MAY 1989

# WRL Research Report 89/5

# Spritely NFS: Implementation and Performance of Cache-Consistency Protocols

*V. Srinivasan*
*Jeffrey C. Mogul*

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

| | |
|---|---|
| Digital E-net: | `DECWRL::WRL-TECHREPORTS` |
| DARPA Internet: | `WRL-Techreports@decwrl.dec.com` |
| CSnet: | `WRL-Techreports@decwrl.dec.com` |
| UUCP: | `decwrl!wrl-techreports` |

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

# Spritely NFS: Implementation and Performance of Cache-Consistency Protocols

**V. Srinivasan**

University of Wisconsin

**Jeffrey C. Mogul**

Digital Equipment Corporation
Western Research Laboratory

**May, 1989**

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

## Abstract

**File caching is essential to good performance in a distributed system, especially as processor speeds and memory sizes continue to scale rapidly while disk latencies do not. Stateless-server systems, such as NFS, cannot effectively manage client file caches. Stateful systems, such as Sprite, can use explicit cache consistency protocols to achieve high performance, without some of the inconsistencies possible in NFS.**

**By modifying NFS to use the Sprite cache consistency protocols, we isolate the effects of the consistency mechanism from the other features of Sprite. We find dramatic improvements on some, although not all, benchmarks, suggesting that an explicit cache consistency protocol is necessary for both correctness and good performance.**


This report is an expanded version of a paper that has
been submitted for publication elsewhere.

# 1. Introduction

Cache strategies are central to performance, reliability, and correctness of distributed file services. Caching improves performance by avoiding unnecessary disk traffic, network traffic, and server use, but caching implies the potential existence of multiple copies of the same data, and keeping these multiple copies consistent is a challenge. This is especially true when the caches are kept by the clients of a distributed file service, which might be attempting concurrent access to the same file.

Several different cache-consistency strategies are used in existing systems. Two important examples are the NFS [10] and Sprite [5] file system protocols. (''Sprite'' is the name of an entire distributed operating system; we are concerned only with the Sprite file protocols.)

NFS adheres to a stateless-server model and uses a probabilistic, stateless consistency scheme[1]. Sprite maintains server state, and uses an explicit consistency protocol. This protocol allows Sprite to guarantee the semantics of concurrent access by several clients to the same file; in addition, Sprite is alleged to provide better performance than NFS.

We were intrigued by the possibility that, by transplanting the Sprite consistency protocol into the NFS file access protocol, we could transfer some of Sprite's benefits to NFS without seriously compromising the NFS model; this is discussed in detail in section 2. This experiment also helps to isolate the effects of the cache consistency protocols from other differences between NFS and Sprite (for example, the different approaches to file name translation).

It proved to be relatively easy to modify the NFS implementation used in the Ultrix™ operating system to use the Sprite consistency mechanism. We call this system ''Spritely NFS[2].'' In section 3 we describe the specific changes to the NFS protocol, and in section 4 we describe how these changes were implemented in Ultrix.

Performance measurements, presented in section 5, are somewhat ambiguous: depending on the benchmark, Spritely NFS either dramatically outperforms NFS, or performs slightly worse. Our expectation is that in actual use, Spritely NFS should perform moderately better than unmodified NFS. In any event, Spritely NFS guarantees that no two clients will have inconsistent cached copies of a file.

In section 6, we look at some of the issues left unresolved by our experiments. In particular, introducing state into the Spritely NFS server requires more attention to crash recovery than is necessary in unmodified NFS. We also discuss some improvements to Spritely NFS that may further improve performance. Section 7 summarizes our experiments and attempts to draw some generally applicable conclusions.

---

[1]The official NFS protocol specification [12], while requiring a stateless server, says nothing about cache consistency. This specification provides insufficient guidance for producing a workable NFS implementation; the so-called ''reference port'' implementation is what actually defines correct behavior of NFS clients and servers. The reference port does imply a particular cache-consistency scheme.

[2] **sprite·ly** *adj.* [obsolete *spright* (sprite), alteration of *sprite*]: marked by a gay lightness and vivacity: SPIRITED **syn** see LIVELY [13]

## 2. Goals of the experiment

We hoped to answer two questions in our experiment:

1. Are the performance advantages claimed for Sprite really the result of the cache-consistency mechanism, or are they attributable to differences in other features or to implementation quality?

2. Would adding Sprite-like consistency protocols to NFS improve NFS performance, and could this be done without significantly complicating the NFS implementation?

Implicit in these questions is the idea that the central difference between Sprite and NFS is their different approach to cache consistency. Indeed, the protocols are similar in many ways, particularly because both are meant to provide a nearly transparent emulation of the Unix® file system, using servers accessed by remote procedure call (RPC). In both Sprite and NFS, portions of a file may be cached in memory at the client (in contrast to the whole-file caching scheme used in Andrew [2] or Cedar [11]).

### 2.1. NFS consistency model

NFS follows a stateless-server model; the server maintains no state information between RPC requests. This simplifies the server implementation, avoids hard limits on the number of simultaneous clients, and makes server-crash recovery trivial.

Since the server has no record of which clients are currently using a file, however, it cannot make guarantees about cache consistency. An NFS client periodically checks with the server to see if a file has been modified[3]; if so, the client invalidates its cache for that file. The interval between checks is a compromise between performance (frequent checking loads the server and delays the client) and consistency (insufficiently frequent checking may mean that a client uses stale data from its cache).

Since one NFS client has no way of identifying other clients that may be concurrently accessing a file, all of its consistency checks must be made with the file server. The file server, because it is stateless, also does not know which clients are caching a file. Therefore, whenever a client modifies a file, it must immediately communicate the change back to the server. In this way, it limits the potential inconsistency between the server's copy and its own cache to a short period. This ''write-through'' policy also limits the amount of damage caused by a crash; since an NFS server is required to write data onto disk (or other stable storage) before returning from the remote procedure call, the amount of cached information that is vulnerable to loss during a crash is quite limited.

The write-through policy has two distinct disadvantages. First, write-through limits the performance benefits of client-side caching, since a server disk access is done for every write. A surprising number of Unix files (in dynamic terms) have short lifetimes and are never shared by multiple clients [8], and thus need not be kept anywhere but in the client-side cache of the host

[3]The interval between probes in Ultrix varies between 3 and 60 seconds, depending on the recent history of the file.

where they are created. NFS, unfortunately, cannot distinguish between shared and unshared files, and so must treat every file as if it were potentially shared. Both client and server waste effort performing unnecessary write-through operations.

The second disadvantage of a strict write-through policy is that it forces applications to run synchronously with the disk. While an application is waiting for the data to make its way over the network, through the server queues, and onto the disk, it is blocked. The application therefore takes longer to complete than it would if disk writes were performed asynchronously, as they are in the local Unix file system. Especially on single-user workstations, this time is wasted.

Actual NFS client implementations do not always write data synchronously. Instead, a block may be handed to a daemon process, which immediately writes it to the server; the original requesting process does not wait for the write to complete. This modification appears to be necessary to obtain reasonable performance. It loosens the consistency guarantee, however, so an NFS client synchronously finishes all pending write-throughs when a file is closed.

## 2.2. Sprite consistency model

Sprite follows a stateful-server model. Unlike NFS, Sprite has explicit *open* and *close* operations. Since the Unix *open* operation specifies if the application intends to write to the file, by tracking *open* and *close* operations the Sprite file server knows not only which clients are currently using a file, but whether any of them are potentially writers.

This is important because files are seldom ‘‘write-shared’’; that is, seldom do two or more clients simultaneously have a file open that one of them is writing. More typically, either all the clients are doing read-only operations, or a single client ‘‘owns’’ the file while it is being modified. (We refer to these as the ‘‘read-only’’ and ‘‘single-writer’’ cases, or together as ‘‘non-write-shared.’’)

Because a non-write-shared file can be cached at the clients without any danger of inconsistency, the Sprite server responds to each *open* request with an indication of whether it is safe to cache the file. Clients can cache without the periodic consistency checks required in NFS; also, a single-writer client need not do write-throughs, and may even keep the file in its cache for the file’s entire lifetime. (If reliability is more important than performance, an application can use explicit file-flush operations to cause write-through.)

If a file is write-shared, none of the clients (writers *or* readers) are allowed to cache it. For writers, this reverts to the write-through policy of NFS, and provides the same single-copy consistency between a writer and the server. For readers, this is stricter than the NFS mechanism; each read operation goes directly to the server. Thus, the readers are guaranteed consistency with the writers, provided that some other mechanism (such as file locking) serializes the reads and writes. NFS cannot feasibly use such a strict reader-caching policy, since NFS cannot distinguish between the infrequent write-sharing situation when it is beneficial, and the normal non-write-shared case when it is wasteful.

Of course, when multiple clients open a file, they do not all issue their *opens* simultaneously. When a file that is shared read-only by several clients is subsequently opened for write, the

Sprite file server must notify not only the newly-arrived writer, but the existing readers as well, that the file is no longer cachable. (Similarly, when a file is opened first by a single writer and then by another client, the first writer must be told to stop caching its copy and to return all the dirty pages to the server.) In order to perform this notification, the Sprite server must make asynchronous calls to the client, or ‘‘callbacks.’’ Callbacks are the other major difference between the NFS and Sprite protocols (in addition to the explicit *open* and *close* operations).

There are other differences between Sprite and NFS that are not further considered in this paper, although they do have a significant effect on performance. For example, Sprite and NFS take distinctly different approaches to file name translation.

## 2.3. Potential advantages of Sprite

The two main advantages of Sprite over NFS are improved consistency and improved performance. Sprite clients do not periodically probe the server to check for inconsistencies, and need not perform write-through.

The Sprite consistency mechanism, unlike the periodic checks used in NFS, *guarantees* consistency between clients accessing the same file. Thus, in some sense Sprite is more ‘‘correct’’ than NFS. We do not know how important this is in practice: since application writers know that NFS does not provide true consistency, and especially since we believe that write-sharing is infrequent in any case, the lack of consistency in NFS may not be significant. (A more frequent case is ‘‘sequential write-sharing,’’ where the writer closes a file before the reader opens it; NFS provides consistency in this case, but only if the interval between *close* and *open* is longer than the NFS consistency-probe period.) On the other hand, the weakness of NFS consistency may be responsible for the lack of shared-database applications.

In addition to better consistency, Sprite should provide better performance. Sprite avoids the cost of periodic consistency probes, in exchange for the cost of doing explicit *open* and *close* operations. In Sprite, unlike Spritely NFS, the *open* operation is ‘‘piggybacked’’ on the file name translation operation in a way that is not possible with NFS. Even so, in the case where a file is opened, read quickly, and then closed, Sprite would require one more RPC operation than NFS (because NFS would not need to do any consistency probes in such a brief interaction).

Fortunately, Sprite gets other benefits from its consistency protocol besides the reduction in probing. Most important is the ability to do write-back instead of write-through. This improves performance in two cases:

1. An application that alternates computation with disk output (such as the compilation of several modules) can do both in parallel, since Sprite allows the client’s writebacks to proceed asynchronously even across file closes.

2. An application that generates short-lived files (such as a compiler with its intermediate files, or a sort program) need not ever pay the cost of writing them to disk. The client can delay write-back long enough that the file may be deleted before the file is ever written to the server. This becomes more interesting as memory chip sizes, and consequently client file system cache sizes, increase.

These two cases are quite common for typical Unix workloads, and the performance improvements can be dramatic.

If the application mix is right, use of the Sprite consistency mechanism should improve performance over NFS by reducing client-to-server traffic and server disk I/O; the latter is especially important because disk access times are not improving nearly as fast as processor and communication speeds. This in turn should improve client response time, and also increase the number of clients that can actively use a single server (and thus that can actively share a single file system).

One of the advantages of statelessness in NFS is that a given NFS server can handle an arbitrarily large number of clients or files, since it keeps no per-client or per-file state. A Sprite server, on the other hand, cannot serve an infinite number of clients or files, since it keeps information about each recently-active file. That comparison may be illusory: while the NFS server may be able to ‘‘handle’’ an arbitrary number of clients, the Sprite server should be able to provide acceptable service to a larger number of simultaneously active clients.

## 2.4. Implications for crash recovery

Because NFS servers are stateless, server crash recovery in NFS is trivial: the server simply restarts. Client crash recovery is also fairly simple, since all client data is written immediately (as soon as possible) to disk, and synchronously on close[4]. Only crashes that occur between the creation of data by an application (for example, keystrokes to a text editor) and the completion of a file-write RPC cause data loss; this is actually better than the local file system reliability in Unix, where a disk write may be delayed for as much as 30 seconds.

Sprite provides roughly the same protection against client crashes as does a local Unix file system. (An application can always do explicit file flushes to provide crash-resistance, but few existing Unix programs are written this carefully. Also, as in Unix, the write-delay period may be adjusted to reduce the crash-vulnerability window.) Sprite server crash recovery is more complex than in NFS, since the server must reconstruct the state it maintains about which files are open by which clients. Server-crash recovery mechanisms have been implemented for Sprite [14]; they rely on two properties of Sprite:

1. The clients together ‘‘know’’ the consistency state of the file, and the server can reconstruct its state from the clients.
2. The consistency state of the file cannot change while the server is down, or until the server is willing to allow it to change.

Most of the complexity in the recovery mechanism comes in detecting crashes and reboots, rather than in rebuilding state. A reliable crash and reboot detection mechanism is, of course, useful for other purposes besides recovering file server state.

[4]Actually, the reference port of NFS delays writes that do not extend to the end of a block, as a means of optimizing improperly-buffered sequential writes.

### 2.5. Related work

A cache-consistency mechanism roughly intermediate between that of NFS and Sprite has been implemented for the System V Remote File Sharing (RFS) system [1]. As in NFS, clients write-through to the server, so the only possible inconsistency is between the server and readers. RFS is not stateless; clients send *open* and *close* messages to the server, so the server is able to send ‘‘invalidate’’ messages back to clients when their caches must be disabled. Unlike Sprite, however, RFS waits until blocks are actually written before invalidating client caches. As in both Sprite and NFS, version numbers are used to maintain client cache consistency when a file is reopened after being closed. RFS provides the same consistency guarantees as Sprite, but because RFS uses the same write policy as NFS, its performance should be closer to that of NFS.

Both Sprite and RFS use entire files as the unit for consistency. Kent [4] describes a system that maintains consistency on individual file blocks; before a client writes a block, it must acquire ownership of that block. Other clients invalidate cached copies of that block, and only one client at a time can own a block. This system required special hardware to implement the consistency protocol with sufficient performance.

The dogma of statelessness associated with NFS has been broached before. Juszczak [3] shows that because the individual NFS operations are not really idempotent, certain kinds of communication failure can result in incorrect behavior. By adding a small amount of state to the NFS server, he managed to resolve this problem, and also to improve the performance of highly-loaded servers.

## 3. Modifications to the NFS protocol

In this section we describe the modifications to the NFS protocol necessary to support the Sprite consistency protocols. This is the ‘‘Spritely NFS’’ (SNFS) *protocol*; most of the complexity in SNFS is in the *implementation*, described in section 4.

### 3.1. New client-to-server calls

In unmodified NFS [12], all RPC calls are initiated by the client. We added two new calls, *open* and *close*, defined here in the same style as the original NFS specification:

```
NFSPROC_OPEN (file, mdev, writeMode)
                    returns (reply)
        fhandle file;
        unsigned mdev;
        boolean writeMode;
        openopres reply;
```

The `file` argument is a file handle, as provided by the lookup procedure. The `mdev` argument is passed to the server for use in any future callbacks; it is used to help the client identify which remotely-mounted file system a callback applies to. The `writeMode` argument, if ‘‘true,’’ indicates that the client intends to write the file.

The `reply` for the *open* operation uses this structure:

```
typedef union switch (stat status) {
    NFS_OK:
        struct {
            unsigned    cacheVersion;
            unsigned    oldCacheVers;
            boolean     cacheEnabled;
            fattr       attributes;
        }
    default:
        struct {}
} openopres;
```

The `reply.cacheEnabled` field tells the client whether it is allowed to cache data for this file. The server keeps a version number for each file, which increases every time the file is opened for writing. The version number may increase by an arbitrary positive increment, but since the *open* call returns both the latest version number (`reply.cacheVersion`) and the previous version number (`reply.oldCacheVers`), the client can check both these numbers against the version number of its cached copy (if it has one). If the client has cached data from either the current or the previous version of the file, this data can be retained; any other cached data is invalid. The `reply.attributes` field has the same value that would have been returned from a *getattr* (get file attributes) procedure; this avoids the need to make the *getattr* call that NFS must make the first time a file is used.

```
NFSPROC_CLOSE (file, writeMode)
               returns (reply)
        fhandle file;
        boolean writeMode;
        closeopres reply;
```

This procedure is called to tell the server that the client is no longer using the specified file handle. The `writeMode` argument should be the same as the one that was provided for the corresponding *open* operation; it must be supplied since *open* could have been called several times, with different modes, on a single file handle.

### 3.2. Server-to-client calls

Since a subsequent *open* operation may make a previously cachable file uncachable, an SNFS server may have to notify a client to stop caching a file that has already been opened. Alternatively, if the file has been written into a client's cache and subsequently closed (but not yet deleted), the client must write the cached data back to the server.

Whenever an SNFS server needs to notify a client, it issues a callback operation. In this case, the RPC call goes from the server to client, so the client must provide RPC service for this request:

```
NFSPROC_CALLBACK
        (file, cbmode, mdev, attr)
                returns ()
        fhandle file;
        unsigned mdev;
        cbMode cbmode;
        fattr attr;
        cbackopres reply;
```

The callback operation identifies the file in question through the `mdev` and `attr.fileid` values (these should be sufficient to allow the client to locate its internal data structures for the file). The file handle is provided to the client for it to use while performing any required write operations on this file, and to disambiguate stale cache entries. The `cbmode` argument specifies the kind of callback being done:

```
typedef enum {
        SNFS_CBWRITE = 1,
        SNFS_CBINVAL = 2,
        SNFS_WRINVAL = 3
} cbMode;
```

`SNFS_CBWRITE` indicates that any dirty blocks in the client's cache should be written back to the server. `SNFS_CBINVAL` indicates that any blocks in the client's cache should be invalidated (removed from the cache), and further caching should be disabled. `SNFS_WRINVAL` specifies both write-back and invalidate.

If the callback involves a write-back, the client should not return from the callback RPC until all the dirty blocks have been written back to the server. This has several implications:

1. The write operations generated by the client in response to the callback go to the same server that is waiting for the termination of the callback. Thus, an SNFS server must be multi-threaded to avoid deadlock (if there are *N* threads, only *N-1* may be doing callbacks simultaneously, so that at least one thread can service the write-backs).

2. Since the server makes a callback while servicing an *open* operation from another client, it cannot wait forever for the callback (since the client doing the *open* will time out). In general, the callback, together with any required write-backs, should finish much sooner than the RPC times out, but this is not guaranteed (the network might be slow, the server might be overloaded, or there might be many dirty blocks). We believe that this is not a serious problem; the callback operation can safely be reissued, so when the client doing the *open* operation times out and retries, no harm is done.

If the client ''serving'' the callback is down, the SNFS can honor the new *open* operation, but it should inform that client that the file may be in an inconsistent state. If the ''dead'' client comes back to life after this point, it must be prevented from making further use of the file until it obtains a new file handle and reopens the file.

# 4. Implementation

We implemented a prototype of Spritely NFS by modifying an existing NFS implementation, that used in the Ultrix 2.2 operating system. By changing the names of entry points and global variables, we made it possible to have both SNFS and unmodified NFS in the same kernel, which in turn made it easy to compare the performance of the two protocols. With the exception of a few utility programs, all the changes are confined to the kernel; for user code, there is no visible difference between NFS and SNFS.

## 4.1. Layering

In a Unix system with more than one kind of file system (for example, NFS and local disks), there must be a level of indirection to separate the filesystem-generic code from the individual filesystem-specific code. In Ultrix, this is done through the ‘‘generic file system’’ (GFS) [9]. GFS manages the file system block buffer cache, and expects the underlying file systems to present a consistent set of primitives for reading and writing file blocks. GFS implements an abstract data type called a *gnode*, which is similar to the traditional Unix in-memory *inode* data structure, but which supports filesystem-specific data and methods.

Our goal was to implement SNFS without modifying the GFS layer; we nearly succeeded (see section 4.2.1). Our changes to GFS make it ‘‘more generic’’; they are not specific to SNFS. We also found it profitable, for improved performance, to add a new function to the GFS buffer cache management code (see section 4.2.5).

On the server side, the NFS (and SNFS) service code simply translates RPC requests into GFS operations on the appropriate file system, normally the standard Unix local file system.

## 4.2. Client changes

### 4.2.1. Additional state information

The *gnode* data structure provides space for filesystem-specific data, some of which is already used by NFS. We extended this to include several new fields, including several flag bits (such as ‘‘caching enabled’’), the file version number, and authorization information to use when doing a delayed write[5]. No additional state tables are needed at the client.

One additional change is that, because a file may be reopened shortly after it is closed, SNFS *gnode* information persists after a file is closed. This allows cached data to persist until a subsequent use of the file, and it allows postponement of write-back, which may then be obviated if the file is removed. GFS may recycle a closed *gnode*, but it calls a cleanup function of the specific file system before doing so. The original GFS implementation assumed that this function never blocks. Since SNFS may have to make several RPC calls to clean up dirty blocks, we had to modify this part of GFS to remove the non-blocking assumption.

---

[5]In unmodified NFS, file system operations such as read or write are always done in the context of the requesting process, so the relevant ‘‘credentials’’ are always available. Since SNFS allows delayed writes, which are done in the context of a daemon process, it must put aside a copy of the requesting process’s credentials.

### 4.2.2. New calls

Although there is no *open* operation in the unmodified NFS protocol, GFS does call code in the NFS layer when a file is opened. NFS then does a *getattr* (get file attributes) operation; it is this ‘‘attributes’’ information that allows NFS to determine, in the future, if the cached file data is still valid.

An SNFS client, in contrast, does the explicit *open* operation at this point. This provides it with the attributes information, the current cachability state of the file, and the version number information. If the file had been opened before, and the version number has been changed by another client (indicating that another client has been writing the file), all previously cached blocks are invalidated.

In unmodified NFS, when a file is closed any pending writes are completed synchronously[6], and all cached blocks are invalidated.

In SNFS, it is not necessary to finish any pending writes; the file may be deleted soon, so these writes may be unnecessary. If the server wants the blocks for another client, it will send a callback to the last writer. That SNFS client then notifies the server, via the *close* operation, that the client is no longer using the file.

### 4.2.3. Cache strategy

Two kinds of information are cached on the client: file data blocks and file attributes. The file data blocks are cached in the GFS buffer pool; each block is marked with the appropriate file ID (and a hash table is used for fast mapping between files and their blocks). The file attributes are stored in the *gnode*. All file system types keep these caches; the local-disk file system, of course, has no consistency problem (ignoring crash recovery).

Unmodified NFS refreshes the attributes cache based on its age; in Ultrix, an adaptive mechanism is used which allows longer residence for files that have not been recently modified. In SNFS, the attributes cache need only be refreshed if the file is write-shared (not cachable). In order to guarantee consistent attributes for write-shared files, SNFS never uses cached attributes in this case. Also, the standard Unix read-ahead is disabled for write-shared files, since the read-ahead block cannot be cached.

NFS uses the normal GFS buffer cache for file data blocks. If, upon refreshing the attributes cache for a file, it discovers that the file modification timestamp has changed, it invalidates all the cached data blocks for the file. Thus, the period of potential inconsistency for cached data is the same as for cached attributes.

SNFS uses the explicit consistency protocol to determine if caching is allowed. As long as it is allowed, there is no need to check the consistency of cached data. While caching is not allowed, cache consistency is not an issue because data blocks are never entered in the GFS buffer pool (actually, existing GFS interface functions always enter the block in the cache, but the

[6]Writes may be pending because unmodified NFS does some asynchronous writing to improve performance, and because it buffers partial-block writes to the end of a file.

SNFS code immediately marks the buffer ‘‘invalid’’). When the cachability of a file changes as the result of a callback, any cached data blocks are invalidated at that time.

### 4.2.4. Callback service

In unmodified NFS, all RPC calls are initiated by the client. In SNFS, the server makes RPC callbacks to the client, so the client host must be able to service RPC requests. This is no problem, since even diskless workstations have the NFS server code in their kernels. Thus, we simply use the existing server mechanism, even on clients that do not actually export any file systems.

We added one RPC procedure to support callbacks. This is implemented as part of the SNFS server code, but conceptually it is part of the SNFS client. Callback handling is straightforward: the information in the callback is used to locate the *gnode* for the specified file, and the action specified in the callback is performed. Cache invalidation is done locally to the client; if the server requests write-back, the client uses the usual SNFS RPC calls to write the blocks back to the server.

Since the client may have closed the file before a write-back is requested, the callback service code may have to revive the *gnode* temporarily. Note that the *gnode* never actually disappears as long as there are dirty blocks associated with it, so it need not be recreated from scratch.

### 4.2.5. Delayed write policy

The local-disk file system used in Ultrix follows the traditional Unix policy of delaying file data writes, unless the user process calls an explicit flush operation. Blocks are written back to disk when the space is needed for other files. To bound the amount of damage caused by a crash, all the delayed-write blocks in the buffer pool are written to disk periodically using the *sync* system call (usually every 30 seconds, by `/etc/update`).

In the Sprite file system, dirty blocks are written back to the server when they reach 30 seconds in age; this is similar to, but somewhat less conservative than, the traditional Unix policy. NFS seldom uses the delayed-write mechanism, because of the effect on consistency. SNFS, on the other hand, uses the normal GFS delayed-write mechanism, so (mostly by default) it follows the traditional Unix policy of `sync`ing every 30 seconds.

Since it is relatively common for Unix applications to create a temporary file and then delete it after a few seconds, Sprite and SNFS attempt to take advantage of this behavior by ‘‘cancelling’’ delayed writes when a file is deleted. (NFS cannot do this, since it is synchronous. The traditional Unix local-disk file system also does not do this, perhaps because until recently, buffer pools have been so small that few writes would be avoided.)

We added a function to the GFS buffer management module that invalidates the delayed-write blocks associated with a specific file, without writing them back to the server. This involves a search of the buffer pool that is not efficiently supported by the traditional buffer pool data structures, especially as the cache size increases. Since large buffer caches otherwise improve performance, we reorganized the GFS buffer pool data structures to make delayed-write invalidation faster; this change also improved the performance of the *sync* system call.

## 4.3. Server state design

### 4.3.1. Server state table

An SNFS server, unlike an NFS server, must retain state about files between RPC calls. In our implementation, the SNFS server maintains a state table, organized as a hash table with collisions resolved by chaining, with one entry for each open file, and one entry for each file that is closed but for which the last writer may still have cached blocks. The hash key is the NFS file handle.

To avoid running out of kernel memory, we put a limit on the number of entries in this table. This limits the number of simultaneously open files for this server, a limit that is not imposed by an unmodified NFS server (but each entry requires only 68 bytes, so the limit can be liberal). When we run low on entries, those recording closed files with outstanding dirty blocks may be reclaimed by sending callbacks to the corresponding clients.

### 4.3.2. State table entries

Each entry in the state table contains the file handle for the corresponding file; this is used as the lookup key. It also contains the file's current version number, its current state (such as read-only or write-shared), and a list of ‘‘client’’ information blocks for each client host that has the file open.

A client information block contains the network address of the client host; this is used as an identifier and also to address the callback RPCs. A client block also contains counts of the number of readers and writers for this file at this client (more than one process there may have the file open) and the client's internal file system identifier, provided when the file was opened and used for the callback RPC.

If the file is closed but the last writer may have dirty blocks in its cache, the state table entry records a client block for that last writer.

### 4.3.3. Version number generation

The server assigns a version number to each file; the version number must increase each time the file is opened for writing. This allows a client to determine, when it opens a file, if the blocks it has in its cache are still valid. Ideally, this version number would be associated with each file on stable storage (as is done in Sprite), but since we did not want to modify the underlying Unix local file system to store additional information, we chose to use a global counter to generate version numbers.

This solution is suitable only for experimental use, as it poses several problems. First, the counter can wrap around; this could be ‘‘cured’’ by simply invalidating *all* client caches at this point. Second, the counter does not persist across server crashes; this could be cured by writing it to stable storage, either on every increment or perhaps only at multiples of a specific value. On crash recovery, the latest value of the stored counter would then be increased by that value to ensure that no duplicates result.

### 4.3.4. State transitions

Each file may be in one of several states. There is some freedom in the choice of state assignments; we chose one that is straightforward, although in retrospect it turned out to have some drawbacks (see section 6.2).

In our implementation of SNFS, the states are:

| | |
|---|---|
| CLOSED | The file is not open by any client. |
| CLOSED_DIRTY | The file is not open, but the last writer may still have dirty blocks[7]. |
| ONE_READER | The file is open read-only by one client. |
| ONE_RDRDIRTY | The file is open read-only by one client, which may have dirty blocks cached from a previous open. |
| MULT_READERS | The file is open read-only by two or more clients. |
| ONE_WRITER | The file is open read-write by one client. |
| WRITE_SHARED | The file is open by two or more clients, including at least one writer. |

Table 1 shows the possible state transitions. Note that no transition occurs (and thus none is shown) if a client that already has a file open for read-only issues another read-only *open* for that file, or if a client that has a file open for read-write issues another *open* of any sort for that file. Also, no transition occurs on a *close* operation except when it represents the last open reference to a file, or when it represents the next to last reference to a file open by multiple readers.

During any operation that causes a state transition, the server locks the corresponding state table entry so that all transitions occur serially. If a callback is necessary, the state table entry remains locked; therefore, while servicing a callback a client cannot perform any *open* or *close* operations on the specified file (or deadlock would result).

## 4.4. Server changes

The state table manager is implemented as a separate module. It takes care of initializing the server state data structures, and has entry points to perform the state transitions necessary on file *open* and *close* operations. Most of the code added to support SNFS is in this module.

The only changes to the original NFS server code were straightforward additions of new RPC service functions. The *open* operation is similar to the existing *getattr* operation, except that it calls the state table management code to record information about the new open, potentially resulting in a callback. The *close* operation is even simpler, since it does nothing but notify the state table manager.

[7]We actually represent CLOSED and CLOSED_DIRTY as subcases of one state; similarly for ONE_READER and ONE_RDRDIRTY.

| From State | To State | When | Caching | Callback |
|---|---|---|---|---|
| CLOSED | ONE_READER | Open for read | Enabled | None |
| CLOSED | ONE_WRITER | Open for write | Enabled | None |
| CLOSED_DIRTY | ONE_RDRDIRTY | Open for read by last writer | Enabled | None |
| CLOSED_DIRTY | ONE_READER | Open for read not by last writer | Enabled | Write-back |
| CLOSED_DIRTY | ONE_WRITER | Open for write by last writer | Enabled | None |
| CLOSED_DIRTY | ONE_WRITER | Open for write not by last writer | Enabled | Write-back and invalidate |
| ONE_READER | MULT_READERS | Open for read by different client | Enabled | None |
| ONE_RDRDIRTY | MULT_READERS | Open for read not by last writer | Enabled | Write-back |
| ONE_READER or ONE_RDRDIRTY | ONE_WRITER | Open for write by same client | Enabled | None |
| ONE_READER | WRITE_SHARED | Open for write by different client | Disabled | Invalidate |
| MULT_READERS | WRITE_SHARED | Open for write | Disabled | Invalidate |
| ONE_RDRDIRTY | WRITE_SHARED | Open for write by different client | Disabled | Write-back and invalidate |
| MULT_READERS | MULT_READERS | Open for read | Enabled | None |
| ONE_WRITER | WRITE_SHARED | Open for read or write by different client | Disabled | Write-back and invalidate |
| MULT_READERS | ONE_READER | Close by last but one reader | Not affected | None |
| ONE_READER or WRITE_SHARED | CLOSED | Final close | Not affected | None |
| ONE_RDRDIRTY | CLOSED_DIRTY | Final close | Not affected | None |
| ONE_WRITER | CLOSED_DIRTY | Final close | Not affected | None, this client recorded as last writer |
| ONE_WRITER | ONE_RDRDIRTY | Final close for write, client still reading | Not affected | None, this client recorded as last writer |

**Table 1:** SNFS server state transitions

### 4.5. Code size

A crude measure of the complexity of the modifications we made is the change in source code size. The unmodified NFS code we started with consisted of 9200 lines of commented C source code in 15 files. The SNFS version consists of 11150 lines in 16 files, with most of the increase coming from the SNFS server state manager. We believe that an implementation supporting both NFS and SNFS protocols would be only a few hundred lines longer than our SNFS code. (Note that our SNFS implementation does not yet include crash recovery code.)

Another measure of the code complexity is the size of the object code. The SNFS object code is about 20% larger than the NFS object code, totalling about 35 Kbytes of VAX™ instructions. Run-time data space requirements vary depending upon the limit imposed on the number of open files; for example, up to 1000 simultaneously open files can be accommodated with about 70 Kbytes of data space.

## 5. Performance

In this section, we look at the performance differences between NFS and SNFS. We are concerned with the case where there is no concurrent sharing of a file between two or more client hosts, because this is by far the most common case. In the write-shared case, SNFS disables the client cache and so performs much worse than NFS — but much more correctly.

### 5.1. Factors affecting performance

The performance differences between NFS and SNFS are the result of variation in several factors:

- The parallelism available with delayed write instead of write-through.
- The writes averted when temporary files are deleted before being written back.
- The number of RPC calls required over the active lifetime of a file.
- The computational demands of protocol support.

We believe that the computational costs of the SNFS implementation are not significantly different from those of NFS. For the other factors, however, NFS and SNFS can differ considerably, depending on the application mix.

For example, SNFS gains most from increased parallelism when only one job is running on the client host, and it can alternate computation with write I/O (such as a compiler). File copying can also benefit as long as the cache does not fill with dirty blocks, because the writes are often postponed so as to overlap with a less I/O-intensive task. Less such I/O parallelism is available if many applications are running in parallel on the client.

Similarly, SNFS gains by avoiding writes only if the application is generating a significant volume of temporary files (and if these files fit easily into the client cache).

Finally, the relative number of RPC operations depends on the pattern of access to a file. For example, a file that is read only once for a brief period (such as a source module) differs from a file that is read over the course of several seconds (some text editors do this, for example). In the

‘‘read-quickly’’ case, NFS will require one fewer RPC than SNFS, since SNFS requires the additional *close* operation (the SNFS *open* operation is equivalent to the *getattr* operation done at file-open time by NFS).

In the ‘‘read-slowly’’ case, the relative RPC counts will be closer, since NFS must do consistency probes every few seconds. Normally, the NFS model wins because most applications follow the ‘‘read-quickly’’ pattern. As we point out in section 6.2, however, a minor modification to our implementation of SNFS would probably provide significant performance gains over NFS in the case where a file, such as a popular header file, is read repeatedly during the course of some seconds. This pattern is actually quite common.

In addition to effects of the application mix, the relative performance of SNFS and NFS depends on system parameters including the file cache size, RPC speed (composed of processor and network costs), and disk access time. As the client’s file cache size increases, the relative benefit of clever cache-management protocols increases as well. Also, when the gap between processor speeds and disk access time widens (as it appears to be doing), cache-management efficiency becomes more important. Finally, since NFS and SNFS differ somewhat in the number of RPC calls used, increases in RPC speed (relative to processor speed) reduce the relative performance difference.

## 5.2. Andrew benchmark measurements and analysis

Our SNFS implementation was originally developed for Ultrix running on a MicroVAX-II™ with a relatively small memory. Because we were interested in the effects of large caches, we ported the code to the experimental Titan workstation; Titans are RISC processors running about 12-15 times as fast as a VAX-11/780, and supporting up to 128 Mbytes of main memory [6]. Identical machines were used for client and server, and the RA81 and RA82 disks used are moderately high performance drives. The operating system running on the Titan is not exactly Ultrix, but the NFS and other file system code is taken directly from Ultrix, with only a few lines changed because of architectural differences. All our measurements were made on Titans.

It is relatively easy to benchmark the individual cases where one might expect SNFS performance to differ from NFS performance. It is harder to measure an aggregate difference, since the weighting for the individual differences depends so much on the application mix. We chose to concentrate on the Andrew benchmark suite [2], since it covers many of the individual cases and does give some idea of the aggregate performance. The Andrew benchmark spends a significant amount of time doing compilation; since the cost of compilation depends upon the target architecture, it is not possible to compare our figures directly to previously published results from the Andrew benchmark[8]. We also benchmarked an external sort application, since this emphasizes the differential performance on temporary files; see section 5.3.

[8]We used a slightly modified version of the original Andrew benchmark, due to John Ousterhout [7], that does produce comparable numbers. This is done by using a portable compiler and loader that produce code for a fixed target architecture, not for the architecture being tested. We hope that future benchmarking will be based on this portable version.

The Andrew benchmark consists of 5 phases, applied to a tree of directories and files; the following description is taken from [2]:

| | |
|---|---|
| *MakeDir* | Constructs a target subtree that is identical in structure to the source subtree. |
| *Copy* | Copies every file from the source subtree to the target subtree. |
| *ScanDir* | Recursively traverses the target subtree and examines the status of every file in it; does not actually read the contents of any file. |
| *ReadAll* | Scans every byte of every file in the target subtree once. |
| *Make* | Compiles and links all the files in the target subtree. |

Different phases highlight different differences between SNFS and NFS. The *Copy* phase favors SNFS, since the delayed-write policy allows more parallelism between the read and write I/O streams. The *ScanDir* and *ReadAll* phases favor NFS, since SNFS has about one additional RPC to do for each file. The *Make* phase favors SNFS because it allows parallelism between file writing and either file reading or computation.

Because the delayed-write policy of SNFS postpones some operations until after the completion of the benchmark, we ran the SNFS benchmarks several times in a row (rather than interleaving them with NFS benchmark runs) so that NFS would not be charged for writes incurred by SNFS.

We ran the benchmark in three configurations: one with all files on the local disk, one with just the data files remotely mounted but temporary files kept locally, and the last with both data and temporary files remotely mounted. The latter configuration should favor SNFS for the *Make* phase, since it allows the ‘‘delete-before-writeback’’ optimization to take effect. In all configurations, the ‘‘compiler’’ programs were on the same file system as the data, and other Unix utility programs were on the local disk.

The results are shown in table 2. Each number shown is an average over 10 trials. The times are measured with an accuracy of no better than a second or two, so slight variations should not be taken seriously.

| Elapsed time in seconds | | | | | |
|---|---|---|---|---|---|
| Phase | Local | NFS, /tmp local | SNFS, /tmp local | NFS, /tmp remote | SNFS, /tmp remote |
| *MakeDir* | 4.7 | 4.6 | 4.6 | 4.5 | 4.3 |
| *Copy* | 24 | 48 | 35 | 48 | 37 |
| *ScanDir* | 43 | 52 | 55 | 52 | 55 |
| *ReadAll* | 37 | 50 | 53 | 51 | 53 |
| *Make* | 215 | 303 | 237 | 377 | 266 |
| **Total** | 322 | 457 | 384 | 532 | 415 |

**Table 2:** Results of Andrew benchmark

During our experiments, neither the client nor server machine were used for any other jobs (although some housekeeping tasks occasionally run in the background). Both machines had large file buffer caches (about 16M bytes on the client and 3.5M bytes on the server), large enough that no data was ever removed from the caches due to replacement. This simplifies analysis but does favor SNFS, which is better able to make use of a large cache than NFS.

The results shown in table 2 confirm our expectations. SNFS performs about 25% better on the *Copy* phase, and 20% to 30% better on the *Make* phase (depending on whether `/tmp` is local or remote). NFS performs about 5% better on the *ScanDir* and *ReadAll* phases. SNFS completes the entire benchmark 15% to 20% faster than NFS, because the complete benchmark places most weight on the *Make* phase.

| Remote Procedure Calls | | | | |
|---|---|---|---|---|
| Call | NFS, /tmp local | SNFS, /tmp local | NFS, /tmp remote | SNFS, /tmp remote |
| *open* | | 700 | | 778 |
| *close* | | 700 | | 776 |
| *getattr* | 757 | 225 | 933 | 311 |
| *setattr* | 22 | 22 | 22 | 22 |
| *read* | 1130 | 699 | 1961 | 1033 |
| *write* | 868 | 590 | 1425 | 921 |
| *lookup* | 3345 | 3345 | 3543 | 3543 |
| *other* | 273 | 273 | 376 | 376 |
| **Total** | 6395 | 6554 | 8260 | 7760 |

**Table 3:** RPC calls for Andrew benchmark

For each of the NFS and SNFS configurations, we collected RPC operation counts, as shown in table 3. (When `/tmp` is remotely mounted, the counts may be insignificantly off, since it is hard to avoid using this directory for housekeeping activities.) With `/tmp` on a local disk, SNFS requires slightly (2%) more RPC operations, but since SNFS substitutes *open* and *close* operations for the more expensive *read* and *write* operations, it probably comes out ahead in total cost. With `/tmp` remotely mounted, SNFS requires 6% fewer total operations, and 42% fewer data transfer operations.

Several entries in table 3 deserve explanation. When `/tmp` is mounted locally, one might expect both protocols to issue the same number of *write* RPC calls. Because the Ultrix NFS implementation delays partial-block writes, it is more sensitive than SNFS to the ‘‘natural’’ file system block size used at the server. During our tests, we used a 4K byte block; NFS might have performed slightly better had we used an 8K byte block size.

NFS also issues far more *read* RPC calls. This appears to be because there are many instances where the client first writes a file, and then reads it. Although the original blocks persist in the cache, when the NFS client opens the file for reading, it sees a more recent timestamp than it saw

previously (because the timestamp is generated at the server), and so it must assume that the file has been modified. The SNFS client, in contrast, knows that it was the last writer of the file, and uses its cached blocks.

Finally, we note that roughly half of the RPC calls are file name lookups (SNFS and NFS use the same protocol for this). Clearly, any mechanism that reduced the number of lookups would improve performance; we suspect that applying the Sprite consistency protocols to a cache of directory entries might be a good approach[9].

We were also interested in the effect of file system protocol on ‘‘server utilization,’’ the CPU load placed on the server for a given application. Measurements of the Sprite operating system suggest that the Sprite file system can support about four times as many clients as can a Unix system with NFS running on identical hardware [5]. We measured the server CPU load (roughly, the percentage of time not spent in the ‘‘idle’’ state) while running the Andrew benchmark for NFS and SNFS; in both cases, `/tmp` was remotely mounted, so we effectively simulated the load of a diskless workstation. We also measured the rate of RPC calls, as well as individual rates for *read* and *write* calls. Graphs of the server load and call rates are shown in figure 1 for the NFS benchmark, and in figure 2 for the SNFS benchmark. All the graphs in one figure are for the same run, so one can see how the rates are correlated in time.

The load varied considerably over the course of the benchmark, and was strongly correlated with the aggregate rate of RPC calls; it was *not* correlated with the rate of *read* or *write* calls. Since SNFS, even when `/tmp` is remotely mounted, requires only slightly fewer operations than NFS, the integral of CPU load over time was only slightly lower for SNFS. In fact, since the SNFS benchmark completes significantly faster, the average server load during the benchmark is slightly higher than for NFS; it also appears to be slightly burstier.

We believe that the advantage, in server CPU utilization, of Sprite over NFS is probably the result of a more efficient RPC protocol and perhaps a more efficient file name translation mechanism. We have no evidence to show that the SNFS cache consistency protocol itself, in isolation from the write policy, leads to significantly different server CPU utilization on the Andrew benchmark. On the other hand, table 3 shows that the server disk utilization with SNFS is 30% to 35% lower.

## 5.3. Sort benchmark measurements and analysis

As noted earlier, one case where the cache consistency protocol *does* have a significant effect is when a single client first writes, then reads a temporary file. We explored this case by benchmarking the Unix *sort* program, which does an external sort and so makes heavy use of temporary files.

We measured the performance of the sort program with its temporary files (kept on `/usr/tmp`) on local disk, remote-mounted via NFS, and via SNFS. Table 4 shows the resulting elapsed times for input files of three different sizes; the important parameter is the amount of temporary storage used, which grows faster than the input file.

---

[9]The reference port of NFS includes a relatively ineffective cache that holds only directory names; we hear rumors that some work has been done on leaf-name caching, and we are trying to track them down.

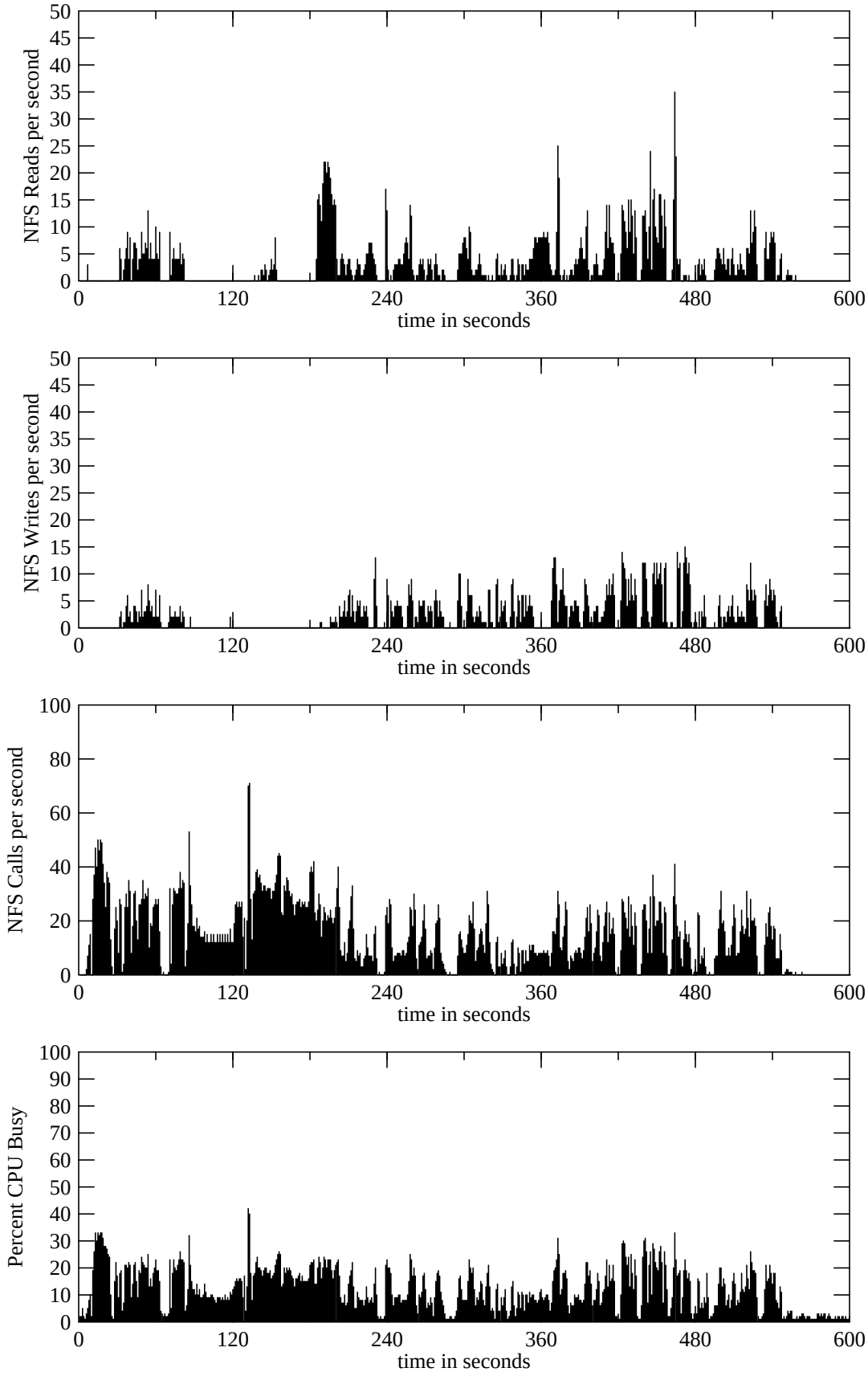


**Figure 1:** Server utilization and call rates for NFS

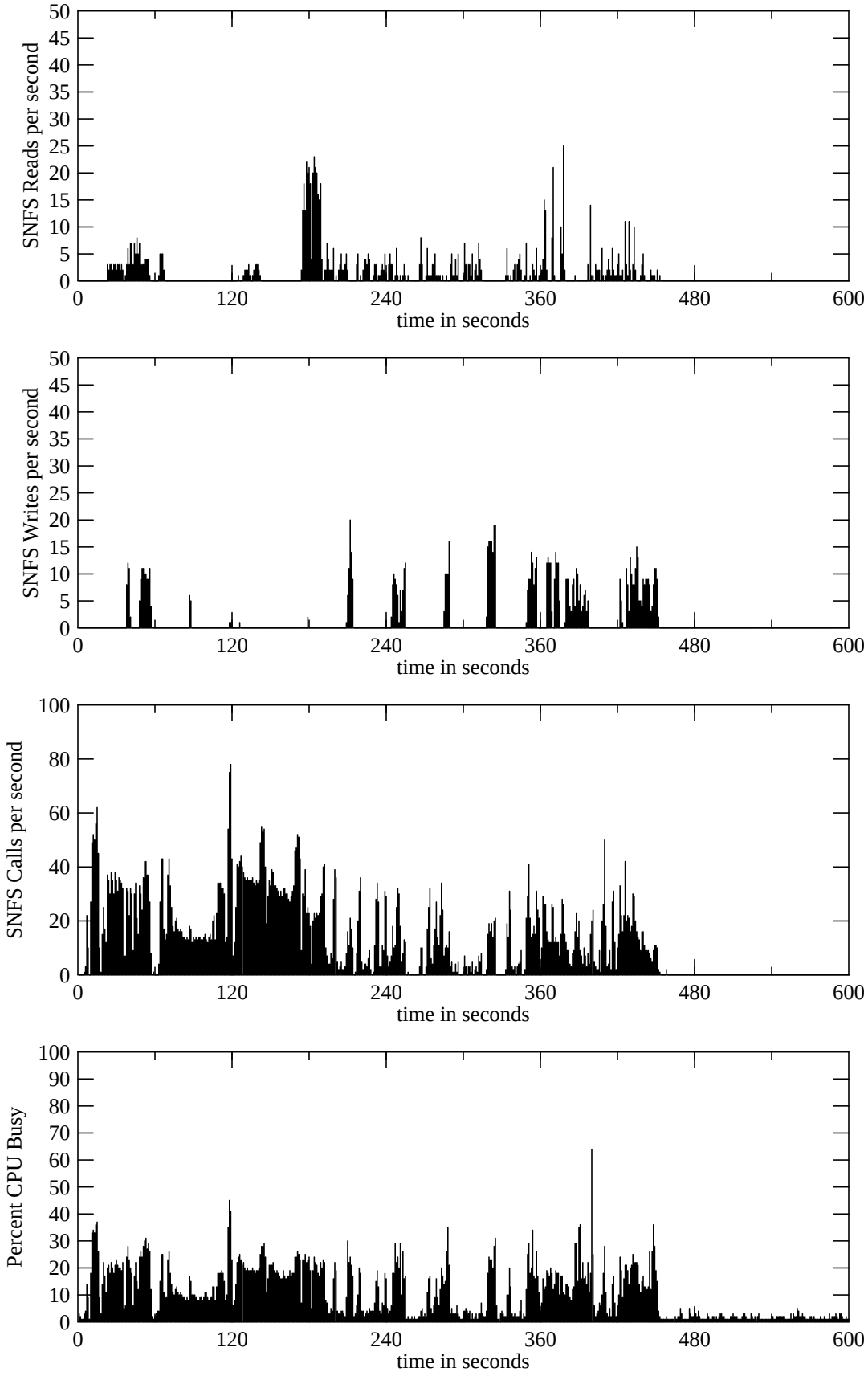


**Figure 2:** Server utilization and call rates for SNFS

| File size | Temp storage | local /usr/tmp | NFS /usr/tmp | SNFS /usr/tmp |
|---|---|---|---|---|
| 281 K | 304 K | 4 sec | 8 sec | 4 sec |
| 1408 K | 2170 K | 33 sec | 105 sec | 48 sec |
| 2816 K | 7764 K | 74 sec | 234 sec | 127 sec |

**Table 4:** Results of Sort benchmark

| Remote Procedure Calls | | | | |
|---|---|---|---|---|
| Call | 1408 Kbytes, NFS | 1408 Kbytes, SNFS | 2816 Kbytes, NFS | 2816 Kbytes, SNFS |
| *open* | | 35 | | 68 |
| *close* | | 35 | | 67 |
| *getattr* | 93 | 37 | 150 | 70 |
| *read* | 681 | 33 | 1340 | 67 |
| *write* | 729 | 706 | 1452 | 1441 |
| *other* | 108 | 107 | 203 | 207 |
| **Total** | 1611 | 953 | 3145 | 1920 |

**Table 5:** RPC calls for Sort benchmark

SNFS dramatically outperforms NFS on this benchmark, completing approximately twice as fast. In all three cases the client CPU utilization is higher for SNFS; in other words, I/O latency is the bottleneck. Table 5 shows that SNFS does far fewer *read* RPC calls than does NFS, indicating that the NFS cache-consistency mechanism is clearly at fault. (It is failing to read from the cached copies of recently-written temporary files.) We believe that this accounts for a third to a half of the performance difference, the rest attributable to the synchronous writeback-on-close required in NFS.

Unlike the Andrew benchmark, on this benchmark the total CPU utilization is about 40% lower for SNFS, probably because SNFS does about 40% fewer RPC calls. This is a significant improvement; however, the Andrew benchmark may be more representative of real applications.

## 5.4. Avoiding file writes for temporary files

A delayed write policy means that data written to short-lived temporary files may never need to be sent to the server. Unix systems normally run a process called `/etc/update`, which writes back dirty blocks every 30 seconds. This is desirable for limiting the damage caused by a crash, but the sort benchmark lasts long enough that one of these write-backs is likely to occur even if few of the temporaries reach the age of 30 seconds.

| File size | Temp storage | local /usr/tmp | NFS /usr/tmp | SNFS /usr/tmp |
|---|---|---|---|---|
| 1408 K | 2170 K | 32 sec | 97 sec | 29 sec |
| 2816 K | 7764 K | 69 sec | 246 sec | 69 sec |

**Table 6:** Sort benchmark, infinite write-delay

| Remote Procedure Calls | | | | |
|---|---|---|---|---|
| Version | `update`? | Reads | Writes | Others |
| NFS | Yes | 1340 | 1452 | 353 |
| NFS | No | 1227 | 1451 | 368 |
| SNFS | Yes | 67 | 1441 | 412 |
| SNFS | No | 65 | 33 | 407 |

**Table 7:** RPC calls for Sort benchmark, 2816 Kbyte input file, with infinite write-delay

To emphasize the benefits of delaying writes of temporary files, we ran the sort benchmark with the `/etc/update` process disabled. The results, shown in table 6, show that for files whose lifetime is short enough, SNFS matches or beats local-disk performance (even though dirty blocks are not written, the local-disk file system still writes out structural information). NFS performance is unchanged, within the limits of measurement error. Table 7 shows that SNFS, in this situation, is doing almost no *write* RPC operations.

## 6. Future work

In this section we touch on several issues, in addition to crash recovery, that we have not yet addressed in our implementation.

### 6.1. Coexistence of NFS and SNFS

SNFS can coexist quite easily with unmodified NFS. We have demonstrated that a single client can remote-mount file systems using either protocol, and that a single server can provide access to separate file systems using either protocol. It is slightly trickier to support simultaneous access via both NFS and SNFS to the same file system, since the NFS clients cannot participate in the SNFS consistency protocol.

One approach is to treat any NFS access to a file already open under SNFS as implying an SNFS open operation. If the file is open for reading under SNFS, as long as the NFS client issues only read operations, this can be treated as if it were the MULT_READERS state. If the file is open for writing under SNFS, or if the NFS client attempts a write operation, the server can issue callbacks to the SNFS clients and put the file into the WRITE_SHARED state. In this state,

the SNFS clients effectively follow the NFS consistency model, but with an even stricter inter-probe interval.

If a file is used first by an NFS client, it might appear necessary to perform a callback if an SNFS client then causes the file to enter the write-shared state. This is not true, since the NFS client will not cache dirty blocks. The hard problem is to know when the NFS client has stopped using a file. We can do this by keeping, for a period no less than the longest NFS attributes-probe interval, a record of each file accessed via NFS. Files that have not been accessed within this interval may safely be declared closed, since an NFS client will not use its cached copy of such a file without first checking with the server.

A server can easily tell if a client is using NFS or SNFS because the SNFS clients will always perform an open operation before doing anything else to a file (we ignore the possibility of an intervening crash.) Conversely, a client can tell if a server supports SNFS because an NFS server will reject an open operation. Thus, SNFS clients and SNFS servers will discover each other, and other combinations will simply revert to the standard NFS protocol.

### 6.2. Delaying the SNFS close operation

Our current SNFS implementation sends an open operation to the server every time a process opens a file. This is not necessary; since most files are reopened soon after they are closed, we can avoid a lot of network traffic if the SNFS clients delayed close operations in anticipation of a subsequent open operation. The client would keep a flag in the *gnode* structure indicating that a ‘‘closed’’ file has not yet been reported to the server; this would allow it to realize that a subsequent open operation can be performed locally.

Delayed-close may create situations where the server perceives write-sharing to be taking place, when in fact it is not. If a client with a delayed-close file receives a callback for that file, the appropriate response is to close the file so that it can be cached by the new client host. The first client cannot, however, simply issue a close operation during the callback, since this would deadlock the server (in our current implementation, at least). Instead, the client could return a special status code for the callback operation that would tell the server to treat the file as closed.

Delayed-close will also cause the server’s state table to fill up with apparently open files. It appears to be necessary, therefore, to create a new callback mode that asks a client to relinquish a closed file; the server would perform these as necessary to attempt to reclaim state table entries that have not been used recently.

## 7. Summary and Conclusions

Our experiments have convinced us that the Sprite approach to consistency is superior to that used in NFS. NFS cannot provide complete consistency without unacceptable performance. Even with the weak consistency provided by most NFS implementations, performance is probably worse than that provided by the Sprite consistency protocol.

We found that adding the Sprite consistency protocol to NFS was possible without major disruption of the NFS implementation, and required only a few programmer-months. In order to

show that statelessness is not perfection, we would have to demonstrate that SNFS has the same fault-tolerance as NFS; this would require implementation of the Sprite recovery protocol.

We did not find that SNFS outperformed NFS as much as Sprite was supposed to have outperformed NFS [5]. One reason may be that the NFS we used has been adjusted to place performance ahead of consistency; perhaps this is the right choice. A more intriguing question is whether the high rate of file lookup calls, as we detailed in table 3, swamps other file system performance differences. NFS and SNFS use the same lookup mechanism; Sprite uses an entirely different approach, which might account for its advantage, and might profitably be applied to the NFS protocols.

Caching in file systems is becoming more crucial as processor speeds and memory sizes improve faster than disk access times. We cannot afford to use inadequate cache mechanisms simply because the good ones seem harder to implement.

## 8. Acknowledgements

Richard Swan prompted our interest in integrating Sprite concepts into a more quotidian operating system. John Ousterhout, in his post-sabbatical semi-residence at our laboratory, helped us understand Sprite and kept us honest. Richard Hyde and Chris Kent helped us brainstorm our design, and Bob Rodriguez kindly took time off from his summer vacation to help us understand the arcana of Ultrix. Chet Juszczak provided both encouragement and a reality check. Bill Hamburgen, Mary Jo Doherty, and Joel McCormack helped to proofread the final drafts. Our mistakes are our own, of course.

## 9. References

[1] M. J. Bach, M. W. Luppi, A. S. Melamed, and K. Yueh. A Remote-File Cache for RFS. In *Proc. Summer 1987 USENIX Conference*, pages 275-280. Phoenix, AZ, June, 1987.

[2] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems* 6(1):51-81, February, 1988.

[3] Chet Juszczak. Improving the Performance and Correctness of an NFS Server. In *Proc. Winter 1989 USENIX Conference*, pages 53-63. San Diego, February, 1989.

[4] Christopher A. Kent. *Cache Coherence in Distributed Systems*. PhD thesis, Purdue University, 1986. Also available as Digital Equipment Corporation Western Research Laboratory Research Report 87/4.

[5] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1):134-154, February, 1988.

[6] Michael J. K. Nielsen. *Titan System Manual*. Research Report 86/1, Digital Equipment Corporation Western Research Laboratory, September, 1986.

[7] John Ousterhout. Private communication. 1989.

[8] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. 10th Symposium on Operating Systems Principles*, pages 15-24. Orcas Island, WA, December, 1985.

[9] R. Rodriguez, M. Koehler, and R. Hyde. The Generic File System. In *Proc. Summer 1986 USENIX Conference*, pages 260-269. USENIX, Atlanta, GA, June, 1986.

[10] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network filesystem. In *Proc. Summer 1985 USENIX Conference*, pages 119-130. Portland, OR, June, 1985.

[11] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A Caching File System For A Programmer's Workstation. In *Proc. 10th Symposium on Operating Systems Principles*, pages 25-34. Orcas Island, WA, December, 1985.

[12] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*. RFC 1094, Network Information Center, SRI International, March, 1989.

[13] *Webster's New Collegiate Dictionary*. G. & C. Merriam Company, Springfield, MA, 1979.

[14] Brent B. Welch. *The Sprite Distributed File System*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California -- Berkeley, 1989. In preparation.

# Table of Contents

# List of Figures

# List of Tables