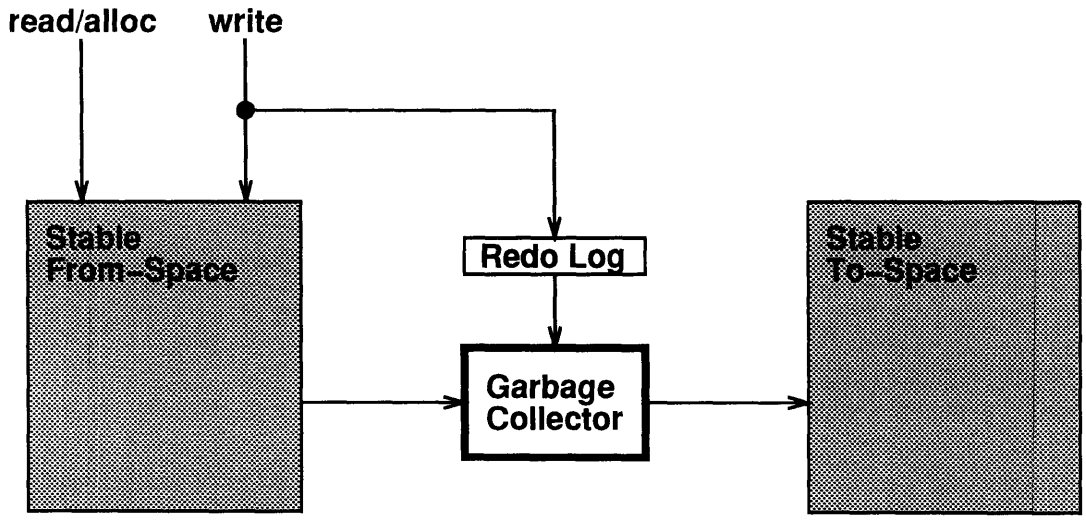

Figure 3-4: Replicating Collection of a Stable Heap

read and write operation.

- *Refinement 1: Transactions Group Updates.* The first refinement adds transactions to the basic design. Transactions allow the client to perform a group of modifications atomically. The log required to support transactions also serves as the log used by replicating garbage collection.

- *Refinement 2: Volatile Images Improve Performance.* The second refinement adds volatile main-memory images of from-space and to-space to improve the performance of the client and the collector. Committed from-space operations must be recovered upon failure. Thus it is necessary for the transaction manager to ensure that all committed operations are recorded in stable from-space.

- *Refinement 3: Transitory Heaps for Temporary Data.* The third and final refinement adds a *transitory heap.* All objects are initially allocated in the transitory heap, and are automatically promoted to the persistent heap when they are made reachable from the persistent root. The transitory heap is garbage collected with respect to the *transitory root* and is not recovered after a failure.

3.3.1 Replicating Collection of Stable Heaps

As shown in Figure 3-4, using stable storage and replicating collection for the persistent heap provides an interface that lacks only transaction support. In this design each write operation is individually committed and becomes permanent as soon as it is performed. Good performance can be achieved by using battery-protected random access memory to implement high speed stable storage. While the client operates on the heap, a replicating garbage collector concurrently builds a compact replica of it.

Replicating garbage collection [40] is a new technique for building incremental and concurrent garbage collectors. The key idea of replicating garbage collection is to copy objects non-destructively. As in any copying garbage collection algorithm, the collector copies the reachable objects from the from-space to the to-space. However, most copying collectors destroy the original object when it is copied because the contents of the object are overwritten by relocation information containing its new address.

In contrast, a replicating collector avoids destroying the contents of the original object. This can be accomplished by storing the relocation information about the object in a reserved portion of the object or elsewhere. The effects of replication are unobservable by the client, so it continues to operate on the original objects in from-space. This differs from other concurrent compacting collection algorithms, which require the client to operate on objects in to-space [5, 3].

Because the client and the collector execute concurrently, the replicas become inconsistent when the client modifies the original objects. To solve this problem, the client records all write operations in a redo log, shown in Figure 3-4. The collector processes the log to ensure that all replicas are consistent. When to-space contains an exact replica of the entire object graph, the collector *flips*, updating the client's roots to refer to the replicas and exchanging the roles of to-space and from-space. While the collector performs the flip action, the client and transaction manager are halted.

After a failure the client can be immediately restarted on stable from-space without any recovery processing. The stable nature of from-space guarantees it will survive

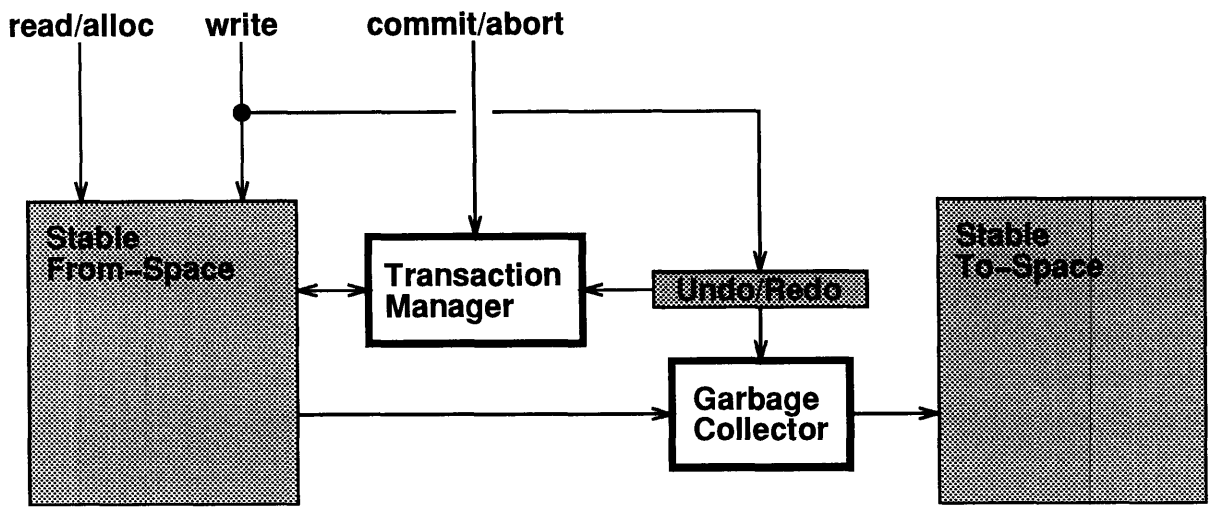

Figure 3-5: A Transactional Heap

failures. After a crash all that is necessary is to locate the stable from-space in memory. Because from-space and to-space exchange roles when the garbage collector flips, the garbage collector must store a from-space identifier to indicate which of the two stable spaces is the current from-space. The flip occurs when this identifier is updated atomically.

The garbage collector can be resumed after a failure if sufficient state is recorded in stable storage. If the collector is to be recoverable, the redo log and the relocation information containing the new address of each replicated object must be in stable storage. These items will collectively be called the *garbage collector state.* Alternatively the garbage collector state can be kept in volatile storage and the garbage collector can be restarted with an empty to-space upon failure. If failures are infrequent this option will result in better performance.

3.3.2 Transactions Group Updates

Figure 3-5 shows how transactions can be added to the basic design. In this model, each modification is part of a transaction and transactions can be independently committed or aborted. All modifications are applied directly to from-space and an *undo log* describing

uncommitted modifications is maintained. If a transaction aborts, all of its modifications are undone by using the undo log. If a transaction commits, then its modifications are atomically removed from the undo log. An important advantage of using replicating collection in a transactional setting is that the undo log necessary to support transactions and the redo log used by the collector can use a shared data structure.

After a failure the client can be resumed on stable from-space once the undo log is applied. Undoing the modifications in the undo log ensures that all uncommitted updates are erased before the client resumes execution. The undo log must be stored in stable storage so that it will survive failures and be available for recovery processing after a failure. After recovery the garbage collector can be resumed if the garbage collector state has been preserved in stable storage.

3.3.3 Volatile Images Improve Performance

When slow stable storage media are used, forcing all operations to access stable storage will not provide acceptable performance. Figure 3-6 shows how this problem can be addressed by caching images of the stable heaps in volatile memory. In this design the client reads, writes, and allocates in a volatile image of from-space. The garbage collector reads the volatile from-space image and writes the volatile to-space image.

When a transaction commits, its updates to volatile from-space must be made stable. Upon commit the transaction manager uses the redo log to identify portions of the volatile from-space image that have been updated and it propgates these modifications to stable from-space. The transaction manager also writes any newly allocated objects to stable from-space. Applying these updates to stable from-space must be performed atomically in order to ensure that the stable from-space is in a consistent state after a failure.

The garbage collector directly writes the volatile to-space image and a background process copies the volatile to-space image onto the slow stable storage media. Thus a slow stable storage medium can be used for to-space without slowing the garbage collector. A flip may occur when all live objects in volatile from-space have been copied, the redo log

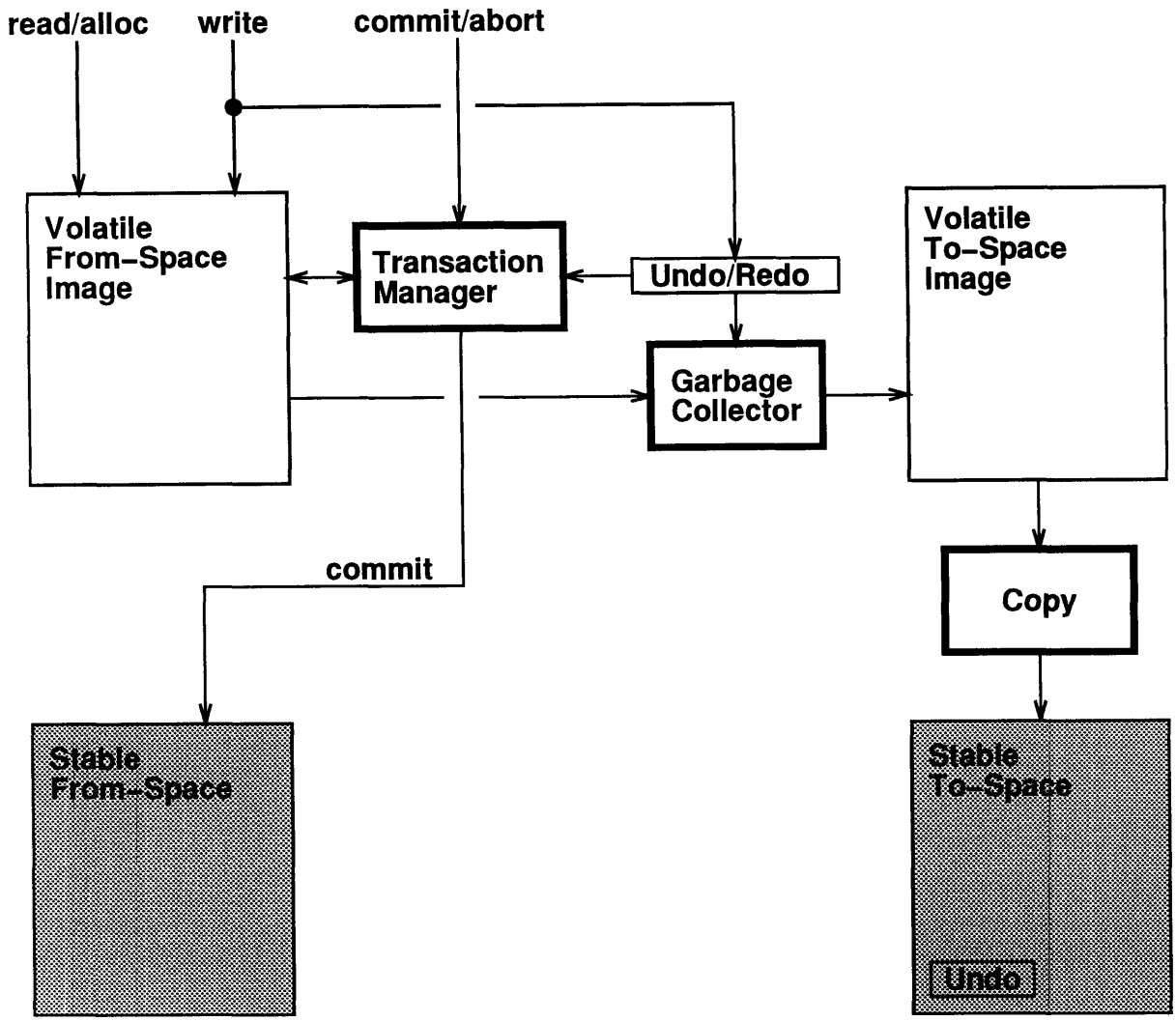

Figure 3-6: Using Volatile Images

is empty, and volatile to-space has been completely copied to stable to-space. The flip causes both stable and volatile from-space to be replaced by their to-space counterparts.

However, the stable to-space may contain uncommitted data because it is an exact copy of volatile to-space, which is a replica of volatile from-space. To prevent this uncommitted data from being used after a crash, the undo log is also written to stable storage as part of the stable to-space. This allows any uncommitted transaction to be rolled back during recovery.

After a failure the client can use the volatile from-space image once it is recovered

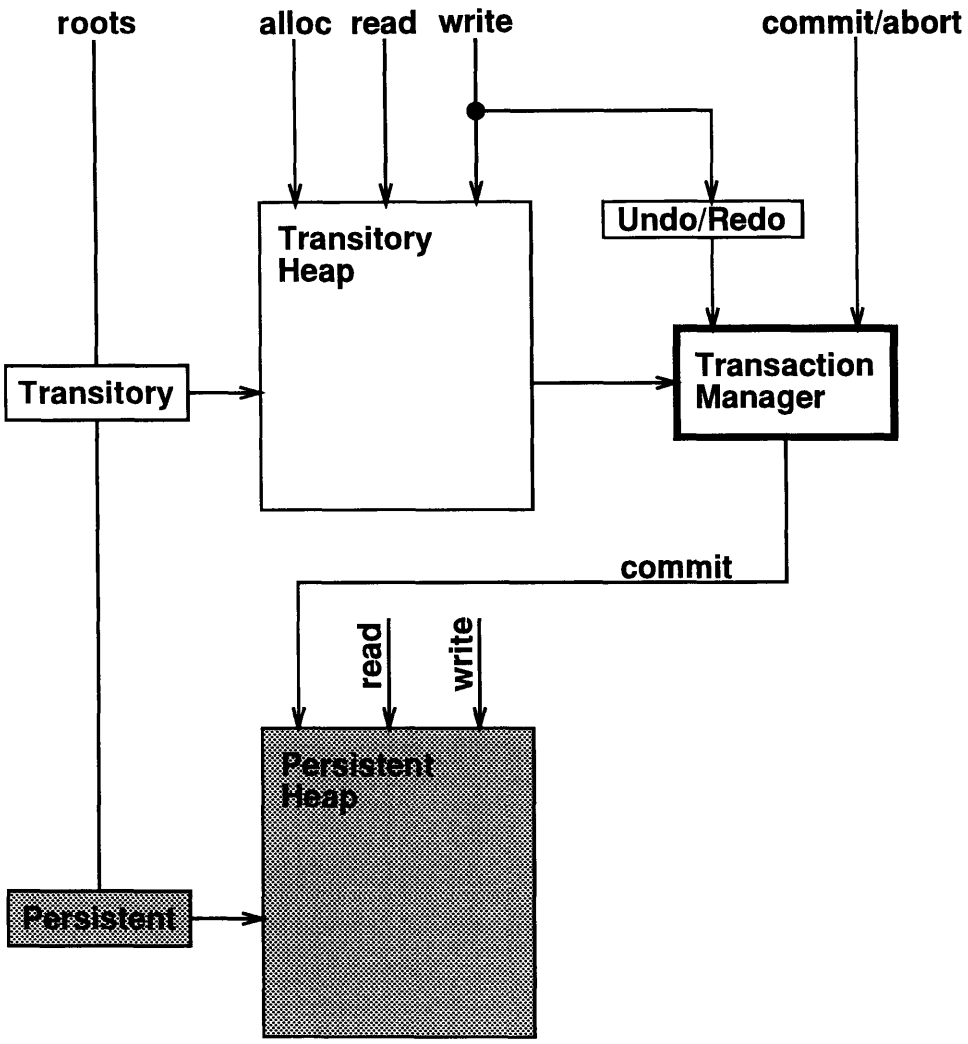

Figure 3-7: Using a Transitory Heap

from stable from-space and the undo log is applied. After recovery the garbage collector can be resumed if its state has been preserved in stable storage. Otherwise it is restarted with an empty volatile to-space.

3.3.4 Transitory Heaps for Temporary Data

Figure 3-7 shows how a transitory heap may be added to the previous design. The persistent heap, shown in gray, represents the volatile and stable versions of from-space and to-space from the previous design. This figure emphasizes that the role of the

transitory heap is to store objects that are not reachable from the persistent root. The transitory heap has its own root and is stored in volatile memory. All objects are initially allocated in the transitory heap.

Upon commit the transaction manager uses the log to detect objects in the transitory heap that have become persistent. The log is used to locate modified objects in the persistent heap that contain pointers to objects in the transitory heap. These objects are newly persistent because they have become reachable via the persistent root.

The modified objects are used as the roots of a stop-and-copy collection that promotes the newly persistent objects into the persistent heap. The promoted objects must be written to stable storage along with any persistent objects modified by the transaction.

It is also possible for persistent objects to become transitory. This happens when objects in the persistent heap are reachable from the transitory root but have become unreachable from the persistent root. Such objects are live but not persistent. When these objects are discovered they may be moved back into the transitory heap or left in the persistent heap. Leaving these objects in the persistent heap simplifies the implementation. After a failure these objects will be unreachable and will be reclaimed by the next collection.

After a failure the client is restarted on the persistent heap using the algorithm outlined in the previous section. During recovery the transitory heap is initialized to be empty.

3.4 A Prototype Implementation

The prototype implementation provides concurrent compacting garbage collection for Standard ML of New Jersey. A persistent heap and a transaction manager were added to the runtime system to support persistence. I re-implemented the garbage collector as a concurrent thread using replicating collection. The purpose of the prototype implementation was to test the feasibility of this design.

Standard ML of New Jersey (SML/NJ) is an implementation of a type-safe programming language that includes an optimizing compiler, a runtime system, and a generational garbage collector [2]. The SML/NJ source code is freely available and is easy to modify for experimental purposes.

I chose to test the replicating garbage collection algorithm in the SML/NJ environment primarily for reasons of convenience. Previous work on persistence by Scott Nettles [42] and my work with Scott Nettles on replicating garbage collection [40, 41, 51] provided several of the components needed for the prototype.

The SML/NJ runtime system is similar to most language implementations with copying garbage collection. The results should therefore apply to languages such as Modula-3, Java, Lisp, Scheme, Smalltalk, Telescript, and, if suitably modified to enable copying garbage collection, C++.

Implementation Overview

An implementation of the basic design and its three refinements requires a replicating collector, an undo/redo log, stable storage, and a transaction manager. Here is a summary of how each of these components is implemented:

- The replicating garbage collector is implemented as described in Nettles and O'Toole [40], except that it runs as a separate thread of control so that it provides concurrent collection as reported by O'Toole and Nettles [52]. The collector thread occasionally interrupts the client to obtain more up-to-date roots and log entries. It also stops the client in order to perform a flip.

- The undo/redo log is written by the client. The SML/NJ compiler was modified to emit instructions in the client code that generate appropriate log entries for every write operation. The original SML/NJ implementation included a simpler log to support generational garbage collection. The log was expanded to include undo information about every write operation in order to support transaction operations

and replicating collection.

- The stable heaps are maintained on disk and their volatile images are maintained in main memory. Recoverable Virtual Memory is used to manage the stable heaps on disk. RVM allows the transaction manager and the garbage collector to apply changes atomically to the stable heap by writing the changes into a stable log. After a failure, the RVM log is used to recover the stable heap.

- The transaction manager in the current implementation performs transaction commit by reading the log, moving objects from the transitory heap to the volatile heap image, and logging the resulting changes to the stable heap via RVM. The transitory heap is merely the simple generational heap present in the original SML/NJ implementation.

The next few sections discuss some choices faced when implementing the prototype. I then describe the structure of the persistent heap and review the threads of control used in the implementation. Finally, I present the step by step procedures used to implement the commit, collection, flip and recovery operations.

3.4.1 The Transitory Heap

The transitory heap contains only temporary objects that will be discarded when a failure occurs. It consists of the two generational heaps present in the original SML/NJ implementation. The details of the these transitory heaps are mostly irrelevant to the persistent heap implementation.

However, when constructing the prototype I faced several implementation choices that involved the transitory heap. There are several situations in which pointers that cross between the transitory and the persistent heap may have to be adjusted when objects are moved.

For example, when newly persistent objects are promoted into the persistent heap, any other temporary objects that contain pointers to them must be updated. Similar

adjustments may be required to either the transitory or the persistent heap when the other heap is being flipped by the garbage collector.

In these situations, the implementation could use any one of at least four different solutions:

1. Scan the heap that contains the pointers requiring adjustment.
2. Garbage collect the heap that contains the pointers requiring adjustment.
3. Somehow track the locations of the pointers that require adjustment so that they can be updated more quickly.
4. Use replication to move the objects, but refrain from updating any pointers to the objects until some later opportunity such as a flip.

In this implementation, pointers from the persistent heap to the transitory heap are always the result of uncommitted write operations. They can be easily located via the transaction log. Therefore, I use the third solution to update these pointers when the transitory heap is garbage collected. To deal with pointers in the other direction, I implemented both of the first two methods: doing a scan and doing a garbage collection. Commit processing can be expensive in this implementation when there is a large transitory heap.

3.4.2 Stable Storage

My design assumes the ability to make multiple updates to the stable heap atomically. The implementation performs atomic updates using the Recoverable Virtual Memory system described by Satyanarayanan, et al. [63] RVM provides simple non-nested transactions on byte arrays and uses a disk-based log for efficiency. The implementation does not exploit the rollback features of RVM at all; RVM provides only disk logging and recovery services.

RVM establishes a one-to-one correspondence between a file and a portion of virtual memory. Separate files are used to store stable from-space and stable to-space. These files also contain starting and ending addresses of the spaces and the sequence number that serves as the stable from-space identifier.

RVM defines a simple interface that allows changes to the volatile heap to be atomically propagated to the stable heap. During commit the transaction manager begins an RVM transaction, informs RVM of the location of each modification to the volatile heap as well as of any newly persistent data, and then ends the transaction. RVM guarantees that this set of operations will be atomic.

When a flip occurs the from-space identifier must be updated to indicate which of the stable spaces is the stable from-space. This update does not need to be made stable until the next client transaction commit after the flip. When the client first commits a transaction, that transaction will be applied to the new stable from-space. The collector writes the from-space identifier via RVM, but using a "no-flush" transaction that does not require a synchronous disk write. Thus, although the client is paused while the garbage collector finalizes the flip and updates the roots, no synchronous disk write occurs during this interruption.

3.4.3 Writing Stable To-space

Volatile to-space is written to stable to-space by an asynchronous process. No writes that synchronize with the client are required. Volatile to-space can be written directly to the file containing stable to-space or indirectly via RVM.

During the garbage collection, large contiguous regions of new data are created in volatile to-space. These regions can be written directly to disk efficiently. Logging them through RVM is less efficient because RVM first writes them into its log and then later writes them again to the data file.

However, the garbage collector also performs small random updates on volatile to-space when it uses the redo log to update inconsistent to-space replicas. RVM is ideal for

applying these changes to stable to-space, because its use of logging converts the small random writes into a single efficient log write. Writing these changes to disk without using a log is of course more expensive.

I tried writing stable to-space using several different methods. It turned out that using RVM for all of the writes put so much data into the RVM log that the client transaction flow was substantially impeded. However, when I wrote stable to-space directly, it was difficult to schedule the flip to avoid blocking while waiting for the last few disk writes to complete.

Now the implementation uses a mixed strategy in which the large contiguous writes are done directly to the file containing stable to-space and the small random writes are done using RVM.

3.4.4 The Persistent Heap

The persistent heap is maintained both in virtual memory and on stable storage through the cooperation of the transaction manager, the garbage collector and RVM. Figure 3-8 shows the primary data structures used by the runtime system to maintain and garbage collect the persistent heap:

- **undo/redo log** — This log is maintained by the client and is used by both the transaction manager and the collector. When the stable heap flips, the log is written to stable storage to enable rollback of uncommitted transactions.
- **volatile from-space** — The client accesses all persistent data through volatile from-space. When a transaction commits, all newly persistent objects are copied from the transitory heap into volatile from-space.
- **volatile to-space** — The garbage collector copies the live objects in volatile from-space to volatile to-space.
- **stable from-space** — The stable representation of from-space is stored on disk. Upon commit the newly persistent objects and the new values of any modified

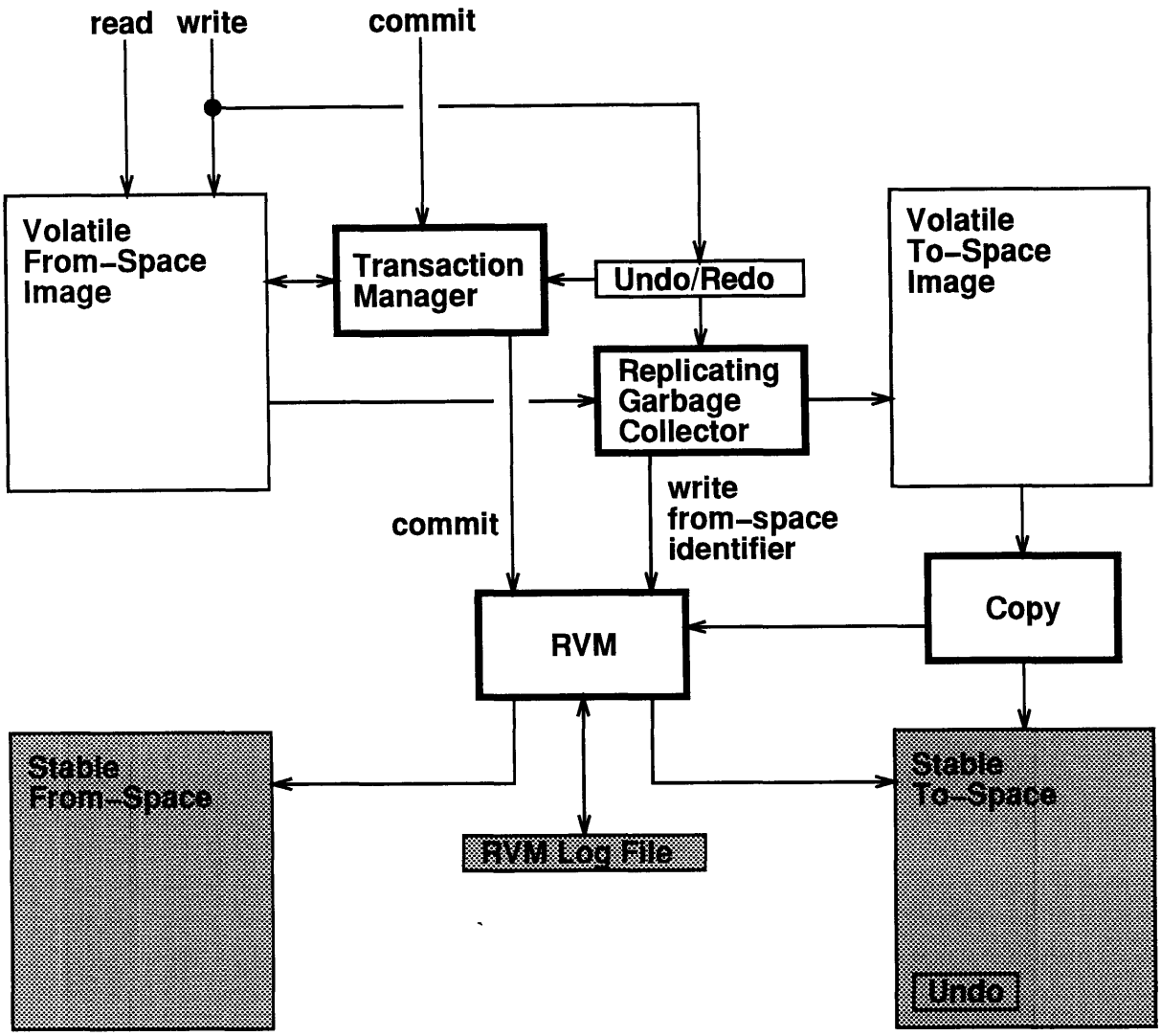

Figure 3-8: The Persistent Heap

locations in volatile from-space are written to stable from-space by the transaction manager using RVM.

- **stable to-space** — The stable representation of to-space is stored on disk. It is a copy of volatile to-space and must be written to disk before a flip.

- **the RVM log** — The stable log is maintained by RVM on disk. The RVM log contains records that describe modifications to stable from-space and stable to-space. RVM updates this log atomically and uses it to recover the contents of the

stable spaces after a failure.

3.4.5 Threads of Control

The prototype contains three threads of control:

- The **Client** thread executes the application program and periodically commits transactions by acting as the Transaction Manager. The Client thread is responsible for writing the undo/redo log. The Client thread also traps into the runtime system when it must synchronize with the garbage collector and perform a flip.
- The **Collector** thread performs replicating garbage collection, copying reachable objects from the volatile from-space into the volatile to-space. When the Collector thread has constructed a complete replica of the volatile from-space, it signals the Client thread that a flip should take place.
- The **Copy** thread writes the contents of volatile to-space onto the stable to-space, either directly to the disk or via the RVM log manager. It performs I/O operations on behalf of the garbage collector so that the Collector thread need not block on I/O.

3.4.6 Operations on the Persistent Heap

Now I present the steps needed to perform the key operations of the design.

Transaction Commit

When the Client thread commits a transaction, it acts as the Transaction Manager and performs the following operations:

1. Scan the undo/redo log to locate references that created pointers from the volatile from-space to transitory data. These references are used to identify the roots of the newly persistent data.

2. Promote all newly persistent objects into volatile from-space, using a standard stop-and-copy garbage collection algorithm. This collection uses the locations identified in step 1 as roots.

3. Atomically update the stable from-space by logging and committing an RVM transaction containing all modifications and additions to volatile from-space.

4. Update all of the transitory heap data to point to the newly promoted objects in volatile from-space. This is done either by scanning the transitory heap for pointers to promoted objects or by garbage collecting the transitory heap.

Persistent Garbage Collection

When a commit adds enough new data to the volatile from-space to exceed a predetermined threshold, a garbage collection of the persistent heap is initiated. At this point the Collector thread performs the following steps asynchronously with respect to the Client thread:

1. Copy all objects in volatile from-space that are reachable from the persistent root into volatile to-space. This preserves the persistent data.

2. Copy all objects in volatile from-space that are reachable from the transitory heap into volatile to-space. This preserves the objects that are live and have become transitory.

3. Scan the undo/redo log, and update replicas of modified objects in the volatile to-space.

The undo/redo log is reset to empty after its entries have been processed by both the transaction manager and the garbage collector. The transaction manager and the garbage collector maintain separate pointers into the shared log to keep track of what prefix of the log they have already processed.

When a collection is in progress the Copy thread asynchronously copies the volatile to-space image into stable to-space. The Copy thread occasionally synchronizes with the Collector thread to obtain information about what portions of volatile to-space have been modified.

Persistent GC Flip

A flip is attempted after the Collector thread has successfully replicated volatile from-space in volatile to-space and the Copy thread has copied it into stable to-space. Then the Collector thread stops the Client thread and performs any remaining collection work required by recent client operations. Finally, the Collector thread performs the following steps before resuming the client:

1. Update the client's roots to point to volatile to-space.
2. Update all pointers in the transitory heap that point to volatile from-space to point to volatile to-space.
3. Write the stable from-space identifier via RVM to indicate that stable to-space is now stable from-space.

The stable replicating collector must ensure that the undo/redo log is preserved in stable storage at the time of a flip. In the prototype, this is accomplished very simply because the transaction manager stores the log in the volatile from-space in the form of a linked list of log records. Therefore, it is preserved in volatile to-space by the Collector thread and written to stable storage by the Copy thread.

The flip has occurred in volatile memory when these steps are complete. The Collector thread does not write the stable from-space identifier synchronously. Instead, it uses a no-flush transaction that will update the from-space identifier before any subsequent client transactions.

Crash Recovery

When a crash occurs and the system restarts the following steps are performed for recovery:

1. RVM applies pending committed log records from its stable log to bring the stable from-space and stable to-space files to their most recently committed state.

2. The garbage collector examines the stable from-space identifier stored in the stable heaps to determine which heap is the stable from-space.

3. The transaction manager uses the undo log stored in stable from-space to roll back the uncommitted operations of any partially complete transactions. These changes are made atomically using RVM.

The Client thread can now be restarted using the recovered volatile from-space and an empty transitory heap. The collector begins with an empty volatile to-space. It is not necessary for RVM to recover the contents of stable to-space in this implementation, but RVM does not provide a way to eliminate recovery processing for stable to-space.

3.4.7 Comparison with Read-Barrier Method

In to-space techniques, the flip takes place at the beginning of the collection. Starting at the flip, the client uses objects that are in to-space and so client operations are reflected in the transaction log using to-space values. The log also contains from-space pointers due to transaction activity that occurred prior to the flip. Therefore, the recovery process must be able to reconstruct the relationship between from-space and to-space objects. This relationship depends on the exact sequence of copy operations performed by the garbage collector. In practice, this means that essentially the entire garbage collection process must be recoverable.

There are two primary disadvantages to making the collection recoverable: complexity and cost. Added complexity arises because each step of the collection must be recoverable,

thereby greatly increasing the interaction of the collection algorithm with the logging and recovery algorithms. Both Kolodner and Detlefs explain how each step of the collection algorithm is logged and recovered. These arguments are quite detailed and complicated. Added cost arises because each step of the collection adds to the logging burden of the system. Detlefs's design attempts to minimize the amount of log traffic but at the cost of introducing some synchronous writes. Kolodner requires no synchronous writes but has greater log traffic.

The replicating gc design avoids these problems because there is never any need for the recovery process to deal with to-space values, although the implementation I describe here does incur some recovery processing costs for to-space because RVM provides no convenient way to bypass these recovery operations. The client only uses from-space objects and only the flip need be coordinated with the transaction manager. A replicating collector can be made recoverable at the cost of added log traffic, but for many applications this may not be necessary.

3.5 Performance

I designed and ran a series of experiments to test whether the replicating collector implementation:

- significantly reduces the duration of collector pauses in comparison with the stop-and-copy collector.
- increases transaction throughput by reducing storage management overhead.

The experiments compare the implementation described in the previous section to the same collector operating in a stop-and-copy manner on the persistent heap. I did not compare this implementation directly to other concurrent collector implementations because they do not support garbage collection of stable heaps.

The experimental results demonstrate that replicating collection interferes with the client less than stop-and-copy collection. For the replicating collector, the longest pauses

are a few hundred milliseconds, the same general magnitude as commits. For heap sizes in the megabyte range, the pause times achieved by the new technique are a factor of ten shorter than stop-and-copy collection. For larger heaps, such as those found in a production object database, the difference would be even greater. Improvements in transaction throughput were achieved because concurrent garbage collection resulted in an overlapping of garbage-collection-related I/O with client computation.

I also measured the commit performance of the system. Commit performance depends mostly on the choice of persistence model, the transaction profile, and the performance of the underlying log manager. I measured this aspect of the system to better understand the prototype's limitations, and found that the commit performance can be quite good for small transactions.

3.5.1 Benchmarks

I used three benchmarks to test the system. Each was chosen to measure and stress different aspects of the system. Two of the benchmarks performed a significant number of garbage collections, while the third was used to measure transaction throughput. Although I did not measure recovery performance, I did crash and recover each benchmark to verify that it was recoverable.

- The *Compiler* benchmark is Standard ML of New Jersey compiling a portion of the SML/NJ implementation. I modified the compiler to store all of its data in the persistent heap and to commit its state every time a module (file) was compiled, modeling the behavior of a persistent programming environment. This 100,000 line program is compute-intensive and contains long-running transactions.

- The *TP-OO1-V* benchmark is a variant of the OO1 Engineering Database benchmark described by Cattell [9]. The benchmark models an engineering application using a database of parts, performing traversals of the database, adding parts, etc. I implemented this algorithm in order to have a representative object-oriented

database application. However, the OO1 benchmark, as specified, does not require garbage collection, so I added deletion operations to make it a more realistic application for this system.

- The *TPC-B* benchmark performs a large number of bank teller operations that perform transfers among various bank accounts. This benchmark is a slightly non-standard implementation in Standard ML of the TPC-B benchmark from Gray [14].

3.5.2 Experimental Setup

All benchmarks were executed on a Silicon Graphics 4D/340 equipped with 256 megabytes of physical memory. The clock resolution on this system is 1 millisecond. The machine contains four MIPS R3000 processors clocked at 33 megahertz. Each processor has a 64 kilobyte instruction cache, a 64 kilobyte primary data cache, and a 256 kilobyte secondary data cache. The secondary data caches are kept consistent via a shared memory bus-watching protocol and there is a five-word deep store buffer between the primary and the secondary caches. With this configuration, the collector can copy between 1 and 2 megabytes per second. (Note that the cache performance of a garbage-collected programming language has been analyzed by Reinhold [58].)

All benchmarks were executed using a *transactional process* model. Each time the application requests a commit, its entire process state is committed to the persistent heap, including processor register contents. I ran the benchmarks using this model because it generally minimizes the size of the transitory heaps and maximizes the workload on the stable heap. The transactional process model itself is implemented in SML using the persistence facilities described here.

3.5.3 Pause Times

The most important property of the concurrent replicating collector is that the pause times it imposes on the client are short and bounded. Very large stable heaps imply

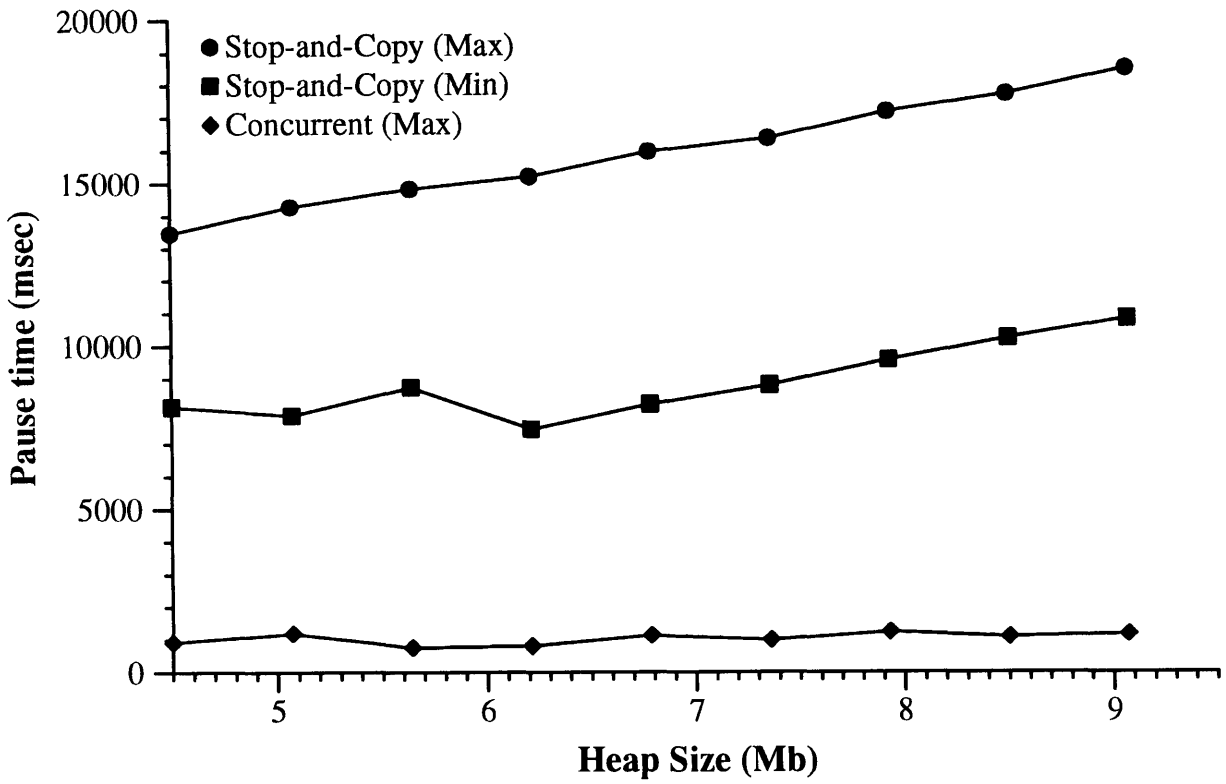

Figure 3-9: OO1 Collector Pause Times

very large stop-and-copy garbage collection pauses. In the sections that follow, I present measurements showing that the stable replicating collector pause times are largely independent of heap size. I also compare the pause times caused by the concurrent and stop-and-copy collectors and to those due to transaction commit operations.

Heap Size Dependence

I ran the OO1 Engineering Database benchmark in Standard ML with various amounts of live data in the heap. This test roughly shows the dependence of collector pause duration on heap size. Figure 3-9 shows the maximum pause times caused by collection of the persistent heap plotted with respect to heap size. The plot also includes the minimum pause times for the stop-and-copy collector. The minimum pause times for the concurrent collector are too small to measure.

Even for the modest heap sizes used here, the pauses created by the stop-and-copy

collector are unacceptably long. As expected, pause times increase with increasing heap size. In contrast, the maximum pause created by the concurrent collector is approximately 1 second and is brief enough not to cause a major disruption. Although not shown in this figure, typical concurrent collector pauses for this benchmark were less than 300 milliseconds. These results show that pause times are independent of heap size for the concurrent collector.

I investigated the cause of the longer pauses and believe that they are preventable. Examining the worst pauses in an earlier implementation convinced me that stable to-space should be written using a combination of direct disk writes and RVM. That change produced a five fold reduction in the maximum pause times and eliminated almost all pauses in the 100 to 1000 millisecond range. Further tuning should be able to reduce maximum pause times to 100 milliseconds or below.

Pause Time Distributions

To explore how the frequency and duration of collection pauses compared to commit pauses, I ran the *Compiler* benchmark using both the concurrent collector and the stop-and-copy collector and measured pauses caused by persistent garbage collector activity and transaction commit operations. The performance of transaction commit processing has not been the focus of my implementation efforts, and the design choices in this area have been made more for simplicity and ease of implementation than for performance, because the primary goal was to explore the feasibility of replicating garbage collection.

Figures 3-10, 3-11, and 3-12 show the pauses from concurrent collector activity, stop-and-copy collections, and transaction commit, respectively. Note that the scale of Figure 3-10 is smaller than in the other figures because the pauses caused by the concurrent collector are very brief. The commit pauses shown here were produced using the concurrent collector, and are essentially the same as those produced when using the stop-and-copy collector. Stop-and-copy and commit pauses have been grouped into histogram bins 300 milliseconds wide.

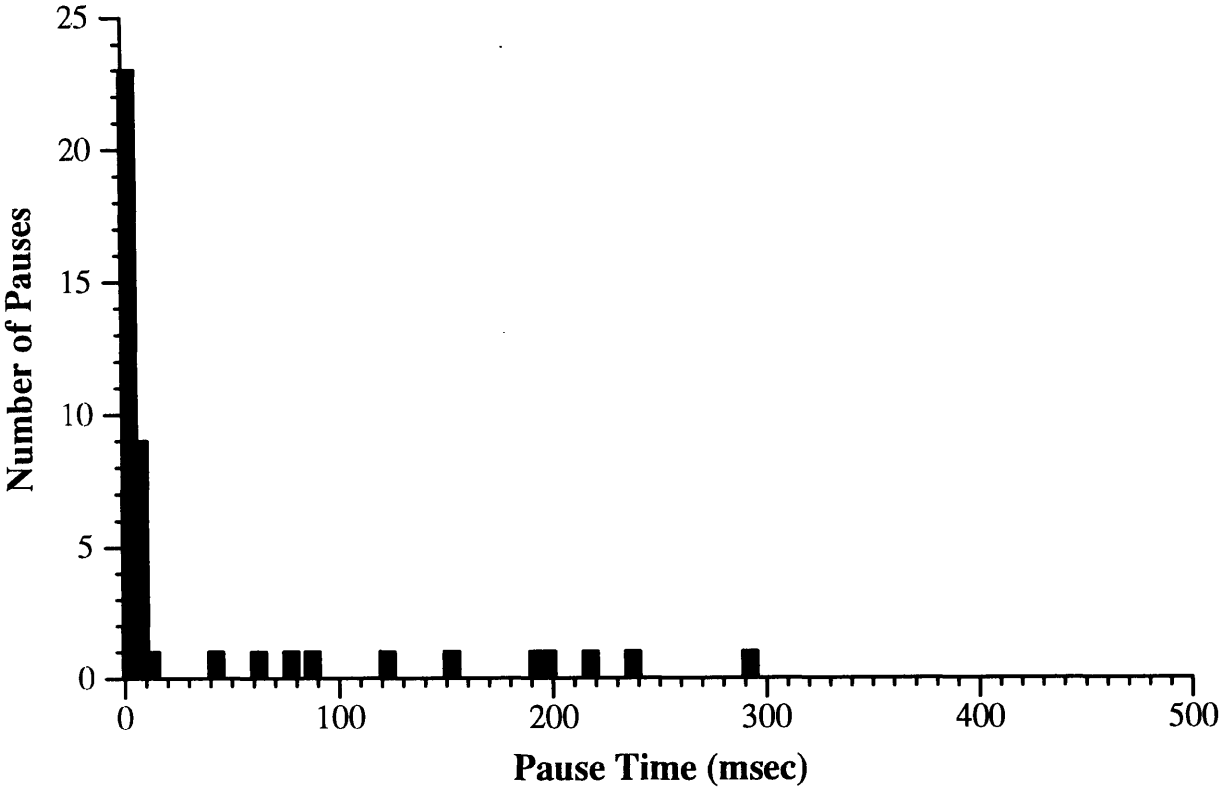

Figure 3-10: Concurrent Collector Pause Distribution

As shown in Figure 3-10, the pauses due to the concurrent collector are mostly very short. The long tail is comprised entirely of pauses during which the collector completed a flip. Even the longest pause is short compared to most commit pauses. The total time spent by the client waiting for the collector was 1760 milliseconds. This time is less than the shortest stop-and-copy pause.

In contrast, the pauses shown in Figure 3-11 show that all of the stop-and-copy pauses are long enough to be disruptive. Many applications would be unable to use this collector due to the long interruptions. The total time spent in collection was 35 seconds, almost twenty times the time spent waiting for the concurrent collector.

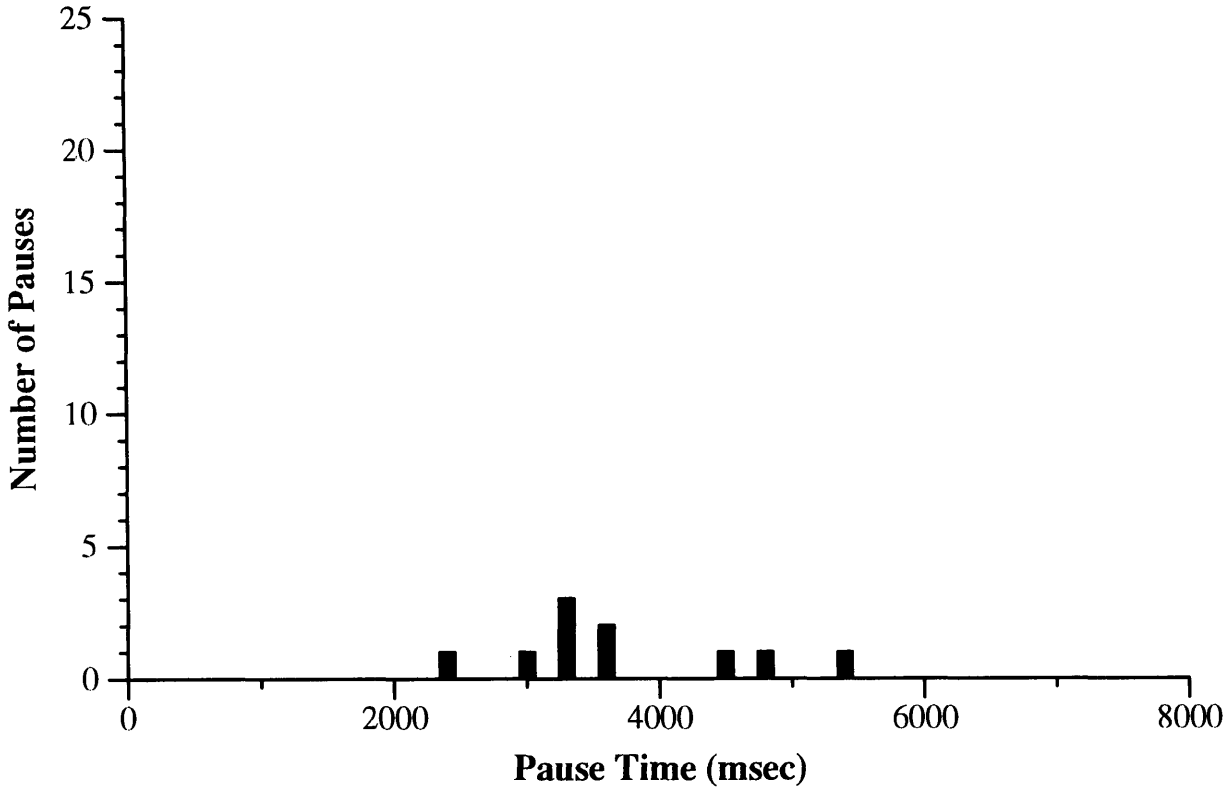

Figure 3-11: Stop-and-Copy Pause Distribution (Note scale)

3.5.4 Transaction Throughput

The throughput of the system is interesting to examine, and two immediate questions come to mind: First, how good is the performance for small simple transactions? Second, does concurrent replicating collection provide increased throughput?

Commit Performance

In order to test the fastest path for transaction commit, I executed the TPC-B program using a 12 megabyte database containing bank account records for 1 branch covering 100,000 customers. During this test, the implementation ran 80 transactions per second. The limiting factor was the synchronous disk write required by each transaction. Note that this database size is smaller than the TPC-B guidelines require for a system claiming 80 transactions per second. I believe that the other requirements of the benchmark

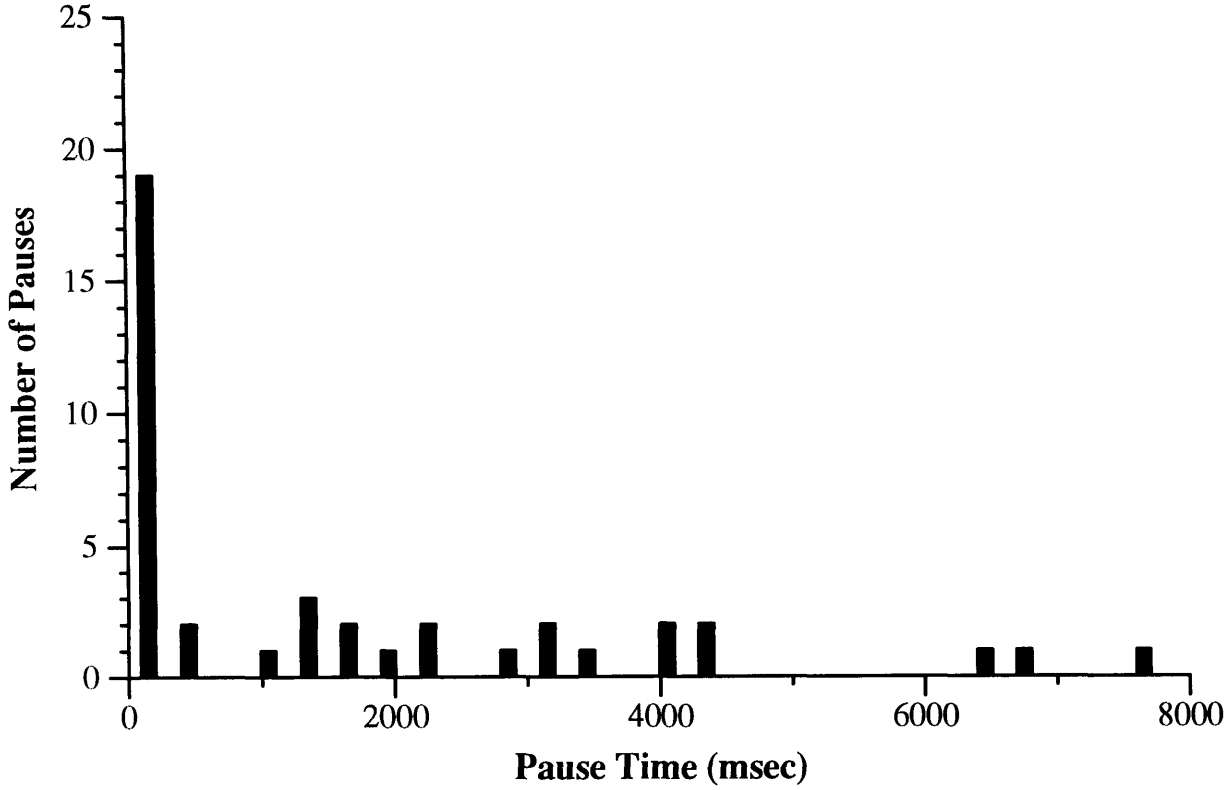

Figure 3-12: Commit Pause Distribution (Note scale)

standard have been met. A similar RVM benchmark [63] also executed approximately 80 transactions per second on the same machine.

When the TPC-B benchmark is completing 80 transactions per second, each transaction requires 12.5 milliseconds of processing. I made more detailed timing measurements to see what was happening during those 12.5 milliseconds. Measurements of the transaction manager and independent measurements of write calls show that the synchronous write accounts for about 8 milliseconds or 65% of the total time. Other processing by RVM during the end transaction accounts for another 10%. A surprise was that an additional 10% was due to an instruction cache flush operation required because in general the collector may copy executable machine code (much of which is stored within the SML/NJ heap). Traversing the undo/redo log and logging the entries to RVM accounts for an additional 5%. No single factor accounts for the remaining 10% of the time. The time spent executing the benchmark code, which was too small to measure accurately,

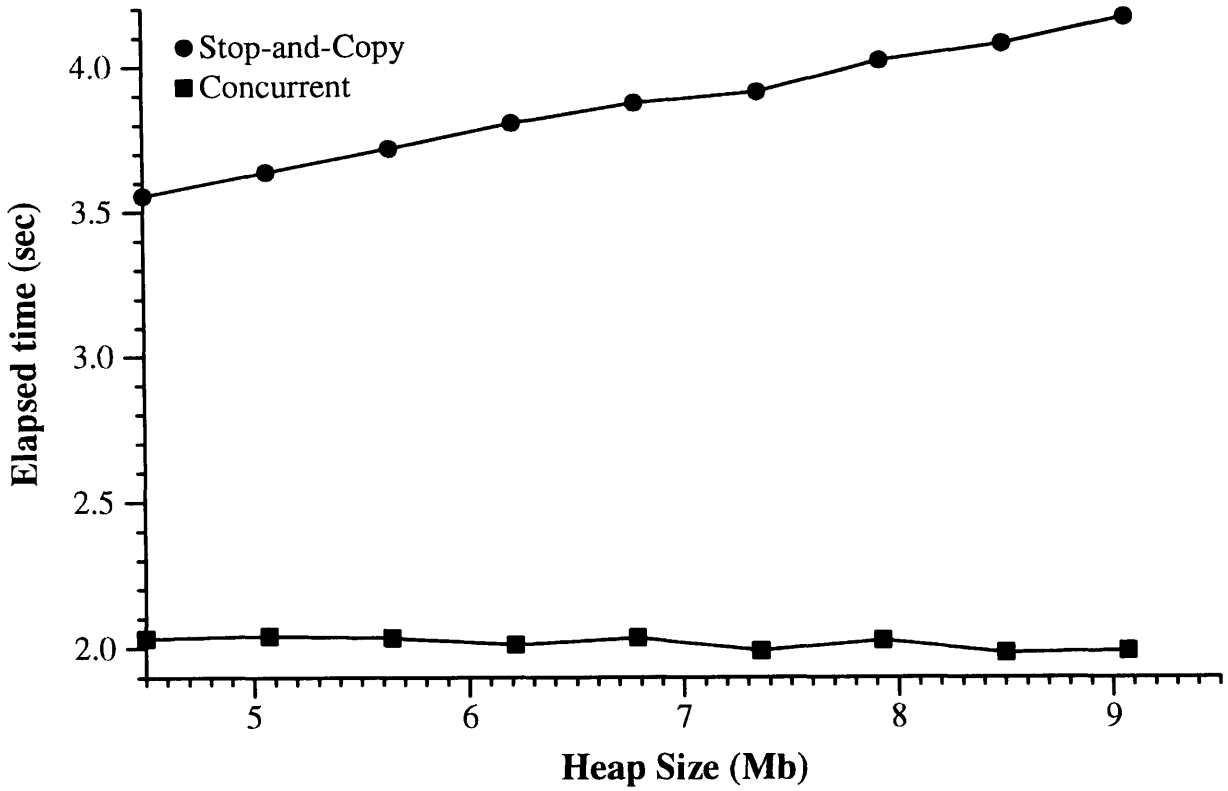

Figure 3-13: OO1 Elapsed Time per Transaction

was less than 0.5% of the total per-transaction time.

Heap Size Dependence

The OO1 benchmark was used to study the transaction throughput of the system in the presence of garbage collection. Remember that this version of OO1 performs twice as many updates to persistent data because each transaction includes one hundred deletions and one hundred insertions. This change to the benchmark causes it to require garbage collection, yet maintain a constant amount of live data in the heap. I measured the elapsed time to perform the standard engineering modification described by the benchmark.

The total heap size was controlled by adding various amounts of live data. Figure 3-13 shows the elapsed time to perform each transaction as a function of heap size. The use of concurrent collection is clearly advantageous. The elapsed time for the concurrent

collector is constant with heap size. As expected, the stop-and-copy collector causes the elapsed time to increase linearly with heap size.

3.5.5 Improving Stable Storage Access

In most transaction systems the critical performance issue is access to stable storage. In this section I discuss a variety of performance issues and enhancements. Most of these issues are closely tied to the use of RVM for stable storage.

Transaction Reordering

I ran the compiler benchmark using configurations in which all writes of stable to-space either only used the disk directly or only used RVM. Using only the disk made it hard to synchronize with the client to achieve the flip. The key difficulty was scheduling the flip when no disk write was in progress. This problem occasionally caused very long pauses, and increased the number of pauses in the few hundred millisecond range. In the configuration that used only RVM, the extra log traffic from the garbage collector thread introduced substantial additional commit delays.

In the concurrent replicating algorithm, the write traffic from the garbage collector need not slow down the commit traffic from the client because the writes are to different spaces. The transactions that the collector performs on to-space can be reordered with the client transactions on from-space without affecting correctness. A suitable change to RVM would allow the GC transactions to move through the log independently of other transactions.

Change Record Batching

For large transactions, such as those in the compiler benchmark, I examined the detailed components of both commit and collection closely. A significant fraction of the time is spent processing the log, primarily in logging the modifications through RVM.

This observation motivates several possible enhancements that should improve commit performance and/or reduce total overhead.

Many of the modifications on the log are to the same location. Recognizing these duplicate log entries and eliminating them would reduce the logging cost substantially. A more efficient interface to transfer the log information to RVM would also be advantageous. Currently, the transaction manager calls RVM once per change record in the log, and RVM validates each such call independently. An interface that allowed a group of change records to be processed together would allow RVM to reduce its overhead without compromising safety.

Write-Ahead Logging

RVM delays capturing the value of a redo record until its transaction commits. Given the current logging methods, it would be acceptable to capture the value when the change record is first logged. Then RVM could use write-ahead logging aggressively.

The transaction manager could also take advantage of write-ahead logging by promoting newly persistent objects eagerly. Currently no promotions are done until commit. Earlier promotion would allow the cost to be absorbed in existing collection work and would be especially beneficial for long running transactions like those in the compiler.

Log Editing

When the RVM log fills with transaction records it must be emptied. RVM does this by applying the log records to the data files and truncating the log. However, if the garbage collector has flipped the stable from-space and the stable to-space, then many log records are obsolete, because they contain changes to the heap that has been reclaimed by the collector. It would be much more efficient to inform the stable log manager (RVM) that these log records can be removed from the log entirely.

In the current implementation, this optimization would also be very useful whenever the Copy thread bypasses RVM and writes directly to stable to-space. Currently the

Copy thread must force an RVM log truncation before issuing these writes in order to ensure that RVM is not holding any old log records that apply to stable to-space. It would be much more efficient to discard these log records instead.

Garbage collectors sometimes benefit from similar features in data caches and virtual memory systems [12]. When the garbage collector reclaims a semi-space it is better that cache lines and virtual memory pages that contain reclaimed data be reset to a zero-fill-on-demand status because there is no need for the underlying memory system to preserve their old contents.

Recently the designers of RVM have begun the implementation of an incremental log truncation mechanism [63]. This new feature will subtly change the semantics of RVM. Currently the only changes to virtual memory that RVM applies to stable storage are the changes that the client explicitly logs via the RVM interface. The replicating collector takes advantage of this property when it overwrites portions of volatile from-space with relocation information containing the new address for each replicated object. Because the collector does not notify RVM of these changes, the original contents of these locations will be restored after a failure. The proposed incremental truncation technique will use pages of the virtual memory to update the disk file instead of using the log entries. Given the collector design, this new RVM behavior would result in an unacceptable loss of committed data.

3.5.6 Recovery Performance

Although I have not measured recovery performance, it is apparent that the cost of recovery is almost entirely attributable to RVM. There are two phases of recovery processing in the implementation. First, the RVM log manager must recover the last committed physical heap image. Second, the transaction manager must undo any uncommitted modifications that are present in the persistent heap. The cost of processing the undo log is small relative to the cost of RVM recovery. RVM recovery must perform disk operations to reconstruct the committed contents of the stable heap from its stable log,

3.6 Extensions and Applications

It should be possible to support multiple client threads by using the Venari transaction model [77] developed by Scott Nettles. It will also be worthwhile to examine the remaining sources of delay in the pauses due to the concurrent collector, but this effort is subject to rapidly diminishing returns as the remaining pause time becomes comparable to the duration of other unpredictable operating system interruptions. There are several simple optimizations that could compress the update logs substantially. Also, a lesson learned from the performance measurements is that there are several desirable features that are candidates for addition to the RVM log manager.

The current implementation does not support the restart of a partially completed garbage collection, but the changes required are minimal. Because the client uses only from-space, the recovered state of to-space is of little importance. The state of the replicating garbage collector can be recovered as long as its volatile data structures are periodically checkpointed to stable storage. As I mentioned in Section 3.4.7, the recoverability of incomplete collections may not be worthwhile for some applications.

The replicating garbage collection design should be applicable in small interpretive-language environments such as those proposed for so-called mobile code or agent-based computing paradigms. Recently proposed examples include Telescript and Java, languages that may be used primarily to implement interactive applications using relatively small garbage collected heaps. Replicating garbage collection may also prove to be valuable for very large persistent heaps that are accessed using swizzled internal and external representations, via a cache, or in a distributed programming environment containing multiple volatile heaps. There are two primary advantages of replicating collection in these configurations: The client and the garbage collector need not be as tightly coupled as in other garbage collection algorithms and therefore system features that depend on knowledge of object modifications and object locations are easier to build.

Chapter 4

Closing Thoughts

I've demonstrated, by building two working systems, that update logs can be used to provide automatic garbage collection and indexing services in information storage systems. Automatic indexing and garbage collection are useful features of storage systems because they make sharing information much easier. Automatic indexing enables users and programmers to locate shared information without substantial prior agreement about information structure. Automatic garbage collection enables programs to share information without detailed agreements about object ownership. In the next few sections, I present some additional conclusions and contributions that follow from the work presented here.

4.1 Automatic Indexing Services

The semantic file system prototype shows that associative access can be added to an existing file storage system by using file type specific transducers. The indexing operations can be performed incrementally and efficiently by using an update log that records modifications to file system objects. This work also shows that the associative access features can be integrated into the file system using virtual directories that are computed on demand. Other contributions of the semantic file system work include the following:

- Virtual directories integrate associative access into existing tree structured file systems in a manner that is compatible with existing applications.

- Virtual directories permit unmodified remote hosts to access the facilities of a semantic file system with existing network file system protocols.

- Transducers can be programmed by users to perform arbitrary interpretation of file and directory contents in order to produce a desired set of field-value pairs for later retrieval. The use of fields allows transducers to describe many aspects of a file, and thus permits subsequent sophisticated associative access to computed properties. In addition, transducers can identify entities within files as independent objects for retrieval. For example, individual mail messages within a mail file can be treated as independent entities.

The implementation of real-time incremental indexing required the SFS process to intercept existing file system traffic and generate an update log that would direct transducing and indexing effort at modified file system objects. It is natural to conclude that effective indexing efforts for other storage architectures, such as the distributed storage of the World Wide Web, will depend on the availability of accurate update logs.

4.2 Replicating Garbage Collection

I have implemented a log-based concurrent compacting garbage collector for a transactional persistent heap. The design is based on replicating collection, a new garbage collection method. The collector uses a log that is shared with the transaction manager. The prototype implementation demonstrates that client activity can continue during the garbage collection of stable data. These innovations were made possible by the development of the replicating garbage collection algorithm.

Experimental measurements of the system show that concurrent replicating collection offers garbage collection pauses that are much shorter than those caused by stop-and-copy

collection. The interrupts suffered by the client are small in comparison to transaction commit latencies and are independent of stable heap size. Transaction performance provided by the prototype is good. The use of garbage collection in the transaction commit processing added little overhead; commit performance remains dominated by the underlying log manager. This design offers garbage collection performance that will be useful to real-time operating systems applications that require safe persistent storage.

4.3 Future Storage Systems

I demonstrated in this dissertation that update logs can be used to provide automatic indexing and garbage collection services in information storage systems. This thesis is subsidiary to my more general thesis that there are practical ways to support information sharing in programming systems. I believe that by providing update logs as an ubiquitous feature of storage systems, systems designers can enable the construction of services that promote information sharing and enable many unanticipated uses of the stored information.

File systems are increasingly being used to store information in loosely standardized formats that support inter-object pointers. However, the paradigm in ascendance as the time of this writing is the World Wide Web, due to its success in enabling users to conveniently share information at low cost. As file system and web storage standards such as OLE 2.0 and HTML continue to evolve, it seems likely that these systems will provide better ways for people to use distributed file systems to store large open networks of interconnected objects.

A persistent world-wide object store will present many complex design and implementation challenges, and the storage semantics that will be found most useful to programmers may be impossible to predict. There are now many active efforts underway to provide automatic indexing in the context of the World Wide Web, and based on the experience of this thesis, I would argue that the long-term success of these efforts will

depend on the adoption of storage interfaces that provide access to update logs. It seems more difficult to predict whether these same update logs will be used to support garbage collection as we know it today, but I believe update logs will prove to be invaluable in supporting automatic storage management services. I expect that future information sharing services will also benefit from the simplicity and flexibility that update logs have provided in my garbage collection and automatic indexing and designs.

Bibliography

[1] Guy T. Almes. Garbage Collection in a Object-Oriented System. Technical Report CMU-CS-80-128, Carnegie Mellon University, June 1980.

[2] A. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software–Practice and Experience*, 19(2):171–183, February 1989.

[3] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Garbage Collection on Stock Multiprocessors. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 11–20, 1988.

[4] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *SIGPLAN Notices*, 17(7):24–31, July 1982.

[5] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[6] Brian N. Bershad and C. Brian Pinkerton. Watchdogs: Extending the UNIX file system. In *USENIX Association 1988 Winter Conference Proceedings*, pages 267–275, Dallas, Texas, February 1988.

[7] Brent Callaghan and Tom Lyon. The automounter. In *USENIX Association 1989 Winter Conference Proceedings*, 1989.

[8] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. In *Fourth International Conference on Architec-*

tural Support for Programming Languages and Operating Systems, pages 200–211, Santa Clara, California, April 1991. ACM.

[9] R. G. G. Cattell. An Engineering Database Benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 247–281. Morgan-Kaufmann, 1991.

[10] CCITT. The Directory - Overview of Concepts, Models and Services. Recommendation X.500, 1988.

[11] Claris Corporation, Santa Clara, California, January 1990. News Release.

[12] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the Performance of SML Garbage Collection using Application-Specific Virtual Memory Management. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 43–52, June 1992.

[13] Lotus Corporation. Lotus Magellan: Quick Launch. Product tutorial, Lotus Corporation, Cambridge, Massachusetts. Part number 35115.

[14] Transactions Processing Council. TPC-B. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 79–114. Morgan-Kaufmann, 1991.

[15] P. B. Danzig et al. Distributed indexing: A scalable mechanism for distributed information retrieval. Technical Report USC-TR 91-06, University of Southern California, Computer Science Department, 1991.

[16] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, pages 57–69, December 1988.

[17] David L. Detlefs. Concurrent, atomic garbage collection. Technical Report CMU-CS-90-177, Carnegie Mellon University, October 1990.

[18] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly Garbage Collection:An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[19] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time Concurrent Garbage Collection on Stock Multiprocessors. Technical Report DEC-SRC-TR-25, DEC Systems Research Center, February 1988.

[20] Ferdinando Gallo, Regis Minot, and Ian Thomas. The object management system of PCTE as a software engineering database management system. In *Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 12–15. ACM, December 1986.

[21] David K. Gifford, Robert G. Cote, and David A. Segal. Walter user's manual. Technical Report MIT/LCS/TR-399, M.I.T. Laboratory for Computer Science, September 1987.

[22] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Semantic file systems. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 16–25. ACM, October 1991. Available as *Operating Systems Review* Volume 25, Number 5.

[23] Computer Systems Research Group. UNIX User's Reference Manual. 4.3 Berkeley Software Distribution, Berkeley, California, April 1986. Virtual VAX-11 Version.

[24] Olivier Gruber. Eos: An Environment for Persistent and Distributed Applications in a Shared Object Space. host ftp.inria.fr directory INRIA/Projects/RODIN file Olivier.Gruber.phd.ps.Z, December 1992.

[25] Lorenz Huelsbergen and James R. Larus. A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data. In *Proceedings of the 1993 ACM Symposiym on Principles and Practice of Parallel Programming*, 1993.

[26] Information Dimensions, Inc. BASISplus. The Key To Managing The World Of Information. Information Dimensions, Inc., Dublin, Ohio, 1990. Product description.

[27] Brewster Kahle and Art Medlar. An information system for corporate users: Wide Area Information Servers. Technical Report TMC-199, Thinking Machines, Inc., April 1991. Version 3.

[28] Michael Leon Kazar. Synchronization and caching issues in the Andrew File System. In *USENIX Association Winter Conference Proceedings*, pages 31–43, 1988.

[29] T. J. Killian. Processes as files. In *USENIX Association 1984 Summer Conference Proceedings*, Salt Lake City, Utah, 1984.

[30] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association 1986 Winter Conference Proceedings*, pages 238–247, 1986.

[31] Eliot K. Kolodner. Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap. Technical Report MIT/LCS/TR-534, Massachusetts Institute of Technology, February 1992.

[32] Legato Systems, Inc. Nhfsstone. Software package. Legato Systems, Inc., Palo Alto, California, 1989.

[33] M. E. Lesk. Some applications of inverted indexes on the UNIX system. UNIX Supplementary Document, Section 30.

[34] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):382–404, July 1983.

[35] Barbara H. Liskov. Overview of the argus language and system. Technical Report Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, February 1984.

[36] Boss Logic, Inc. Boss DMS development specification. Technical documentation, Boss Logic, Inc., Fairfield, IA, February 1991.

[37] Jeffrey C. Mogul. Representing information about files. Technical Report 86-1103, Stanford Univ. Department of CS, March 1986. Ph.D. Thesis.

[38] Scott M. Nettles. Safe and Efficient Persistent Heaps. Technical Report CMU-CS-95-225, Carnegie Mellon University, 1995.

[39] Scott M. Nettles and James W. O'Toole. Implementing Orthogonal Persistence: A Simple Optimization Based on Replicating Collection. In *Proceedings of the SIGOPS International Workshop on Object-Orientation in Operating Systems*, December 1993.

[40] Scott M. Nettles and James W. O'Toole. Real-Time Replication Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 217–226. ACM, June 1993.

[41] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-Based Incremental Copying Collection. In *Proceedings of the SIGPLAN International Workshop on Memory Management*, pages 357–364. ACM, Springer-Verlag, September 1992.

[42] S.M. Nettles and J.M. Wing. Persistance + Undoability = Transactions. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 2, pages 832–843. IEEE, January 1992.

[43] B. Clifford Neuman. The virtual system model: A scalable approach to organizing large systems. Technical Report 90-05-01, Univ. of Washington CS Department, May 1990. Thesis Proposal.

[44] NeXT Corporation. 1.0 release notes: Indexing. NeXT Corporation, Palo Alto, California, 1989.

[45] NeXT Corporation. Text indexing facilities on the NeXT computer. NeXT Corporation, Palo Alto, California, 1989. from 1.0 Release Notes.

[46] ANSI Z39.50 Version 2. National Information Standards Organization, Bethesda, Maryland, January 1991. Second Draft.

[47] Brian M. Oki. Reliable Object Storage To Support Atomic Actions. Technical Report MIT-LCS-TR-308, Massachusetts Institute of Technology, 1983.

[48] Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler. Reliable object storage to support atomic actions. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 147–159, New York, December 1985. ACM.

[49] James O'Toole, Scott Nettles, and David Gifford. Concurrent Compacting Garbage Collection of a Persistent Heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles.* ACM, SIGOPS, December 1993.

[50] James W. O'Toole. Garbage Collecting an Object Cache. Technical Report MIT/LCS/TM-485, Massachusetts Institute of Technology, April 1993.

[51] James W. O'Toole and Scott M. Nettles. Concurrent Replication Garbage Collection. Technical Report MIT-LCS-TR-570 and CMU-CS-93-138, Massachusetts Institute of Technology and Carnegie Mellon University, 1993.

[52] James W. O'Toole and Scott M. Nettles. Concurrent Replicating Garbage Collection. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1994.

[53] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2bsd file system. In *Symposium on Operating System Principles*, pages 15–24. ACM, December 1985.

[54] Jan-Simon Pendry. Amd — an automounter. Department of Computing, Imperial College, London, May 1990.

[55] Jan-Simon Pendry and Nick Williams. Amd: The 4.4 BSD automounter reference manual, December 1990. Documentation for software revision 5.3 Alpha.

[56] Larry Peterson. The Profile Naming Service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[57] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. UK UUG proceedings, 1990.

[58] Mark B. Reinhold. Cache Performance of Garbage-Collected Programming Languages. Technical Report MIT/LCS/TR-581, MIT Laboratory for Computer Science, September 1993.

[59] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Comm. ACM*, 17(7):365–375, July 1974.

[60] Marc J. Rochkind. *Advanced UNIX Programming.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[61] Gerard Salton. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[62] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *USENIX Association 1985 Summer Conference Proceedings*, pages 119–130, 1985.

[63] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[64] Michael F. Schwartz. The Networked Resource Discovery Project. In *Proceedings of the IFIP XI World Congress*, pages 827–832. IFIP, August 1989.

[65] Mark A. Sheldon. *Content Routing: A Scalable Architecture for Network-Based Information Discovery.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1995.

[66] Mark A. Sheldon, Andrzej Duda, Ron Weiss, James W. O'Toole, Jr., and David K. Gifford. Content routing for distributed information servers. In *Fourth International Conference on Extending Database Technology*, pages 109–122, Cambridge, England, March 1994. Available as Springer-Verlag LNCS Number 779.

[67] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, March 1987. Sixth Edition, Version 18.

[68] Richard Marlon Stein. Browsing through terabytes: Wide-area information servers open a new frontier in personal and corporate information services. *Byte*, pages 157–164, May 1991.

[69] Sun Corporation. The Network Software Environment. Technical report, Sun Computer Corporation, Mountain View, California, 1988.

[70] Sun Microsystems, Sunnyvale, California. *Network Programming*, May 1988. Part Number 800-1779-10.

[71] NFS: Network file system protocol specification. Sun Microsystems, Network Working Group, Request for Comments (RFC 1094), March 1989. Version 2.

[72] ON Technology. ON Technology, Inc. announces On Location for the Apple Macintosh computer. News Release ON Technology, Inc., Cambridge, Massachusetts, January 1990.

[73] Verity. Topic. Product description, Verity, Mountain View, California, 1990.

[74] Peter Weinberger. CBT Program documentation. Bell Laboratories.

[75] Brent B. Welch and John K. Ousterhout. Pseudo devices: User-level extensions to the Sprite file system. In *USENIX Association 1988 Summer Conference Proceedings*, pages 37–49, San Francisco, California, June 1988.

[76] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 SIGPLAN International Workshop on Memory Management*, pages 1–42. ACM, Springer-Verlag, September 1992.

[77] J.M. Wing, M. Faehndrrich, N. Haines, K. Kietzke, D. Kindred, J.G. Morrisett, and S.M. Nettles. Venari/ML Interfaces and Examples. Technical Report CMU-CS-93-123, Carnegie Mellon University, March 1993.

[78] Benjamin Zorn. The Measured Cost of Conservative Garbage Collection. *Software—Practice and Experience*, 23(7):733–756, July 1993.