An Application-Aware Data Storage Model *

Todd A. Anderson, James Griffioen
Department of Computer Science
University of Kentucky

Abstract

We describe a new application-controlled file persistence model in which applications select the desired stability from a range of persistence guarantees. This new abstraction extends conventional abstractions by allowing applications to specify a file's volatility and methods for automatic reconstruction in case of loss. The model allows applications, particularly ones with weak persistence requirements, to leverage the memory space of other machines to improve their performance. An automated (filename-matching) interface permits legacy applications to take advantage of the variable persistence guarantees without being modified. Our prototype implementation shows significant speed-ups, in some cases more than an order of magnitude over conventional network file systems such as NFS version 3.

1 Introduction

In recent years, workstations and personal computers have become extremely powerful and have seen impressive increases in storage capacities. The cost effectiveness of these systems combined with emerging high-speed network technologies have led to local and intra-area networks of workstations with tens of gigabytes of memory storage and hundreds of gigabytes of disk storage. Studies show that the aggregate capacity and processing power of these systems are grossly underutilized. Acharya and others [1] discovered that, on average, 60% to 80% of a network's workstations are idle at any point and 20% to 70% of workstations are always available. Once a workstation has been idle 5 minutes, it will remain idle for an average of 40 to 90 minutes, implying that idle machines can be selected with high confidence that they will remain idle.

While aggregate processing power has grown sharply, applications have seen only modest performance improvements [21]. This is due in large part to the historic reliance of file systems on disk-based storage, characterized by slow access times. Moreover, the performance gap between disks and memory/remote memory is increasing. Several file system designs have noted the growing gap and have proposed ways to improve file system performance. Local client caching is a well-known technique to improve read times. More recent cooperative caching systems [10, 13, 23] have extended the caching model to use memory of other workstations in the system to improve the cache hit rate. Systems such as xFS [3] have improved write performance by striping data across the aggregate disk capacity of the system. Other approaches are described in Section 8.

Our goal is to shrink the gap between file system performance and rapidly improving hardware performance by introducing a new file system abstraction that can capitalize on the large aggregate idle resources of current distributed systems. We extend past approaches by allowing the application to control the tradeoff between persistence and performance.

2 Derby

In our previous work, we investigated a new file system design called Derby [15, 14] that used idle remote memory for both read and write traffic. Reads and writes occurred at remote memory speeds while providing the disk persistence necessary for database transaction-oriented storage. Derby assumes a small fraction of the workstations were equipped with uninterruptable power supplies (Workstations with UPS or *WUPS*). The system operates as follows. All active data resides in memory. Read requests that cannot be satisfied from the local cache use a dynamic address table lookup to find the idle machine that holds the data. The request is sent to a server process on the remote machine that returns the data from its memory. Write requests occur in a similar

*This work supported in part by NSF grant numbers CCR-9309176, CDA-9320179, CDA-9502645 and DARPA grant number DAAH04-96-1-0327.

manner but also send the written/modified data to one or more WUPS machines. The data is held temporarily in WUPS memory until the server process asynchronously writes the data to disk and informs the WUPS that the newly written data can be removed. By using WUPS for short-term persistence and disks for long-term persistence, Derby achieves disk persistence at remote memory speeds.

The advantage of Derby and similar main memory storage systems [18, 17, 24] is the ability to achieve traditional disk persistence at memory-speeds. However, disk persistence comes at a price. Such systems require special purpose hardware such as NVRAM or uninterruptable power supplies. In Derby, disk persistence increases the communication overhead between clients, servers, and WUPS servers. In addition, the disk system poses a potential bottleneck because *all* write traffic (including small, frequent operations such as file creation, deletion, and other metadata changes) hits the disk. Note that past file system analysis shows average file lifetimes to be quite short (80% of files and 50% of bytes last less than 10 minutes [4]). Thus, many files are likely to be written to cache and disk, read from the cache, and deleted, without ever being read directly from disk. This unnecessarily consumes CPU, bus, memory, disk bandwidth, and network resources in a distributed file system.

3 Enhancing Derby

Given the overhead and performance drawbacks associated with disk persistence, a file system that offers multiple persistence guarantees rather than a "one-size-fits-all model" has several potential benefits. Consider the Unix Temporary File System (tmpfs) which stores data in the high-speed (volatile) memory of the local machine and never writes the data to disk. Despite the potential for data loss, many applications are willing to take that chance in exchange for fast file access. Unfortunately, to use tmpfs, users must know where the tmpfs file system is mounted (assuming it is mounted) and must selectively place their temporary files in the tmpfs portion of the directory structure. This also forces logically related files with different persistence requirements to be stored in different places. As another example, local file systems often delay writes to disk to improve performance. In this case, all data eventually hits the disk, but recently written data may be lost, often to the surprise and dismay of the user. We conclude that applications are often willing to trade persistence for performance and that a one-size-fits-all persistence model will be suboptimal for most applications. In this paper, we develop a new file system that supports a *per-file, selectable-persistence* model.

Our analysis shows that the majority of data written to the file system can tolerate a persistence level between tmpfs and disk persistence. Note, this does not mean "the data will be lost" but rather that we will accept a slightly higher probability of loss in exchange for better performance. For example, web-browser cache files can be lost without ill effect. Locally generated object files (.o's) and output files from word processing and typesetting applications can be easily regenerated or replaced. Likewise, whether adding to or extracting from an archive file (for example, tar or zip), the resultant file(s) can be recreated as long as the original file(s) survive. Even files initially retrieved by HTTP or FTP, if lost, usually can be re-downloaded. Certainly, there are exceptions to the above generalizations and so a per-file persistence model is necessary to provide the correct persistence for atypical files.

To quantify the amount of file data that can take advantage of a selectable persistence abstraction, we snooped NFS traffic on our computer science networks. The trace was conducted for 3 days and recorded all NFS traffic sent to our 5 file server machines running Solaris 2.6. All user accounts reside on the file servers as is typical of many distributed file systems. The majority of activity consisted of reading mail, browsing the web (browser cache files), editing/word-processing/typesetting, compiling, and archiving (for example, tar and gzip). We recorded the names of all files opened for writing and the number of bytes written to each file. We then classified files as requiring disk persistence or weak persistence (something less than disk persistence). If there was any doubt as to the persistence required by a file, we erred on the side of caution and classified the file as requiring disk persistence stability. Note that the amount of weakly persistent data would be higher had traffic to `/tmp` been captured as part of this trace. The results of the study are reported in Table 1.

File Type	# of files	# of bytes
Weak persistence	24477 (75%)	4,410,711,305 (97%)
Disk persistence	8165 (25%)	116,717,201 (3%)

Table 1: Number of files/bytes written or modified.

Although 25% of files required disk persistence, most of these files were small text files (.h and .c files for example) created with an editor and did not require memory-speed write performance. Disk-persistent files also only accounted for a small portion of the total file system write traffic. Conversely, weakly persistent files made up the bulk of write traffic, consisting of compiler and linker output, LaTeX output (for example, .dvi, .log, and .aux files), tar files, Netscape cache files, and temporary files created by a variety of programs. From this we conclude that a large percentage of write traffic would trade weaker persistence guarantees for performance. Furthermore, we believe that the aggregate memory space of current networks is ideal for storing weakly-persistent data requiring fast access.

4 MBFS Overview

Our objective was to replace Derby's (and conventional file systems') one-size-fits-all persistence model with a configurable persistence model thereby removing the need for non-volatile memory and the other overheads of disk persistence. The new design, called *MBFS (Memory Based File System)*, is motivated by the fact that current distributed systems contain massive amounts of memory storage - in many cases tens or hundreds of gigabytes. The aggregate memory space, consisting of both local and remote memory, can be used to provide high-speed file storage for a large portion of file traffic that does not require conventional disk persistence. These files are often short lived and require fast access. In MBFS, memory rather than disk serves as the primary storage location for these files. Disks, tapes, writable CD's, and other high-latency devices are relegated to the role of archival storage, out of the critical path of most file operations.

MBFS supports a continuum of persistence guarantees and allows applications to select the desired persistence on a per-file basis. Weaker persistence guarantees result in better performance than strong persistence guarantees, allowing applications with weak persistence requirements to avoid the performance penalties and overheads of conventional disk-persistent models.

Because systems cannot guarantee persistence (data will not be lost under any circumstance), they actually only guarantee "the probability of persistence" or "the probability the data will not be lost." In conventional file systems, persistence is defined as (1-(probability of disk failure)) , whereas MBFS supports any persistence probability. The difficulty in supporting such a continuum lies in an application's need to know exactly what each position in the continuum means. For example, what does it mean to select a persistence at the midpoint of the continuum, halfway between "will lose the data immediately" and "will keep multiple almost indestructible copies"? The midpoint definition depends on the the devices used to store the data, their failure characteristics, and if or when data is archived to more persistent storage. Also, the same persistence probability can be implemented different ways. For example, storing data in two NVRAM's may have an equivalent persistence probability to storing the data on an inexpensive disk. Exposing applications to the details of the storage devices and their failure rates is undesirable as it complicates the file system interface and ties the application to the environment, thereby limiting application portability. As a result, MBFS attempts to hide these details from the application.

4.1 Storage Hierarchy

To clarify the "guarantees" provided at different settings of the persistence spectrum without binding the application to a specific environment or set of storage devices, MBFS implements the continuum, in part, with a *logical storage hierarchy.* The hierarchy is defined by N levels:

1. **LM (Local Memory storage):** very high-speed volatile storage located on the machine creating the file.
2. **LCM (Loosely Coupled Memory storage):** high-speed volatile storage consisting of the idle memory space available across the system.
3. **-N DA (Distributed Archival storage):** slower speed stable storage space located across the system.

Logically, decreasing levels of the hierarchy are characterized by stronger persistence, larger storage capacity, and slower access times. The LM level is simply locally addressable memory (whether on or off CPU). The LCM level combines the idle memory of machines throughout the system into a loosely coupled, and constantly changing, storage space. The DA level may actually consist of any number of sub-levels (denoted $DA_1, DA_2, ..., DA_n$) each of increasing persistence (or capacity) and decreasing performance. LM data will be lost if the current machine crashes or loses power. LCM data has the potential to be lost if one or more machines crash or lose power. DA data is guaranteed to survive power outages and machine crashes. Replication and error correction are provided at the LCM and DA levels to improve the persistence offered by those levels.

Each level of the logical MBFS hierarchy is ultimately implemented by a physical storage device. LM is implemented using standard RAM on the local machine and LCM using the idle memory of workstations throughout the network. The DA sub-levels must be mapped to some organization of the available archival storage devices in the system. The system administrator is expected to define the mapping via a system configuration file. For example, DA-1 might be mapped to the distributed disk system while DA-2 is mapped to the distributed tape system.

Because applications are written using the logical hierarchy, they can be run in any environment, regardless of the mapping. The persistence guarantees provided by the three main levels of the hierarchy (LM, LCM, DA_1) are well defined. In general, applications can use the other layers of the DA to achieve higher persistence guarantees, without knowing the exact details of the persistence guaranteed; only that it is better. For applications that want to change their storage behavior based on the characteristics of the current environment, the details of each DA's persistence guarantees, such as the expected mean-time-till-failure, can be obtained via a **stat()** call to the file system. Thus, MBFS makes the layering abstraction explicit while hiding the details of the devices used to implement it. Applications can control persistence with or without exact knowledge of the characteristics of the hardware used to implement it.

Once the desired persistence level has been selected, MBFS's loosely coupled memory system uses an addressing algorithm to distribute data to idle machines and employs a migration algorithm to move data off machines that change from idle to active. The details of the addressing and migration algorithms can be found in [15, 14] and are also used by the archival storage levels. Finally, MBFS provides whole-file consistency via callbacks similar to Andrew[19] and a Unix security and protection model.

4.2 File Persistence

The continuity of the persistence spectrum is provided by a set of per-file *time constraints* (one for each level of the storage hierarchy) that specify the *maximum* amount of time that modified data may reside at each level before being archived to the next level of the storage hierarchy. Since each level of the storage hierarchy has increasing persistence guarantees, the sooner data is archived to the next level, the lower its possibility of loss. Conversely, if data plans to linger in the higher levels, the chance of that data surviving decreases. A daemon process tracks modified data at each storage level, archiving the data to the next lower level within the prescribed amount of time. When data is archived to the next level, it continues to exist at the current level unless space becomes scarce.

Conceptually, as the cost of recreating the data increases, so should the file's persistence level. For example, while a file may be relatively easy to regenerate (for example, an executable compiled from sources) and may never need to hit stable storage, moving it to stable storage after some period of time can prevent the need for the data to be regenerated later (for example, after a power failure). If a file is highly active (i.e., being written frequently), such as an object code file (.o), the time delay prevents the intermediate versions of the file from being sent to lower levels of the hierarchy but will ultimately archive the data at some point in the future, after the active period has finished.

Although archiving of data to lower levels of the hierarchy occurs at predefined time intervals, there are several other events that can cause the archival process to occur earlier than the maximum time duration. First, the daemon may send the data to the next level sooner than required if the archiving can be done without adversely affecting other processes in the system. Most networks of workstations experience an inactive period each night. Consequently, modified files are often archived through all the levels at the end of each day regardless of how long their respective time constraints are. Second, some storage levels such as LM and LCM are limited in capacity. At such levels, modified data may need to be sent to the next level early to make room for new data. Lastly, a modified file may be archived to stable storage (DA level) earlier than required if one of the file's dependencies is about to be changed (see Section 6).

5 MBFS Interface

In order to support variable persistence guarantees, we designed and implemented a new file system abstraction with two major extensions to conventional file system interfaces: a *storage abstraction extension* and a *reconstruction extension.* The storage abstraction extension allows programmers to specify the volatility of a file. MBFS stores the low-level volatility specification as part of the file's metadata and uses callbacks to maintain metadata and block consistency. The reconstruction extensions allows the system to automatically recreate most data that has been lost. The reconstruction extension is described in Section 6.

Although the MBFS logical storage hierarchy allows applications to specify the desired persistence, programmers will not use the system or will not make effective use of the system if the interface is too complex or difficult to select the correct persistence/performance tradeoff. While we expect many performance critical applications (for example, compilers, typesetters, web browsers, archival and compression tools, FTP) will be modified to optimize their performance using the new file system abstraction, we would like to offer MBFS's features to existing applications without the need to change them. In short, we would like to simplify the interface without sacrificing functionality. To that end, we use a multi-level design: a kernel-level raw interface that provides full control over the MBFS logical storage hierarchy and programmer-level interfaces that build on the raw level to provide an easy-to-use interface to applications. The following describes the raw *kernel-level interface* and two programmer-level interfaces, the *category interface* and the *filename matching interface.* Note that additional interface libraries (such as file mean-time-to-failure) could also be implemented on top of MBFS's raw interface. Each library is responsible for mapping their respective volatility specifications to the low-level storage details that the kernel interface expects. Although we describe our extensions as enhancements to the conventional file system abstraction, the extension could also be cleanly incorporated into emerging extensible file system standards such as the Scalable I/O Initiative's proposed PFS[8].

5.1 Kernel Interface

The kernel interface provides applications and advanced programmers with complete control over the lowest-level details of MBFS's persistence specification. Applications can select the desired storage level and the amount of time data may reside at a level before being archived. Applications must be rewritten to take advantage of this interface. For this interface, only the open() routines are modified to provide volatility specification and to support the reconstruction extension.[1] Consequently, a file's volatility specification can be changed *only* when a file is opened. The open() call introduces two new parameters: one for specifying the volatility and one for specifying the reconstruction extension. The reconstruction parameter is discussed in Section 6.1.

The volatility specification defines a file's persistence requirements at various points in its lifetime. The volatility specification consists of a *count* and a variable size array of *mbfs_storage_level structures* described below. Each entry in the array corresponds to a level in the logical storage hierarchy (LM, LCM, $DA_1, ..., DA_n$). A NULL volatility specification is replaced by a *default volatility specification* defined by a system or user configuration file. The default volatility specification is typically loaded into the environment by the login shell or read by a call to a C-level library.

```
typedef struct {
    struct timeval   time_till_next_level;
    void            *replication_type;
} mbfs_storage_level;
```

time_till_next_level: Specifies the maximum amount of time that newly written or modified data can reside at this level without being archived to the next level of the hierarchy. Values greater than or equal to 0 mean that write() operations to this level will return immediately and the newly written data will be archived to the next level within time_till_next_level time. Two special values of time_till_next_level can be used to block future write() and close() operations. UNTIL_CLOSE (-1) blocks the close operation until all file blocks reach the next level. BEFORE_WRITE (-2) blocks subsequent write() operations at this level until the data reaches the next level.

replication_type: This is used to specify the type of replication to use at this level.

For increased persistence, availability, and performance, MBFS supports data replication and striping as specified by the *replication_type* field. The replication_type field defines the type and degree of replication used for that level as defined by the following structure:

[1] If the system supports other file open or file creation system calls (for example, the Unix creat() call), these calls also need to be modified to include a volatility and reconstruction extension

```
typedef struct {
  int  Type;
  int  Degree;
} mbfs_replication;
```

Type: A literal corresponding to the desired form of replication selected from SINGLE_COPY, MIRRORING, and STRIPING [22]. SINGLE_COPY provides no replication. MIRRORING saves multiple copies on different machines. STRIPING distributes a single file and check bits across multiple machines. The default is SINGLE_COPY.

Degree: The number of machines to replicate the data on. If *Type* is SINGLE_COPY this field is ignored. If MIRRORING, it defines the number of copies. If STRIPING, it defines the size of the stripe group.

Mirroring and striping increase reliability by ensuring data persists across single machine or disk failures. Because unexpected machine failures are not uncommon in a distributed system (for example, the OS crashes, a user accidentally or intentionally reboots a machine, user accidentally unplugs machine), replication at the LCM level greatly increases the probability LCM data will survive these common failures.

The kernel-level interface also requires modifications to the system calls used to obtain a file's status or a file system's configuration information (for example, stat() and statvfs() in Solaris). For applications requiring complete knowledge of the environment, MBFS returns information about a file's persistence requirement, reconstruction information, or the estimated mean time to failure of each of the DA levels based on manufacturer's specifications. The raw kernel-level interface provides full control over a file's persistence, but this control comes at the price of elegance and requires that the application provide a substantial amount of detailed information on each open() call.

5.2 Category Interface

To simplify the task of specifying a persistence guarantee, the MBFS interface includes a user-level library called the *category interface*. The premise of the category interface is that many files have similar persistence requirements that can be classified into persistence categories. Thus, the category interface allows applications to select the category which most resembles the file to be created. Category names are predefined or user-defined ASCII character strings that are specified in the open() call. The open() call optionally also takes a reconstruction parameter like the kernel interface.

The category library maps category names to full volatility specifications and then invokes the raw kernel-level interface. The mapping is stored in a process' environment variables which are typically loaded at login from a system or user-specific category configuration file. The environment variable is a list of (category name, volatility specification) pairs.

The system configuration file must minimally define the categories listed below to ensure portability of applications. Programmers and applications may also create any number of additional custom categories. The (minimal) system categories are divided into two sets. The first set defines categories based on the class of applications that use the file. The second set defines categories that span the persistence continuum. The continuum categories are useful for files that do not obviously fall into any of the predefined application classes. The application categories are:

EDITED: Files that are manually created or manually edited and typically require strong persistence (for example, source code files, email messages, text documents, word processing documents).

GENERATED: Files generated as output from programs that require very weak persistence because they can be easily recreated (for example, object files, temporary or intermediate file formats such as *.aux, *.dvi, *.log, and executables generated from source code).

MULTIMEDIA: Video, audio, and image files that are down-loaded or copied (as opposed to edited or captured multimedia data) such as gif, jpeg, mpeg, or wav files.

COLLECTION: A collection of files (archive) create from other files in the system or down-load (for example, *.Z, *.gz, *.tar, *.zip)

DATABASE: Database files often of large size, requiring strong persistence, high-availability, and top performance.

Categories that span the persistence spectrum are (from most volatile to least volatile):

DISPOSABLE: Data that is to be immediately discarded (such as /dev/null).

TEMPORARY: Temporary files that can be discarded if necessary and which will not reach DA_1.

EVENTUAL: Data can be easily recreated but should reach DA_1 if it lives long enough (several hours or more).

SOMETIME: Reconstructable data that is likely to be modified or deleted soon. If the data is not modified soon, the data should be archived to DA_1 relatively soon to minimize the need for reconstruction (repeatedly generated object files).

SOON: Data that should be sent to DA_1 as soon as possible, but it is not a disaster if the data is lost within a few seconds of the write.

SAFE: Data is guaranteed to be written to DA_1 before the write operation completes.

ROBUST: Data is stored at two or more DA levels.

PARANOID: Data is replicated at multiple DA levels of the storage hierarchy.

5.3 Filename Matching Interface

The filename matching interface does not require any changes or extensions to the conventional `open()` operation in order to obtain variable persistence guarantees. Instead, it determines the volatility specification from the filename. The filename matching interface allows the user to define categories of files based on filenames. Consequently, applications (or existing software libraries) need not be rewritten to use the filename matching interface. To take advantage of MBFS's variable persistence, applications can either be re-linked or can install an `open` system call hook that intercepts all `open` operations and redirects them to the filename matching library. This allows legacy applications to benefit from MBFS's variable persistence model without being modified in any way.

The filename matching library maps filenames to full volatility specifications needed by the kernel. Like the category interface, the mapping is stored in environment variables loaded from system or user-defined configuration files. The mapping consists of (regular expression, volatility specification) pairs. At `open` time, the library matches the filename being opened to a regular expression in the environment. If successful, the library uses the associated volatility specification, otherwise a default persistence is assigned.

5.4 Metadata and Directories

In most file systems, write() operations modify both the file data and the file's metadata (for example, file size, modification time, access time). Although it is possible to allow separate persistence guarantees for a file's metadata and data, if either the metadata or the data is lost, the file becomes unusable. Moreover, separate volatility specifications only complicates the file system abstraction. Consequently, MBFS's volatility specifications apply to both the file's data and its metadata.

A similar problem arises is determining how the volatility for directories is specified. In MBFS, directory volatility definitions differ from file volatility in two important ways.

First, all modifications to directory information (*e.g.,* file create/delete/modify operations) must reach the LCM immediately so that all clients have a consistent view of directory information. Only the metadata needs to be sent to LCM. The file's modified data can stay in the machine's LM if the file's metadata was sent to the LCM and a callback is registered with the LCM, and the file's LM timeout has not occurred.

Second, a directory inherits its volatility specification from the most persistent file created, deleted, or modified in the directory since the last time the directory was archived. If a file is created with a volatility specification that is "more persistent" than the directory's current volatility specification, the directory's specification must be dynamically upgraded. If the directory was lost and the directory's persistence wasn't greater than or equal to its most persistent file, the file would be lost from the directory (even if the file's data is not lost). Once the directory is archived to level N, the directory's volatility specification for level N can be reset to infinity. This produces optimal directory performance. Assigning stronger persistence guarantees to directories than the files they contain degrades performance and wastes resources because of the unnecessary persistence costs.

6 Reconstruction

The volatile nature of MBFS may result in lost file data due to unexpected failures. To encourage programmers and applications to select weaker persistence guarantees, MBFS aids in the process of restoring lost data. It should be noted that restoration is a mechanism that is applicable to many, but not necessarily all, files. In other words, the restoration mechanism cannot guarantee successful reconstruction. Some aspects of the environment (for example, date, time, software versions, access to the network) in which the program was originally run may be impossible to capture or recreate and may render reconstruction impossible. In these few cases, manual reconstruction or disk persistence should be used. However, if data is committed to stable storage within some reasonably short period of time, such as 24 hours, the environment is unlikely to change significantly during the volatile interval when reconstruction may be necessary. Therefore, we believe a large percentage of performance-critical files can use the reconstruction mechanism to achieve improved performance without fear of data loss or manual restoration.

MBFS supports two methods for restoring lost data: *manual reconstruction* and *automatic reconstruction*. The most obvious and painful method is manual reconstruction in which users manually reruns the program that originally created the lost file. The second approach relieves the user of this burden by having the file system automatically reconstruct lost files after system failures. Prior to a failure, the user or application specifies the information needed to reconstruct the volatile file. Using this information, the file system automatically reconstructs the lost data by re-executing the appropriate program in a correct environment.

6.1 The Reconstruction Extension

The MBFS open() call in the kernel, category, and filename matching interfaces optionally support a reconstruction parameter defined by the following structure:

```
typedef struct {
    char *reconstruction_rule;
    envp *environment[];
    int   num_dependencies;
    char *dependencies[];
    int   data_freshness;
    Bool  reconstruct_immediately;
} mbfs_reconstruction;
```

reconstruction_rule: A string containing the command-line that the system should execute to reconstruct the file.

environment: An array of environment variables and values to be set in the shell before invocation of this file's reconstruction rule. If NULL, the current environment in assumed.

data_freshness: Specifies how "up-to-date" the contents should be with respect to it's dependencies:

- LATEST_DATA - This corresponds to a guarantee that the file will always contain the newest possible data. The file should be reconstructed if any of its dependencies have changed. This feature can be used to implement intentional files.
- VERSION_DATA - The system guarantees that some version of the file data exists, not necessarily the latest. Unlike LATEST_DATA, the file's contents are not regenerated when a dependency changes. Only if the data is lost will the data be regenerated. If a dependency is deleted, the reconstruction rule cannot be executed, so the system immediately archives the data to an DA level to ensure its persistence.
- OLD_DATA - This guarantees that the file's contents will be based on versions of the dependencies as they existed at the time the file was created. Before the system will allow a dependency to be changed or deleted, the system will ensure the data reaches an DA level because reconstruction might produce different data.

reconstruct_immediately: A boolean specifying when the system should execute the file's reconstruction rule.

- TRUE - The system will invoke the reconstruction rule as soon as lost file data is detected or, if LATEST_DATA is selected, as soon as a dependency is changed.
- FALSE - The system may postpone the invocation of the reconstruction rule for any amount of time up until the file is referenced. This is the default.

num_dependencies: The number of entries in the dependency array.

dependencies: An array of entries corresponding to the MBFS files on which the given file is dependent.

With the inclusion of dependencies and the LATEST_DATA freshness quality, the reconstruction extension naturally provides an automatic form of the

Unix “make” utility and provides a form of intensional files. While an automatic “make” is useful in some circumstances, users may still invoke the standard make utility manually.

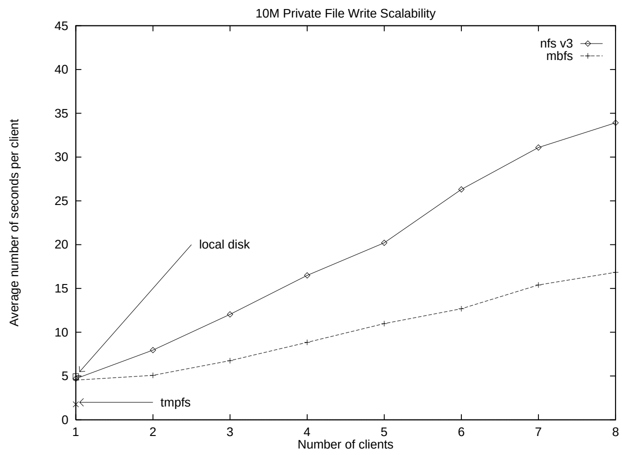
6.2 Reconstruction Environment

The system must determine on which machine the rule should be run. Remembering the machine that created the file is useful but not sufficient. First, the original machine may be down or heavily load at reconstruction time. Second, the original machine’s environment may have changed since the file was created. Before a reconstruction rule is saved, MBFS collects the architecture and operating system of the machine and stores them with the reconstruction rule. At reconstruction time, the system searches for machines satisfying both architecture and OS, and chooses one of them on which to run the reconstruction rule. If no machine is found, an error message is sent to the owner of the file saying the file could not be reconstructed.

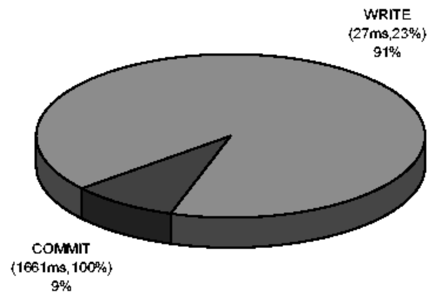
7 MBFS Prototype

To quantify the potential performance gains of a variable persistence model, we implemented a prototype MBFS system with support for LM, LCM, and DA_1 (via disks). The prototype runs on Solaris and Linux systems. The current prototype does not yet support the reconstruction extension. We then ran tests using five distinct workloads to evaluate which data can take advantage of variable persistence and to quantify the performance improvements that should be expected for each type of workload. Our tests compared MBFS against NFS version 3, UFS (local disk), and tmpfs. Only one point for UFS and tmpfs per graph are provided since distributed scalability is not an issue associated with localized file systems. In all but the edit test, no file data required disk persistence and typical performance improvements were in the range of three to seven times faster than NFS v3 with some tests almost two orders of magnitude faster.

The MBFS server runs as a multi-threaded user-level Unix process, experiencing standard user-level overheads (as opposed to the NFS server which is in-kernel). Putting the MBFS in-kernel and using zero-copy network buffer techniques would only enhance MBFS’s performance. The MBFS server runs on both Solaris and Linux. An MBFS server runs on each machine in the system and implements the



(a) Run-time scalability

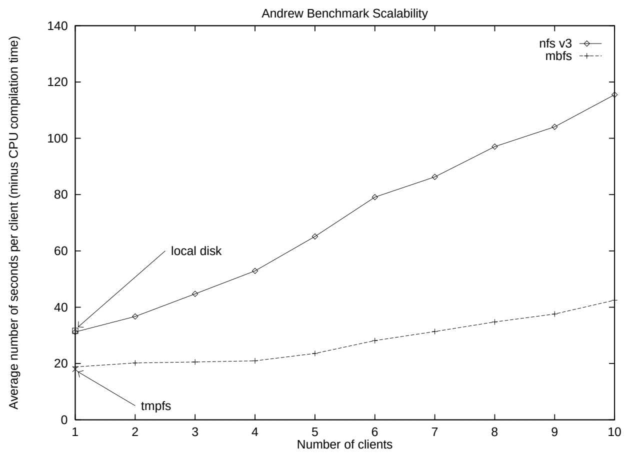


(b) Operation Improvement

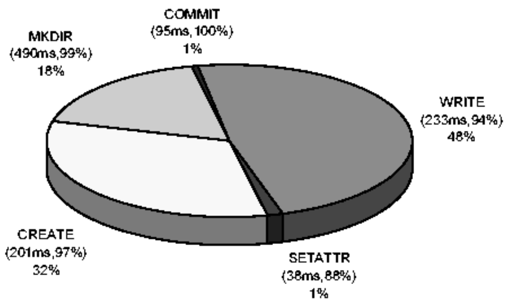
Figure 1: Results of the large write throughput test.

LCM, and DA_1 storage levels. The LCM component monitors the idle resources, accepts LCM storage/retrieval requests, and migrates data to/from other servers as described in [14]. Similarly, the DA_1 component uses local disks for stable storage and employs an addressing algorithm similar to that used by the LCM. To eliminate the improvements resulting from multiple servers (parallelism) and instead focus on improvements caused by variable persistence, we only ran a single server when comparing to NFS.

The MBFS client is implemented as a new file system type at the UNIX VFS layer. Solaris and Linux implementations currently exist. MBFS clients redirect VFS operations to the LCM system or service them from the local cache. The system currently uses the filename-matching interface. The current implementation does not yet support callbacks, so the time_till_next_level of LM must be 0 so data is flushed to the LCM to ensure consistency. The sys-

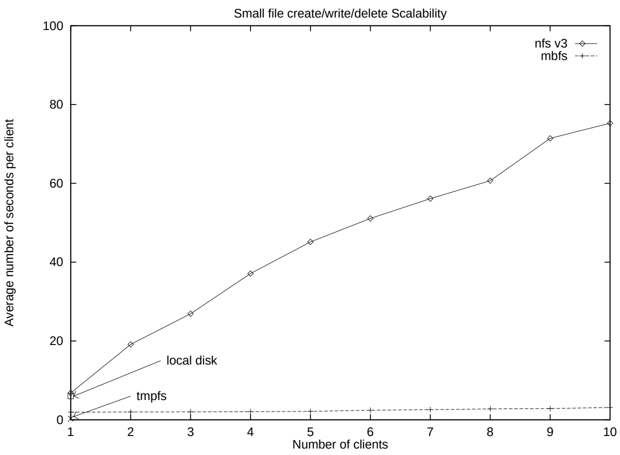


(a) Run-time scalability

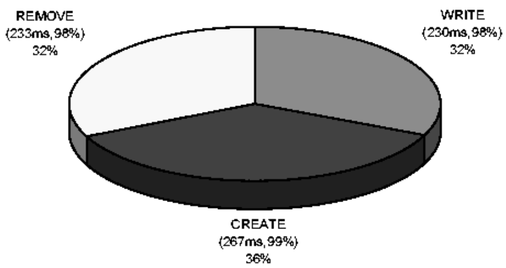


(b) Operation Improvement

Figure 2: Results of the Andrew Benchmark.



(a) Run-time scalability



(b) Operation Improvement

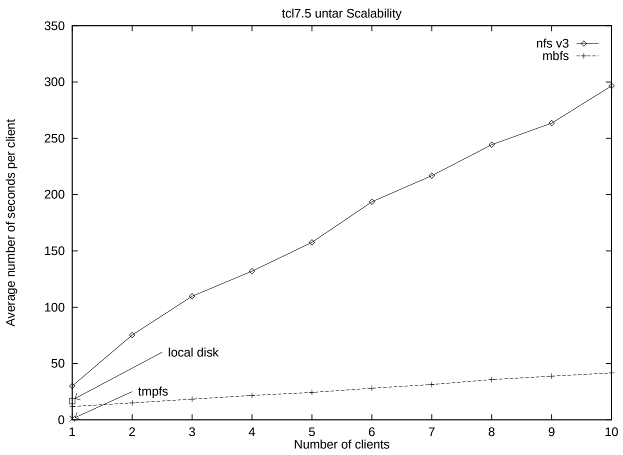
Figure 3: Results of the small file create/write/delete test.

tem currently provides the same consistency guarantees as NFS. Callbacks would improve the MBFS results shown here because file data could stay in the LM without going over the network. Replication is not currently supported by the servers. Communication with the LCM is via UDP using a simple request-reply protocol with timeouts and retransmissions.

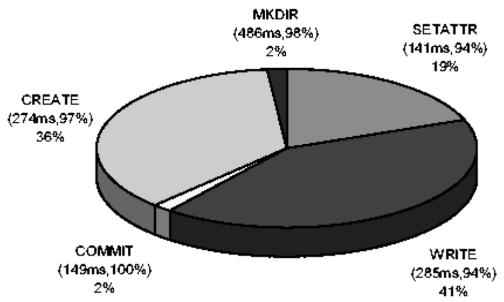
7.1 Solaris Results

The MBFS and NFS servers were run on a SPARC 20 with 128 MB of memory and a Seagate ST31200W disk with a 10ms average access time. We ran up to 10 simultaneous client machines on each server. Each client was a SPARC 20 with 64 MB of memory and a 10ms Seagate local disk (for the UFS tests). Tmpfs tests used the standard UNIX temp file system. All machines were connected by a 100 Mbps Ethernet and tests were run during the evening hours when the network was lightly loaded. We computed confidence intervals for each test. The confidence intervals were typically within 5% of the total runtime and as a result are not shown in the graphs.

Five tests with different workloads were run: (1) write throughput for large files, (2) small file create/write/delete throughput, (3) a mixture of file and directory creation with large and small files, (4) a manual edit test, and (5) the Andrew Benchmark. Each of the tests focused on write traffic to files or directories. Read traffic was minimal and did not contribute to any speedups observed. Each test was run several times and averaged. The results show two graphs: (1) a line-graph illustrating the scalability of MBFS versus NFS in terms of total runtime, and (2) a pie-chart of the 10-client test describing how much each operation contributed to the overall speedup. Each slice of the pie-chart depicts the percentage of runtime improvement caused by that operation. The



(a) Run-time scalability



(b) Operation Improvement

Figure 4: Results of the untar test.

numbers in parenthesis list the average speedup over NSF for the operation. The first number in the pair gives the absolutes speedup in milliseconds and the second number gives the relative speedup in terms of a percentage ($\frac{NFStime-MBFStime}{NFStime}$).

To measure the baseline write performance, we used a large-write throughput test. Each client creates a copy of an existing 10 MB file. Because the new file is a copy, disk persistence is not required. The original 10 MB files is preloaded into the client cache to eliminate read traffic from the test. Figure 1(a) shows that MBFS performs better than NFS (despite NFS's use of asynchronous writes) because of contention at the disk. Note that as the number of clients increases, the server's maximum receive rate quickly becomes the bottleneck in this test. Figure 1(b) shows that 91% of the overall runtime savings were due to improved write operations, with 9% of the improvement arising from the fact the MBFS does not issue a final commit operation. In other words, even when writes are asynchronous, the server response time is significantly slower than MBFS memory-only writes.

Figure 2(a) shows the results of the Andrew Benchmark that tests several aspects of file system performance: making directories, copying files to those directories, scanning the directories, read the file contents, and perform a compilation. To more accurately isolate the file system performance, we subtracted the CPU time used by the compiler during the compilation phase. Because all the data and directories are generated or copied, none of the writes required disk persistence. Improvements range from 40% with one client (a perceivable improvement to the user) to as much as 64%. Figure 2(b) clearly illustrates that all savings come from operations that typically require disk persistence: mkdir, create, write, setattr, and commit.

Figure 3(a) shows the results of the small file test where each client repeatedly (100 times) creates a file, writes 1K of data, and deletes the file. The test measures directory and metadata performance and models applications that generate several small files and then deletes them (for example, compilers). The results are extremely impressive with MBFS processing 313 files per second compared with NFS's 13 per second at 10 clients.

Figures 4(a) and 4(b) show the results of untaring the TCL 7.5 source code. Untar is an I/O bound program that creates multiple directories and writes a variety of file sizes (all easily recreatable from the tar file). Again, the results are perceivably faster to the user.

In all the tests MBFS outperformed both NFS and UFS (local disks). More surprising is how often MBFS challenged tmpfs performance despite the lack of callbacks in the current implementation. Similar performance tests were performed in a Linux environment with even better results since Linux's NFS version 2 does not support asynchronous writes.

Finally, we ran an edit test in which we edited various files, composed email messages, and created web pages. All files required disk persistence. As expected there was no, or minimal, performance gains.

8 Related Work

Several researchers have investigate high-speed persistent storage techniques to improve write performance. Remote memory systems have also been proposed as a high-speed file caches. The following briefly describes this related work.

8.1 Persistence vs. Performance

The performance penalties of a disk-persistent file storage model are well known and have been addressed by several file systems [3, 18, 6, 9, 5, 25, 20]. Unlike the application-aware persistence design we propose, the following systems have attempted to improve performance without changing the conventional one-size-fits-all disk-persistence file system abstraction.

The xFS file system [3] attempts to improve write performance by distributing data storage across multiple server disks (i.e., a software RAID [22]) using log-based striping similar to Zebra [16]. To maintain disk-persistence guarantees, all data is written to the distributed disk storage system. xFS uses metadata managers to provide scalable access to metadata and maintain cache consistency. This approach is particularly useful for large writes but does little for small writes.

The Harp[18] and RIO[6] file systems take an approach similar to the one used by Derby. High-speed disk persistence is provided by outfitting machines with UPS's to provide non-volatile memory storage. Harp also supported replication to improve availability. However, Harp used dedicated servers as opposed to Derby which uses the idle memory of any machine. RIO uses non-volatile memory to recover from machine failures and could be used to implement the non-volatile storage of Harp or Derby. Alternatively it could be used to implement memory-based writes on an NFS file server, but would not take advantage of the idle aggregate remote memory storage space like DERBY. Also, because RIO does not flush data to disk (unless memory is exhausted), UPS failures may result in large data losses. Because Derby only uses UPS's as temporary persistent storage, UPS failures are less catastrophic.

Other systems have introduced the concept of delayed writes (asynchronous writes) to remove the disk from the critical path. For example, conventional Unix file systems use a 30-second delayed-write policy to improve write performance but create the potential for lost data. Similarly, Cortes et al. [9] describe a distributed file system called PAFS that uses a cooperative cache and has cache servers and disk servers. To remove disks from the critical path, PAFS performs aggressive prefetching and immediately acknowledges file modifications once they are stored in the memory of a cache server. Cache servers use a UNIX-like 30 second delayed write policy at which point they send the data to a disk server. The Sprite [20] file system assumed very large dedicated memory file servers and wrote all data to disk on a delayed basis creating the potential for lost data. A special call to flush data to disk could be invoked for applications worried about persistence. NFS version 3 [5] introduced asynchronous writes whereby the NFS server would place asynchronous writes in memory, acknowledge the write requests, and immediately schedule the new data block to be written to disk. Before an NFS client can close a modified file, the client would issue a commit operation to the NFS server. The NFS server will not acknowledge the commit until all the file's modified blocks have been committed to disk. The Bullet file server [25] provides a "Paranoia Factor" which when set to zero provides the equivalent of asynchronous writes. For other values, N, of the paranoia factor, the Bullet file server would replicate the file on N disks. Both NFS and Bullet write all data to disk, even short lived files. Tmpfs implements a ramdisk and makes no attempt to write data to disk. Tmpfs users understand that tmpfs files are volatile and may be lost at any time.

8.2 Remote Memory Storage

A significant amount of the large aggregate memory capacity of a network of workstations is often idle. Off-the-shelf systems provide access to this idle remote memory an order of magnitude faster than disk latencies. Therefore, many systems have been developed to make use of idle memory capacity, primarily for paging and caching.

Comer and Griffioen [7] introduced the *remote memory model* in which client machines that exhaust their local memory capacity paged to one of a set of dedicated *remote memory servers*. Each client's memory was private and inaccessible even if it was idle. Data migration between servers was not supported.

Felten and Zahorjan [11] enhanced the remote memory model to use any idle machine. Idle client machines advertise their available memory to a centralized registry. Active clients randomly chose one idle machine to page to. Like [7], data was not migrated among idle clients.

Dahlin et al. [10] describe an N-Chance Forwarding algorithm for a cooperative cache in which the file caches of many client machines are coordinated to form a global cache. N-Chance Forwarding tries to keep as many different blocks in global memory as it can by showing a preference for *singlets* (single copies of a block) over multiple copies. The cache only stores *clean* (unmodified) blocks. Thus, all file block modifications are written to the file server's disk. A similar approach is used in xFS[3] and PAFS[9].

Feeley et al. [13] describe the Global Memory Service (GMS) system that integrates global memory management at the lowest level of the operating system enabling all system and higher-level software, including VM and the file cache, to make use of global memory. GMS uses per node page age information to approximate global LRU on a cooperative cache. Like N-Chance Forwarding, GMS's only stores *clean* file blocks and so all file writes must hit the file server's disk.

Hartman and Sarkar [23] present a *hint-based* cooperative caching algorithm. Previous work such as N-Chance Forwarding [10] and GMS [13] maintain *facts* about the location of each block in the cooperative cache. Although block location hints may be incorrect, the low overhead needed to maintain hints outweighs the costs of recovering from incorrect hints. All file modifications are written to the file server's disk so that if a hint is missing or incorrect, a client can always retrieve a block from the server. Using hints, block migration is done in a manner similar to that of GMS [13]. Unlike MBFS, none of the above systems considers a client's CPU or memory load when deciding the movement or replacement of pages.

Franklin et al. [12] use remote memory to cache distributed database records and move data around using an algorithm similar in nature to that of N-chance forwarding. Client load was not considered by the data migration mechanism.

The Trapeze network interface [2] provides an additional order of magnitude improvement in remote memory latencies versus disk latencies by improving the network subsystem.

8.3 Application Aware Storage

The Scalable I/O Initiative (SIO), introduced a new file system interface [8] for parallel file systems. The interface enables direct client control over client caching algorithms, file access hints, and disk layout. As such, it is a suitable platform for integrating our proposed variable persistence model.

9 Summary

In this paper we presented a new file system abstraction for memory-based file management systems. The abstraction is unique in the fact that it allows applications or users to control file persistence on a per-file basis. Applications that can tolerate weak persistence can achieve substantial performance improvements by selecting memory-based storage. The new abstraction has two major extensions to conventional file systems abstractions. First, applications can define a file's persistence requirements. Second, applications can specify rules to reconstruct file data in the unlikely event that it is lost. Analysis of current file systems indicates that a large percentage of write traffic can benefit from weak persistence. To support large amounts of data with weak persistence guarantees, we developed a high-speed loosely-coupled memory storage system that utilizes the idle memory in a network of workstations. Our prototype implementation of the MBFS system running on Solaris and Linux systems shows applications speedups of an order of magnitude or more.

10 Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. We would also like to thank our shepherd, Darrell Long, for his insightful suggestions and comments. Also thanks to the CS570 class for using the system during the fall 1998 semester.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proceedings of 1997 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Seattle, June 1997.

[2] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 143–154, Berkeley, USA, June 15–19 1998. USENIX Association.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, October 1991.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification, June 1995. RFC 1813.

[6] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating systems crashes. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Cambridge, Massachusetts, 1–5 Oct. 1996. ACM Press.

[7] D. Comer and J. Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *The Proceedings of the 1990 Summer USENIX Conference*, pages 127–136. USENIX Association, June 1990.

[8] P. Corbett, J. Prost, J. Zelenka, C. Demetriou, E. Riedel, G. Gibson, E. Felten, K. Li, Y. Chen, L. Peterson, J. Hartman, B. Bershad, and A. Wolman. Proposal for a Common Parallel File System Programming Interface Part I Version 0.62, August 1996.

[9] T. Cortes, S. Girona, and J. Labarta. Avoiding the cache-coherence problem in a parallel/distributed file system. In *Proceedings of the High-Performace Computing and Networking*, pages 860–869, Apr. 1997.

[10] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remove Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[11] E. W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.

[12] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *18th International Conference on Very Large Data Bases*, 1992.

[13] M. Freeley, W. Morgan, F. Pighin, A.Karlin, and H. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedins of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[14] J. Griffioen, T. Anderson, and Y. Breitbart. A Dynamic Migration Algorithm for a Distributed Memory-Based File Management System. In *Proceedings of the IEEE 7th International Workshop on Research Issues in Data Engineering*, April 1997.

[15] J. Griffioen, R. Vingralek, T. Anderson, and Y. Breitbart. Derby: A Memory Management System for Distributed Main Memory Databases. In *The Proceedings of the IEEE 6th International Workshop on Research Issues in Data Engineering (RIDE '96)*, February 1996.

[16] J. Hartman and J. Ousterhout. Zebra: A Striped Network File System. In *Proceedings of the Usenix File System Workshop*, pages 71–78, May 1992.

[17] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A High Performance Main Memory Storage Manager. In *Proceedings of the VLDB Conference*, 1994.

[18] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, October 1991.

[19] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A Distributed Personal Computing Environment. *CACM*, 29:184–201, March 1986.

[20] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, Feb. 1988.

[21] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[22] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD 88*, pages 109–116, June 1988.

[23] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 35–46, October 1996.

[24] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In B. Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 146–160, New York, NY, USA, Dec. 1993. ACM Press.

[25] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The Design of a High Performance File Server. *Proceedings of the IEEE 9th International Conference on Distributed Computing Systems*, 1989.