InfoMonitor: Unobtrusively archiving a World Wide Web server

Brian Cooper, Hector Garcia-Molina
Department of Computer Science
Stanford University
{cooperb,hector}@DB.Stanford.EDU

Abstract

It may be important to provide long-term preservation of digital data even when that data is stored in an unreliable system, such as a filesystem, a legacy database, or even the World Wide Web. In this paper we focus on the problem of archiving the contents of a web site without disrupting users who maintain the site. We propose an archival storage system, the InfoMonitor, in which a reliable archive is integrated with an unmodified existing store. Implementing such a system presents various challenges related to the mismatch of features between the components, such as differences in naming and data manipulation operations. We examine each of these issues as well as solutions for the conflicts that arise. We also discuss our experience using the InfoMonitor to archive the Stanford Database Group's web site.

1 Introduction

Important and irreplaceable data is constantly being lost. For example, some data from the 1960 U.S. Census has disappeared, and files detailing land usage patterns and the location of natural resources in New York State are irretrievable now [12]. As more and more data is placed on the unreliable World Wide Web (technical papers, virtual museums, product catalogs), the losses of valuable information are sure to increase in the future. The fundamental problem is that important data has been placed and will continue to be placed on storage systems (web servers, email servers, file systems, legacy database systems) that were not designed with data archiving as the primary goal. The designers of these *legacy* data stores focused on other design goals, such as efficiency, ease-of use, or reduced cost, usually to the detriment of the long term reliability of the system. These systems are used for convenience, since there are large code bases and applications available.

Our goal is to build an information archiving system that combines the best of both worlds, legacy and archival. This combined system should allow data to be accessed precisely as it is accessed today in the unreliable legacy system, so that all applications continue to run, and users are not aware of the reliability features unless they need them. At the same time, the combined system should preserve the information selected by a site administrator, storing data for historical purposes and also making it possible to restore data into the unreliable system when necessary.

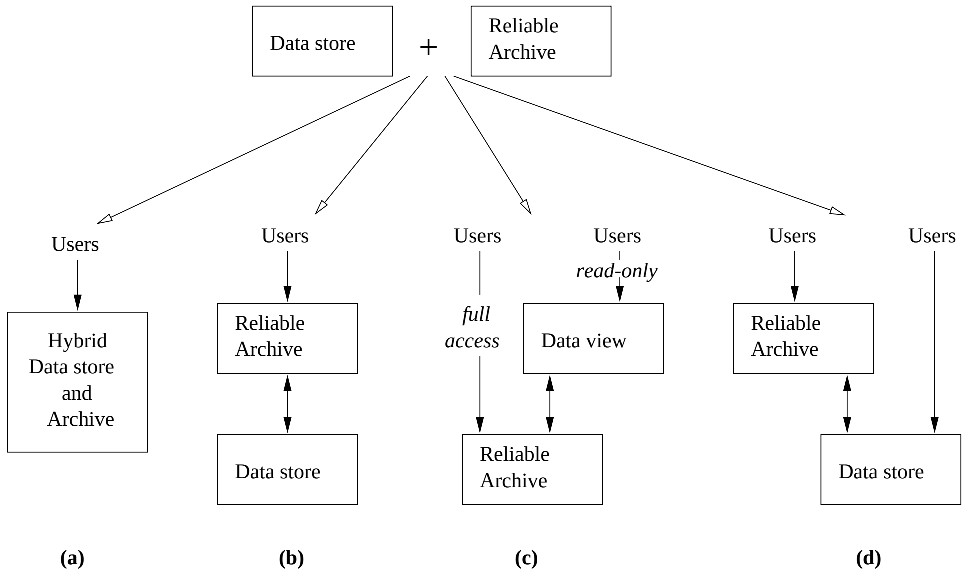

Figure 1: Options for combining an existing data store with an archival component.

Let us see how we can reconcile these apparently "impossible" goals for the combined system. Figure 1 illustrates four approaches that might be considered. The first option (Figure 1a) is to redesign the legacy system to have both the traits of the unreliable data store and a reliable archive. This of course is an expensive solution if we must build a system from scratch. Even if there is an existing code base, we have to get access to the code and permission to modify it. In either case, the resulting hybrid system may not even be backwards compatible in the end.

The remaining three solutions avoid these problems by integrating the existing legacy system with a component built for archival storage. In the second solution, Figure 1b, an archival repository module intercepts calls to the legacy system, and performs necessary actions to preserve information. For example, if a file is updated, the previous version may be automatically saved (perhaps in a different underlying legacy system for added reliability). In order for users to access the system, their applications would have to be relinked to access the reliable archive module. Even if users have the source code for all of their applications, the new interface may require significant application rewriting, and they may not be willing to deal with the cost and effort to redeploy applications. In the third solution (Figure 1c), the data is actually moved to a reliable archival system, and a read-only view is made available on the legacy system. This approach is used today by many web sites, which store their base data in a DBMS, and generate HTML views for their web site. The main disadvantage is that only read access is provided through the original legacy system (the web site). Changes to the data must be directed to the new system, which once again involves changing any application that updates the data.

Finally, the fourth solution, Figure 1d, is to take *existing* legacy and archival systems and combine them so that each still offers its native services. Users are free to access legacy data in traditional ways, while the archival system "tracks" changes in the legacy system and "translates" them into equivalent archiving operations in order to preserve the information. Although all four alternatives have strengths and may be the correct choice for certain domains, there are situations where the first three options are not available. If existing systems and applications should be retained unmodified, the alternative (d) offers a great deal of flexibility for accomplishing archiving without disrupting usage. This is the design we examine here and in our implemented archiving system, the *InfoMonitor*.

However, there are still challenges to implementing solution (d) since the two components we are integrating can have very different properties, and it may be hard to combine them in the absence of modifications to the systems themselves. For example, if the legacy and archival systems use different object naming mechanisms, it is not clear how a user-initiated name change in the legacy system should be reflected in the named object structure of the archival system. We address these challenges in this paper.

To illustrate the challenges and our solutions we will focus on the case study of a reliably archived web site. Many web sites store pages on a standard filesystem rather than a DBMS or reliable archive. This allows users to modify these files using their favorite applications, including text editors and graphics software, while providing a convenient environment for server side programming, such as CGI scripts. However, filesystems are notoriously unreliable. Thus, it is necessary to archive the data served by the web site in a reliable database system. If the web site authoring and management software does not provide such archiving capabilities (and most packages do not; see Section 7), a reliable archive should be integrated with the filesystem itself. We have chosen to use the Stanford Archival Vault (SAV) [9, 4], a simplified version of a DBMS which focuses on long term archiving, for the reliable archive component. Note that preserving web pages requires effective storage of the pages themselves as well as the hyperlink structure between them; the issue of maintaining links by preserving page names is discussed in Section 3.

In this paper we consider the various problems that arise when integrating an archiving system with a traditional data store, focusing on the situation where the data store cannot be expected to provide active assistance for archiving. Specifically, we make several contributions:

- A proposed architecture for adding archiving capabilities to an existing data store. This architecture is flexible and provides archiving transparently.

- Solutions for specific issues that arise when implementing this architecture, including data feature mismatch, naming and the definition of the set of data to archive.

- An implemented system, called the *InfoMonitor*, for the archived web server situation, but which is applicable to other domains as well.

- Discussion of performance and scalability issues. The system performs well on a large-sized web site, and this performance scales linearly with the size of the archived data set.

Our system is similar in many ways to previous work in areas such as data backup and file versioning. Unlike these previous systems, we focus on the heterogeneity involved in archiving data

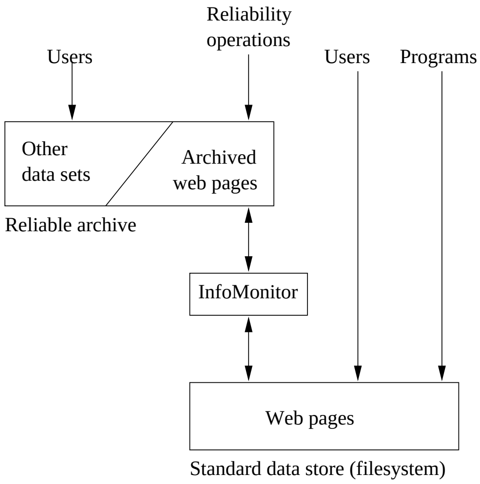

Figure 2: The architecture of our implemented InfoMonitor system.

to a store different from the filesystem. Moreover, our approach offers a great deal of flexibility and new features. We will discuss these differences in more detail in Section 2 after we present an overview of our system.

The InfoMonitor architecture is applicable to archiving data on any legacy store, although modifications to the implementation may be required. For example, archiving the entire World Wide Web instead of a single website would would require examination of copyright issues, whether and how to "restore" data after a failure, and the different types of information provided by the HTTP protocol versus the filesystem interface we use here. Moreover, ensuring scalability at the level of the entire web would be challenging. We discuss scalability issues in Section 6.

The paper is organized as follows. First, we examine the architecture of the InfoMonitor in Section 2 by examining the steps users go through to archive data. Next, we examine issues that arise when implementing the InfoMonitor. Section 3 illustrates the problem of detecting filesystem changes and representing those changes in the archive. Section 4 examines the problem of representing filenames in the archive, while Section 5 examines the situation where only a subset of the filesystem's data should be archived. In Section 6 we present our experience using the system to archive our own web site, including performance measurements. Finally, Section 7 examines related work and in Section 8 we outline our conclusions.

2 The InfoMonitor

This design of our system is shown in Figure 2. Each of the three boxes in the figure represent a distinct component that can run on its own machine. The standard data store is an existing filesystem supporting a web server. If the InfoMonitor runs on a separate machine from the web server, it can access the server's files using a network filesystem. The InfoMonitor accesses the archive using the archive's native protocol. The InfoMonitor can work with a variety of archives, and the archive may store other data sets not managed by the InfoMonitor.

The InfoMonitor is started by a system administrator who specifies which portions of the filesystem should be archived. The specification gives a start point for a depth-first traversal of the filesystem, and a *filter set* that determines which files are archived (see Section 5). The start point for archiving a web collection is usually the root of the public_html hierarchy. In order to reach individual user's public_html directories, the InfoMonitor can utilize symbolic links from the central public_html directory (if they exist) or can start at the root of the filesystem and filter out non-WWW specific directories. The administrator also specifies the archive to use. After startup, the InfoMonitor runs unattended, and the administrator only intervenes to restore data after a failure.

Once the InfoMonitor is started, it scans the filesystem, transferring all of the files that pass its filter set into the archive. For each file, the InfoMonitor archives the content of the file as well as some metadata (e.g., the filename). Issues related to managing filename metadata are discussed in Section 4. Moreover, the InfoMonitor archives the filesystem's directory hierarchy by creating directory and membership objects. The InfoMonitor also keeps an in-memory "summary" of the files (including a content signature) to be used to detect new and modified files.

After startup, the InfoMonitor unobtrusively monitors file activity by continually scanning the filesystem. When a user creates, modifies or deletes a file, the InfoMonitor detects this event as a discrepancy between the filesystem state and its in-memory summary, and copies the affected file into the archive. Changes are represented in the archive using version chains. In particular, deleted files are a treated as a new, "empty" version. This process is described in more detail in Section 3, where we also discuss the challenge of efficiently scanning the filesystem.

Because each created object is also timestamped, the archived files can be viewed in two ways: historically as files change over time, or as an instantaneous filesystem snapshot. The InfoMonitor provides a GUI for both of these perspectives. The *files view* shows the historical chain of modifications; a screenshot from our operational system is shown in Figure 3. (Our GUI demonstrates the capabilities of our operational prototype, and is not intended to be a sophisticated final product.) Files are displayed hierarchically, as they exist in the directory structure of the filesystem. Each folder icon represents a directory, while each "stacked document" icon represents a version chain. The user can expand a directory to view its contents, and can expand a version chain to see the individual versions. Each version is represented by a "single document" icon and the timestamp when that version was archived. In the figure, the file `cooperb.html` exists in three different versions. The user can select a version and click the "View" button to see the version's contents. The files view allows an administrator to navigate the history of the filesystem on a file by file basis.

A second view, the *snapshot view*, displays the entire state of the filesystem at a specified point in time. An example screenshot is shown in Figure 4. The snapshot is organized hierarchically using the directory structure. The time and date at the bottom of the window is the instant at which the snapshot was valid. Users can update the snapshot to the current filesystem state using the "Reset date" button, or can specify a point in the past in the "Date" box and click the "Apply date" button. Users can select a file and click the "View" button to examine the file contents.

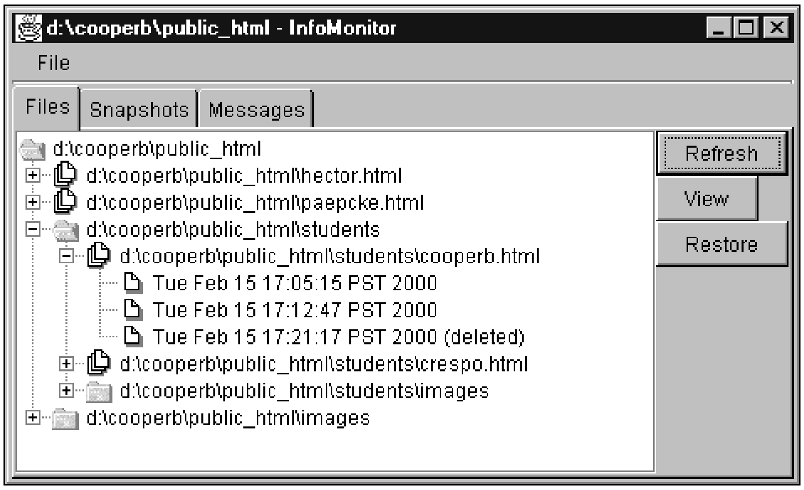

Figure 3: The *files view*.

Note that the archiving process is both transparent and comprehensive. It is transparent because users continue to access and modify files using their favorite applications, and do not expend extra effort to achieve archiving. It is comprehensive because it covers objects not available through the HTTP interface, such as CGIs and scripts that generate dynamic web pages. Our current implementation does not archive a database backend if it is not stored on the standard filesystem. Usually such backends provide their own archiving capabilities, although the InfoMonitor can be extended if necessary to retrieve database information.

Human intervention is only necessary in the event of a filesystem failure. Both the *files view* and the *snapshot view* provide facilities for quickly restoring data. In each case, the restored data is treated as a "new version," allowing the InfoMonitor to maintain a simple linear version chain. The "Restore" button in Figure 3 can be used to restore previous versions of individual files. In contrast, the "Restore" button in Figure 4 restores an entire snapshot. Restoring a snapshot is tricky because it is possible to overwrite portions of the filesystem not covered by the snapshot, especially if the filter has changed over time. This issue is discussed in Section 5. Failures of the archive are handled by its own internal mechanisms without intervention from the InfoMonitor.

It may be necessary to restart the InfoMonitor because of crashes or upgrades. On startup, the InfoMonitor writes a "bookmark" into the archive referencing the archived directory hierarchy. When the InfoMonitor is restarted, it reads the bookmark, traverses the archived hierarchy, and reconstructs its internal file summaries. After this recovery process, it continues scanning the filesystem looking for file modifications. This recovery process is efficient because the InfoMonitor only needs to examine its own archived objects, not the entire archive.

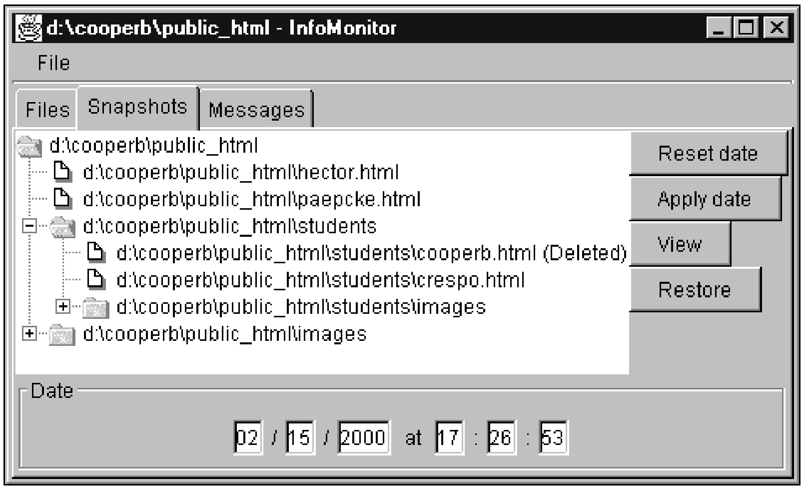

Figure 4: The *snapshot view*.

2.1 The Stanford Archival Vault (SAV)

In our prototype, the archiving system is the Stanford Archival Vault (SAV). In this section, we give a brief overview of the SAV. For more details about the design and implementation of SAV, see [9, 4].

The SAV system has the following characteristics:

- Objects are constructed as a list of tagged fields, containing either data or metadata.
- A signature function is used to create object names. Specifically, a 64 bit CRC is calculated for every object and used as the name. The signature provides several functions, including
  - A location service: signatures are used to locate objects in the store.
  - Error detection: signatures are cached and periodically checked against the object; differences indicate object corruption.
  - Efficient object comparison: objects are identical if their signatures are identical.

  Although there is the possibility of signature collisions between different objects, the nature of the CRC function minimizes the probability that different objects have the same signature. Additional steps can be taken to prevent collisions; see [9] for a full discussion of this issue. Here we assume that the naming scheme is sufficiently robust that collisions do not occur.
- Metadata fields in objects can also contain signatures, which are interpreted as references to other objects. This provides the capability to create graph structures of objects.
- The store provides an interface to create new objects and read existing objects. However, data, once written, cannot be deleted or overwritten. The goal is to prevent data loss due to accidental or intentional data erasure.

- Reliability is provided by automatic replication and periodic comparison of data between several SAV sites. Corrupted or missing objects at one site are automatically replaced with pristine copies obtained from other sites.

2.2 Differences with existing approaches

As mentioned in the introduction, there has been a lot of work on tracking changes and failure recovery. For example, a data warehouse monitors data sources and copies changes to a centralized store [1, 21, 29]. In our context, the filesystem is the data source, the reliable archive is the warehouse, and the InfoMonitor is the data extractor. However, unlike data warehousing, the InfoMonitor focuses on the ability to restore data to the original source. Thus the InfoMonitor must save information necessary to map archive objects back into the corresponding source objects. Furthermore, the InfoMonitor is designed to deal with very large numbers of relatively small files (e.g., web pages) as opposed to a few very large files (e.g., tables containing sales records). As a result, the techniques for extracting changes are different than the ones used in say [22]. Finally, the focus of a data warehouse is to provide processing, not archiving. While it is possible for a warehouse to provide data histories, the primary focus of the InfoMonitor is to store complete historical snapshots of the archived information.

Our work is also related to file backup systems [3]. While file backup systems are in widespread use, there have not been many academic studies of backup systems. In addition, such systems tend to copy data from a filesystem to a similarly structured backup system, avoiding the heterogeneity that we must deal with here. Furthermore, the InfoMonitor offers a lot more flexibility than traditional backup utilities. For example, the InfoMonitor offers powerful filters to define what is tracked, and it checks for corrupted or changed file contents, not just for changed timestamps. The InfoMonitor GUI also offers more fine grain control for browsing and restoring files.

The InfoMonitor borrows concepts from previous versioning technologies [14, 28]. For instance, versioning systems like RCS provide a way to maintain a history of changes to files. However, there are three important differences with "traditional" versioning systems. One, the InfoMonitor tracks an *autonomous* filesystem, and stores the versions on a separate system. In a traditional system, the versioning mechanism is notified whenever an object is changed. In contrast, the InfoMonitor must detect changes itself and must minimize the number of missed changes. Two, the filesystem and the archive use different object representations. For example, if the name of a file is changed, this generates a completely different action at the archive. Three, the InfoMonitor can use a *simple* version mechanism targeted to file restore only. That is, the InfoMonitor does not need to consider alternatives but can store versions as a linear chain. (There is always an unambiguous "current version" instead of multiple alternate current versions created by branches in the version tree.)

Other related work is discussed in Section 7.

3 Data manipulation

The archive should record information about operations performed on the filesystem, even though those operations are not native to the SAV. Specifically, the following types of information should be archived:

- *content*: Data contained in the contents of files.
- *structure*: Information about relationships between files.
- *events*: Information about the sequence of modifications to a file.

An archive must obviously retain file content, but structure between files is equally important. For example, when archiving a web site it is important to archive the hyperlink structure between HTML pages, so that if the web site must be reconstructed all of the links still work. This requires the InfoMonitor to both detect structures and record them in the archive. Unfortunately, in a filesystem the only structure that is explicitly represented is the hierarchical organization of files in directories. Other structures, such as hyperlinks, are represented in an application specific manner, and the InfoMonitor must take steps to interpret and preserve this structure information.

Finally, the sequence of events should be recorded so that if necessary certain events can be undone. For example, if a user accidentally erases or overwrites important data, it should be possible to select that event to be undone, while preserving previous intentional modifications. The concept of a version chain (such as in versioning systems like RCS [28]) can be used to represent filesystem events as a sequence of resulting file versions. However, as mentioned earlier, the filesystem does not provide explicit notification when a new version has been created (e.g., there is no file "check-in"). Note that here we treat a file creation as the start of a new version chain, and file deletion as a modification which produces an empty version.

3.1 Archiving inter-file structure

The only native filesystem structure present in a Unix filesystem is the directory hierarchy. In this hierarchy, there are two types of objects: files and directories. Files are atomic, while directories are complex objects that can contain files and other directories. The filesystem provides operations for determining which objects are directories, and for listing the contents of those directories. The InfoMonitor creates a "set" object in the SAV to represent directories, and adds members to the set to represent files. Nesting sets provides the ability to represent nested subdirectories.

Application-specific structure can vary widely. However, we are primarily concerned with the domain of a web server, where the structure is limited to hyperlink references between files. Because these hyperlinks are constructed from filenames, successfully solving the naming problems described in Section 4 means that the hyperlink structure will be effectively archived. While it would be possible to parse the HTML and "swizzle" the links to refer to archive objects (using explicit SAV object references) this is unnecessary work that does not add any expressiveness to the archiving.

Other application-specific structure types could be detected by the InfoMonitor and represented as explicit object references, if necessary.

3.2 Detecting file modifications

The InfoMonitor must detect and report *filesystem events*: new files, modifications and deletions. The InfoMonitor constructs a summary of the filesystem when the monitor is first started, and then continually scans the filesystem, comparing the summary to the files. A discrepancy between the summary and the filesystem indicates a filesystem event, causing the InfoMonitor to migrate the affected data into the archive and update the summary. Because we expect that the filesystem will contain a large amount of data, the summary must be small and facilitate time-efficient data comparison, or events will be missed.

We have divided the scanning process into two components:

- *quick scan*: uses filesystem timestamps to indicate events
- *slow scan*: compares file contents using a signature algorithm

Both scans start at an initial location in the filesystem, and perform a depth first search of the directory structure. The selection of initial location and the nature of the search is further described in Section 5.

In order to perform the *quick scan*, the InfoMonitor's in-memory summary is a list of file names along with the filesystem timestamp for each file. By relying on timestamps, the InfoMonitor is able to quickly scan the filesystem for events and archive affected data. The quick scan operates as follows:

- A file on the filesystem that is not in the summary is new. The filename and timestamp are added to the summary and the file is written to the archive.
- A file that is in the summary but does not exist on the filesystem has been deleted. The filename is marked as "deleted" in the summary and an "empty version" is archived.
- A file that is "deleted" in the summary but exists on the filesystem has been undeleted. The deletion mark is removed from the summary, and the undeleted version is archived.
- If a file is in the summary and on the filesystem, then the filesystem timestamp is compared to the timestamp stored in the summary. If the filesystem timestamp is newer than the summary timestamp, the file has been modified. The summary timestamp is updated, and the new version of the file contents is archived.

Unfortunately, timestamps on the filesystem cannot always be trusted. The primary reason for adding the reliable archive is that objects in the filesystem can become corrupted, including directory objects containing timestamps. There are also other factors which may reduce the reliability of timestamps, such as clock skew between hosts on the same network filesystem.

Therefore, the InfoMonitor also performs the *slow scan.* The slow scan operates using the same steps as the quick scan. However, instead of comparing timestamps, the slow scan compares the current contents on the filesystem to the last version that was archived. Differences indicate that the file has changed and that the new version should be archived.

One way to perform this comparison would be to retrieve each file from the archive and compare it to the filesystem copy. We rejected this *retrieve-from-archive* solution due to the performance implications. In our implemented system, the SAV and the InfoMonitor reside on different machines, and it is prohibitively expensive to transfer large amounts of data over the network connecting the machines. Even if the archive and the InfoMonitor were on the same host, repeatedly reading every file from the archive is expensive and should be avoided if possible.

An alternative scheme to achieve this comparison is to use *caching*: store the entire contents of every archived file in the memory of the InfoMonitor process. These cached files would then be compared to the current filesystem file to detect changes. However, storing gigabytes (or more) of file contents in main memory is simply not feasible.

In order to achieve reasonable performance, we use the following *signature-based* process:

1. The 64 bit CRC of every new or modified file detected by the quick scan is calculated and stored in the summary (together with the filename and timestamp).

2. The slow scan calculates the CRC for every file on the filesystem, and compares it to the CRC in the summary.

    - If the signatures are different, the file has changed and the new version must be archived. The new signature replaces the old signature in the summary.

Note that if the signatures are different but the timestamp has not changed, this is a hint that the file has been corrupted, since every "normal" filesystem operation should update the timestamp. Because timestamps cannot be fully trusted, we cannot be sure that corruption has occurred based on this hint alone, so the InfoMonitor prints a message to the user that the file may have been corrupted. The user can take appropriate action.

The signature-based scheme has two disadvantages compared to the retrieve-from-archive or caching approaches. First, the signature-based scheme may miss file modifications if the signature of the modified file is the same as the signature of the original file. The nature of the CRC algorithm allows us to make the probability of such false negatives arbitrarily small by using enough signature bits. Second, signatures stored in memory may become corrupted due to memory errors or be lost if the InfoMonitor host crashes. The InfoMonitor can be restarted in these cases and will recalculate all of the signatures based on the data reliably stored in the archive. Alternately, the content signatures could also be archived and retrieved periodically at less cost than retrieving the entire data set.

The InfoMonitor runs the quick scan much more frequently than the slow scan (which is much more time consuming). The quick scan will detect most of the filesystem events. A few modified files will be missed due to timestamp problems but these files will be caught by the slow scan. A few events will still be missed because the quick scan is not instantaneous, and a file may change several times between repeated examinations. However, the nature of the data in our system does not require that every modification be archived. If a user is editing a file, it important to archive the final version of the file, and intermediate saves of the file during the editing are not as vital.

4 Naming

The archive must retain file names as reliably as the file contents. For example, web pages are located using URLs which contain the filename. Other documents link to pages using the URL and losing naming information could mean creating dangling references. Moreover, if a filesystem must be restored, the filenames must be restored as well. Users can change the names of files arbitrarily, and the InfoMonitor must update the archive appropriately after a name change. Ideally, the InfoMonitor would store a *naming history* for each archived file.

There are several challenges that arise when dealing with naming:

- It is difficult to determine when a file's name has changed.
- It must be possible to locate archived objects using their filename.
- Files can have aliases.

We examine each of these issues in this section.

4.1 Detecting name changes

The InfoMonitor must update the archive after a name change. As with the data modifications discussed in Section 3.2, the InfoMonitor cannot expect explicit notification of a name change. Unlike file modifications, however, the quick and slow scans are unable to detect name changes. The problem is that it is difficult to distinguish between the case where a file's name changes and the case where one file is deleted and another, with similar content but a different name, is created.

One solution is to find files with identical content. For example, if a user renames `index.html` to `home.html`, the InfoMonitor would see the disappearance of `index.html` followed by the appearance of the same content in `home.html`. The InfoMonitor could then record a simple name change rather than a deletion followed by a file creation. This is a naive solution that fails to acknowledge that files with the same content may not be related by a name change. Moreover, it is expensive to examine the content of every file.

A better solution would be to utilize some sort of *arbitrator* that can determine whether two files are the same. In Unix, files are identified by an inode number that does not change across simple name changes. Two files with the same inode number are the same, even if the filename changes. This solution is limited in applicability to Unix filesystems; other filesystems may not have inodes.

The approach we take in the InfoMonitor is to view a file renaming as a deletion of one file followed by the creation of a different file. Thus, the InfoMonitor updates the archive after a name change but does not keep an explicit naming history for individual files. The domain of archiving a web server suggests our approach is appropriate. From the perspective of a web server, objects with different names are in fact different objects. If a user changes the name of a file, then the web server can no longer find the file under the previous name. Only by updating the hyperlinks from other pages is it possible to locate the new object. From the web server perspective, this is equivalent to deleting the old object, creating a new one with a different name, and updating the hyperlinks. Therefore it is not necessary to resort to an arbitrator to determine if objects with different names are the same object; we simply assume that they are not.

4.2 Locating archived files by filename

The SAV uses its own naming scheme for objects (specifically, content-based signatures), and does not provide a direct mechanism for finding a file based on the filesystem name. However, users may need to find archive objects based on the filename. For example, to restore a file to the filesystem, the user should be able to specify the filename, not the SAV signature. The obvious solution is to build a mapping from filename to SAV signature by loading every SAV object and examining its filename field. The mapping could list all of the signatures of multiple versions of the given file. This mapping could be maintained incrementally whenever the quick scan archives a new object.

However, the InfoMonitor provides another option. Recall from Section 3.2 that the InfoMonitor maintains an in-memory summary of the filesystem. This summary is indexed by filename, and contains the SAV signature of the most current version of the file in order to build version chains. The signature is also used to locate SAV objects based on the filesystem name. The InfoMonitor performs a lookup into the summary to get the current version signature, and then traverses the version chain to find all of the versions. This capability eliminates the need to build and maintain a separate name mapping, reducing time and memory overhead.

4.3 Filesystem aliasing

Filesystems provide the ability for the same file to have different names. Usually, the InfoMonitor can treat two different names as two different files (for the same reason it ignores renaming; see Section 4.1). However, symbolic links can create cycles in the filesystem graph. Such cycles can create an infinite number of aliases for the same file. For example, if directory `/dir` contains a symbolic link named `subdir` which points to `/dir`, then every object named `file` in `/dir` is simultaneously named `/dir/file`, `/dir/subdir/file`, `/dir/subdir/subdir/file`, and so on. Because we regard each name as a different file, there would be an infinite numer of files to be archived. We cannot ignore symbolic links because they may form a vital part of the filesystem scan (e.g., a symbolic link from the central public_html directory to a user's personal public_html directory may be the only way to reach that user's files from the InfoMonitor's start point).

We resolved this issue by reducing symbolic links to check for previously scanned objects. For each file and directory, the filesystem is queried for its *canonical name*: the name constructed from the directory hierarchy DAG that results from removing all symbolic links. If `dir` is an actual directory and not a symbolic link, then the canonical name of `file` is `/dir/file`. When `/dir/subdir` is encountered, it is reduced to its canonical name, `/dir`, and ignored since the InfoMonitor has already scanned `/dir`.

5 Determining what to archive

It is possible to archive the entire contents of the filesystem, but it is often desirable to archive only a subset. In a web server, the same networked filesystem that contains the web data also contains applications, the operating system, private user data, etc., none of which should be archived. The administrator should be able to configure the InfoMonitor to select a particular set of files to archive. This *logical set* should contain semantically related items (e.g., all web pages and associated data).

One way to construct this set would be to maintain a list of all files to be archived. This scheme would require frequent human intervention to add newly created files to the list, a requirement we want to avoid. Another option is to specify a directory, and have the InfoMonitor archive every file in this directory and its subdirectories. This option may not be appropriate if there are files in that directory tree that should not be archived. Moreover, the web files may not be organized in a single directory tree but instead spread throughout the filesystem in different users' directories. Symbolic links can be used to deal with the latter problem, but administrators should not be required to create symbolic links explicitly for the purpose of archiving (which should occur transparently).

Thus, it should be possible to specify the characteristics of a file that would be in the logical set, and let the InfoMonitor decide which files fit these characteristics. This can be viewed as a query on the filesystem: the query defines the logical set, and any files in the query results are archived. Unfortunately, the filesystem does not provide sophisticated querying, and the InfoMonitor must provide its own (simple) query engine.

Another option (which we do not use) is to utilize the hyperlink structure contained in a set of web pages. The logical set could be defined as all web pages reachable via hyperlinks from a starting web page. Any time a user modifies a page in the existing logical set to link to a new page, that page would be implicitly added to the logical set. It may be necessary to stipulate that the hyperlink traversal does not cross filesystem boundaries, or that it extends no more than n links from the start page. This may be the correct approach if the InfoMonitor archived pages retrieved via HTTP instead of filesystem files.

5.1 The InfoMonitor uses simple querying to define a logical set

The InfoMonitor uses querying in the form of a *filter set*: a sequence of regular expressions, each of which filter out certain files based on the absolute path and filename of the file. The filter set is applied to a filename as follows:

1. The name is checked against each filter in sequence.

    - If the filter matches the name and is a *reject* filter, the name is rejected.
    - If the filter matches the name and is an *accept* filter, the name is accepted.

2. If the name is not accepted or rejected by any filter in the sequence, then the default policy, either accept or reject, is applied.

This process gives an administrator flexibility in designing a logical set. For example, it is possible to accept or reject all names except a specified few.

This filter set is applied by the InfoMonitor during both the quick and slow scans to both select individual files in the logical set and to prune the filesystem search tree. When the InfoMonitor finds a directory, it applies the path name of the directory against the filter set and recurses into the directory if the name passes. When the InfoMonitor finds a file, it applies the path and filename against the filter set and examines the file if the path and filename pass.

The administrator does not need to change the filter set once it is specified. New files that are created will be archived if they pass the filter set, allowing the logical set to grow implicitly. However, we do allow the administrator to change the contents of the filter set if necessary. Recall from Section 2 that a "snapshot" of the entire logical set as of a point in the past can be restored; the set of restored items is defined by what filter set was used when the snapshot was valid. Therefore, it is important to store a timestamped version chain of the filter sets. A filter set is just a file that is automatically added to every logical set and archived like other files.

We use the following criteria to determine how a snapshot is restored, based on the filter set that was active at the time of the snapshot:

- A file present in the snapshot must have passed the filter set, and is restored to the version it had when the snapshot was valid. This includes files that no longer exist on the filesystem.
- A file that was "deleted" when the snapshot was valid is only deleted from the filesystem if that file would have passed the filter set for the snapshot. A change to the filter set can cause the file to be removed from the logical set, and the InfoMonitor would record this as a "deletion" of the file. Files removed from the InfoMonitor's jurisdiction should not be affected by the restore.
- Files present in the filesystem but not mentioned in the snapshot are not affected.

Thus, the version chain of filter sets is critical to the snapshot restore process.

6 Performance measurements

We have measured the performance of our implemented system. Our goal is not to produce the most efficient implementation or a production-level system. Instead, we wish to demonstrate a proof-of-concept and to explore performance. Nonetheless, we have attempted to construct a system that

	InfoMonitor	**SAV**	**Web Server**
System	Sun Ultra 2	IBM Intellistation	Sun Ultra 2
Processor	Dual 450 mhz UltraSPARC	450 mhz Pentium II	Dual 200 mhz SPARC
Memory	256 MB RAM, 1800 MB swap	256 MB RAM, 512 MB swap	256 MB RAM, 1230 MB swap

Table 1: The hardware specifications of the machines used to collect performance data.

Logical Data Set	Period	24 contiguous weekday hours
	Number of files	24,885
	Total size	1.6 GB
Initial Load	Duration	2 hours, 8 minutes
Quick Scans	Average duration	5 minutes, 20 seconds
	New files detected	5,020 files (155 MB)
	Modified files detected	14
	Deleted files detected	14,462
	Undeleted files detected	9,452
Slow Scans	Average duration	55 minutes, 31 seconds

Table 2: The results of running the InfoMonitor on the entire web site.

performs well. We have focused especially on performing the quick scan as rapidly as possible to reduce the number of missed events (see Section 3.2).

The InfoMonitor system, including the user interface, consists of 3,400 lines of Java 2 code. The monitor communicates with the SAV, also written in Java, using Visibroker CORBA. Table 1 describes the hardware we used to collect performance data. The InfoMonitor and the SAV ran on different machines connected via 10 Mbps ethernet. This models the situation where the SAV is running on an archiving host and the InfoMonitor serves as a remote client. The InfoMonitor machine and the HTTP server machine share the same DFS filesystem.

We used the InfoMonitor to archive the files from the Stanford Database Group's web site. The web data consists of over 24,000 files (1.6 gigabytes). In addition to text HTML files, this data set contains images, programs, compressed files, word processor documents, raw data files, and a variety of other file types. The size and diversity of this filesystem allowed us to test the system for both scalability and robustness for different kinds of data. While the InfoMonitor was running, users were actively accessing and modifying files. The measurements for the phases of the InfoMonitor's operation are shown in Table 2. (See Section 3.2 for descriptions of these tasks.)

We believe these measurements are appropriate for an effective demonstration of the system. First, the files represent a wide variety of data produced by many different users. Moreover, the data set is larger than the data stored at an average web site. A recent study by Inktomi reports that on average, a website contains 200 HTML pages [6]. Our group's web site contains over five times as many pages, as well as many more non-HTML files (representing 23 graphics, scripts, downloadable documents, etc. per HTML page). Therefore, running the InfoMonitor on our site

Task	Time (seconds)	Percent
Read files from disk	722	9.3 %
Build summary	199	2.6 %
Buffer copy	2124	27.4 %
Network transfer to SAV	3340	43.1 %
Store in SAV	1373	17.7 %

Table 3: The breakdown of the initial load time.

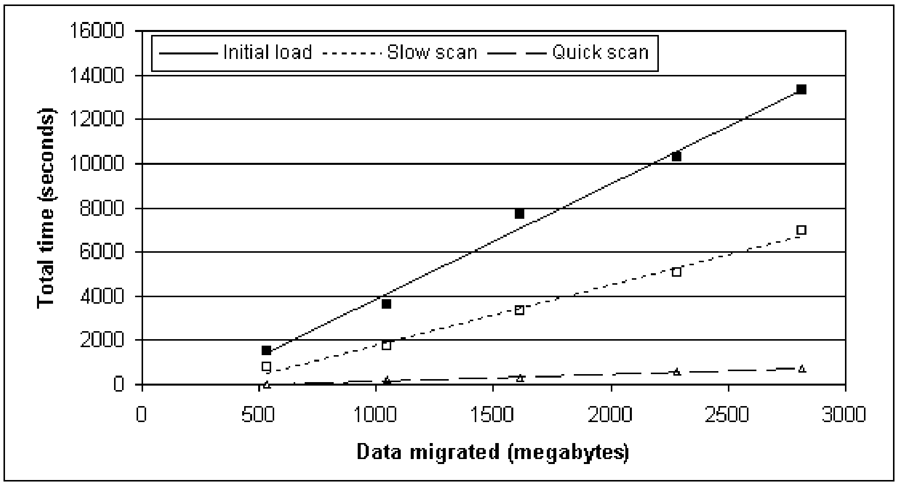

Figure 5: The total time of the InfoMonitor's tasks as a function of the size of the data set.

reveals the performance of the system for a large-sized web site.

Much of the time required to perform the archiving is unavoidable. To illustrate this point, we list the breakdown of times for the initial load phase in Table 3. An "ideally performing" archiving system would still have to read files, transfer them to the archive, and save them in the archive's storage. According to Table 3, this unavoidable overhead accounts for over 70 percent of the initial load time. Of the remaining 30 percent, 27.4 percent is for copying file data into messages sent to the SAV; this "buffer copy" could be avoided with a more streamlined implementation. Building the in-memory filesystem summary only adds 3.5 percent overhead to the "ideal" archiving system, indicating that the InfoMonitor (other than the buffer copy) is very efficient.

To explore scalability, we used the InfoMonitor to archive data sets ranging from one-third to almost twice the size of our web site. These data sets were produced either by taking a subset of the 24,000 files in the group's site, or by duplicating certain files[1]. Therefore, each of these data sets contain similar mixtures of file types. The measurements are shown in Figure 5, where we have plotted the total time for each task against the size of the data set. The InfoMonitor's performance scales linearly with the size of the data archived.

[1]The InfoMonitor treated aliased files as a fixed number of different files, avoiding the infinite looping discussed in Section 4.3. Aliased files thus appeared as different objects, and the effect is that the data set contained more files.

Another interesting result shown in Figure 5 is that the quick scan duration increases very slowly as the data set gets larger. This is desirable, since the quick scan is the task performed most frequently. The use of the quick scan as the primary event detection mechanism results in a system that scales well to larger data sets.

Although the performance of our system scales linearly to data sets on the order of a few gigabytes, it is reasonable to ask whether this linearity will continue to much larger data sets (hundreds of gigabytes or larger). We believe it will. First, each of the data sets shown in Figure 5 is larger than the 256 megabytes of physical RAM, and two of the data sets are larger than the total swap space available (1,800 megabytes). Thus, working with data sets that do not fit in main memory (or even swap space) does not hinder linear scalability. The nature of the algorithms we use require individual files to be loaded and examined; since we do not join information between files we do not keep more than one file in memory at a time. Even if a file is large it can be examined in chunks. Simply examining more files should not introduce nonlinearity in performance.

The InfoMonitor keeps an in-memory summary of the files, and we can ask what will happen when the summary no longer fits in memory. Only 50 bytes of memory is required per file[2], and if we allocate 128 megabytes of main memory as the maximum summary size, the InfoMonitor can index in main memory 2.7 million files (roughly 170 gigabytes). For larger data sets, it is either necessary to purchase more RAM (which is relatively inexpensive) or to store some of the summary on disk. The InfoMonitor can organize a disk index using the order in which files are examined (based on a depth first traversal of the filesystem), and can load the index sequentially into memory based on the files currently being scanned. As a result, even an index stored on disk incurs a fixed cost per file archived, and the performance of the InfoMonitor for data sets larger than 170 gigabytes (with 128 megabytes of RAM) should be linear in the amount of data archived.

There are a few more conclusions we can draw from our experiments. First, the duration of the quick scan means that some filesystem events are missed, and we would like to decrease the quick scan duration. One solution is to divide the data set into subsets, and start a different InfoMonitor process for each subset. The processes could operate in parallel, possibly on different machines with the same networked filesystem.

Second, it is apparent that the InfoMonitor is responsible for detecting events in a small fraction of the filesystem. Although the quick scans detected almost 29,000 filesystem events in a 24 hour interval, 86 percent of the events were localized in 10 percent of the directories, and 65 percent of the directories experienced no events at all. Perhaps the scans can be optimized to focus more precisely on the "busy" directories; for example:

- Increasing the frequency of scans of "busy" directories while performing fewer scans of directories whose contents never change.
- Calculating a signature of the sequence of timestamps of files in a particular directory, and

[2]Each entry contains the filename, timestamp, content signature, a "file deleted" flag, the SAV object handle, and a pointer to the path name. The cost of shared storage of path names is amortized over an average of 20 files per directory. The sum of the fields, amortized cost of the path name, and hash table overhead is 50 bytes.

performing a file-by-file comparison only when the signature changes. This is similar to the signature snapshot comparison algorithm described in [8].

Finally, whenever a file is undeleted or modified (even slightly), the entire file is copied into the archive; 9,466 files were archived for this reason. The InfoMonitor could reduce space usage in the archive due to these new versions by creating a "diff" that represents only the changes to the file. Another option is to utilize a compression algorithm (such as the Lempel-Ziv coding [30] used in gzip) to reduce the space requirements of all files. Compression could be time consuming, and should not be done during the time-sensitive quick scan. Instead, the InfoMonitor could archive entire files, and then ask the archive to lazily compress all versions during periods of archive inactivity.

These optimizations will be tested in future work.

7 Related work

Considerable work has been done in the area of increasing the reliability of traditional filesystems. Some of this work has focused on data backup. A survey of current backup techniques is presented by Chervenak et al in [3], and Hutchinson et al [16] discuss the nature of the backup snapshot. Traditional strategies often require human intervention to mount magnetic tape, run the backup, and store backup tapes in a coherent way. Recently, systems designed to better automate this process have been introduced by IBM [5, 17] and UniTree Corporation [18]. King et al [20] suggest that a remote system storing backed up data could provide quick recovery in the event of a crisis. We discuss differences between the InfoMonitor and backup systems in Section 2.2.

Other work has focused on making the primary medium itself more reliable. Patterson et al argue for Redundant Arrays of Inexpensive Disks (RAID) in [23], and Schloss and Stonebraker extend this idea using distributed disks in [27]. Several researchers have proposed using logs as an integral part of filesystems to increase reliability; these include Rosenblum and Ousterhout's Log-Structured Filesystem [25]. Goldberg et al [13, 2] have designed and implemented an "Archival Intermemory" system that models a reliable archive as a RAM-like data store. Each system assumes the filesystem can modified significantly to incorporate the reliability features, which may not be possible. Moreover, these systems do not address the problem of integrating components with different properties; instead, the feature set of these reliable systems resemble that of a filesystem.

The data store integration problem is a common one in data warehousing. Researchers have investigated the problems of designing the architecture of warehouses, detecting changes in source data and maintaining view consistency [21, 22, 1, 29]. We discuss the relation of data warehouses to our work in Section 2.2. Layered filesystems are examined by Khalidi and Nelson [19], although they assume that filesystems contain hooks to facilitate layering.

Archiving data, including legacy data, is a significant challenge facing the digital library community. A survey of the problems and possible solutions is presented in [12], and the implications for solving archiving problems are discussed in [26]. Many digital library archiving schemes circumvent the component integration problem by migrating data to a reliable archive and providing access as a service. Examples include the Computing Research Repository (CoRR) [15], and the collections being built by the San Diego Supercomputer Center [24]. These schemes do not allow direct access to the original data store, which could be important to minimize the impact on users.

Many web sites use tools to author and manage the web pages, and it may be possible to integrate versioned archiving into these tools. For example, Microsoft's FrontPage package [7] provides RCS-like versioning of documents. However, current tools do not provide archiving at the level provided by the SAV system, and may not even provide any backup capabilities at all. Moreover, a robust solution should be applicable to web sites that do not use a central authoring system, allowing different users to use different HTML editing packages.

8 Conclusions

In this paper we have presented the InfoMonitor, a system that increases the long term reliability of data objects, even if those objects are not already stored on a reliable archive. This problem is especially apparent in the context of a web server's unreliable filesystem. Although this filesystem is convenient for users, an archiving component must be added to provide long-term reliable storage. The InfoMonitor demonstrates how the filesystem can be unobtrusively archived, preserving information without disrupting users. The InfoMonitor must bridge the gap between the features of the filesystem and the reliable archive. We have discussed how the system detects data modification events, deals with naming, and finds members of a logical set. We have also explored the performance of the InfoMonitor, examining the scalability to large data sets and suggesting future improvements which could further optimize the system. The InfoMonitor deals with the specific situation of a web server, but the principles it illustrates can be applied to a wide variety of data storage systems whenever it is desirable to construct additional layers to preserve data.

Acknowledgements

We wish to thank Arturo Crespo for his comments and suggestions while we were building the system, and Sriram Raghavan for his comments on the paper draft.

References

[1] S.S. Chawathe, A. Rajarman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, June 1996.

[2] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[3] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings Joint NASA and IEEE Mass Storage Conference*, March 1998.

[4] Brian Cooper, Arturo Crespo, and Hector Garcia-Molina. Implementing a reliable digital object archive. http://www-db.stanford.edu/pub/papers/arpaper.ps, 1999. Technical Report.

[5] IBM Corporation. Adstar distributed storage manager (ADSM) - distributed data recovery white paper. http://www.storage.ibm.com/storage/software/adsm/adwhddr.htm, 1999.

[6] Inktomi Corporation. Web surpasses one billion documents. http://www.inktomi.com/new/-press/billion.html, 2000.

[7] Microsoft Corporation. Microsoft FrontPage. http://www.microsoft.com/frontpage/, 2000.

[8] Arturo Crespo and Hector Garcia-Molina. Awareness services for digital libraries. In *Lecture Notes in Computer Science*, volume 1324, 1997.

[9] Arturo Crespo and Hector Garcia-Molina. Archival storage for digital libraries. In *Proceedings of the Third ACM International Conference on Digital Libraries*, 1998. Accessible at http://-www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1998-0082.

[10] Arturo Crespo and Hector Garcia-Molina. Modeling archival repositories for digital libraries. http://www-db.stanford.edu/pub/papers/archsim.ps, 1999. Technical report.

[11] Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, Upper Saddle River, New Jersey, 2000.

[12] John Garrett and Donald Waters. Preserving digital information: Report of the Task Force on Archiving of Digital Information, May 1996. Accessible at http://www.rlg.org/ArchTF/.

[13] Andrew Goldberg and Peter Yianilos. Towards an archival intermemory. In *Advances in Digital Libraries*, 1998.

[14] Anja Haake and David Hicks. Verse: Towards hypertext versioning styles. In *Hypertext '96*, 1996.

[15] Joseph Halpern and Carl Lagoze. The Computing Research Repository: Promoting the rapid dissemination and archiving of computer science research. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, August 1999.

[16] Norman C. Huchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[17] Tivoli Systems Inc. Tivoli storage manager. http://www.tivoli.com/products/index/storage_mgr/, 1999.

[18] UniTree Software Inc. Unitree technical overview. http://www.unitree.com/overview/overview.htm, 1999.

[19] Yousef Khalidi and Michael Nelson. Extensible file systems in spring. In *Proceedings 14th Symposium on Operating Systems Principles*, December 1993.

[20] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Management of a remote backup copy for disaster recovery. *TODS*, 16(2):338–68, 1991.

[21] W. J. Labio, D. Quass, and B. Adelberg. Physical database design for data warehousing. In *Proceedings of the International Conference on Data Engineering*, April 1997.

[22] Wilburt Labio and Hector Garcia-Molina. Efficient snapshot differential algorithms in data warehousing. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, September 1996.

[23] David Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record*, 17(3):109–116, September 1988.

[24] Arcot Rajasekar, Richard Marciano, and Reagan Moore. Collection-based persistent archives. http://www.sdsc.edu/NARA/Publications/OTHER/Persistent/Persistent.html, 2000.

[25] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings 13th Symposium on Operating Systems Principles*, 1991.

[26] Jerome H. Saltzer. Technology, networks, and the library of the year 2000. In A. Bensoussan and J.-P. Verjus, editors, *In Future Tendencies in Computer Science, Control, and Applied Mathematics. Proceedings of the International Conference on the Occasion of the 25th Anniversary of INRIA*, pages 51–67, New York, 1992. Springer-Verlag.

[27] G.A. Schloss and M. Stonebraker. Highly redundant management of distributed data. In *Proceedings of Workshop on the Management of Replicated Data*, pages 91–95. IEEE, IEEE Computing Society, November 1990.

[28] Walter Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[29] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.

[30] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.