APRIL 1992

WRL
Technical Note TN-27

A Recovery Protocol for Spritely NFS

Jeffrey C. Mogul

d|i|g|i|t|a|l **Western Research Laboratory** 250 University 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 technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

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

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University 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`
Internet:	`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.

A Recovery Protocol for Spritely NFS

Jeffrey C. Mogul

April, 1992

Abstract

NFS suffers from its lack of an explicit cache-consistency protocol. The Spritely NFS experiment, which grafted Sprite's cache-consistency protocol onto NFS, showed that this could improve NFS performance and consistency, but failed to address the issue of server crash recovery. Several crash recovery mechanisms have been implemented for use with network file systems, but most of these are too complex to fit easily into the NFS design. I propose a simple recovery protocol that requires almost no client-side support, and guarantees consistent behavior even if the network is partitioned. This proves that one need not endure a stateless protocol for the sake of a simple implementation.

I also tidy up some loose ends that were not addressed in the original experiment, but which must be dealt with in a real system.

This is a preprint of a paper presented at the *USENIX Workshop on File Systems*, May, 1992.

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

Table of Contents

1. Introduction

NFS has been extremely successful, in large part because it is so simple and easily implemented. The NFS ‘‘stateless server’’ dogma makes implementation easy because the server need not maintain any (non-file) state between RPCs, and so need not recover state after a crash.

Statelessness is not inherently good. Since many NFS operations are non-idempotent and might be retried due to a communication failure, to get reasonable performance and ‘‘better correctness’’ the server must cache the results of recent transactions [8]. Such cache state is not normally recovered after a crash, although this exposes the client to a possible idempotency failure.

A more serious problem with NFS statelessness is that it forces a tradeoff between inter-client cache consistency and client file-write performance. In order to avoid inconsistencies visible to client applications, NFS client implementations (by tradition, rather than specification) force any delayed writes to the server when a file is closed. This ensures that when clients use the following sequence:

Writer	**Reader**
open()	
write()	
close()	
	open()
	read()
	close()

the reader will see the most recent data, *if* the writer and reader explicitly synchronize so that the reader’s *open* takes place after the writer’s *close*.

Unfortunately, this means that the *close* operation is synchronous with the server’s disk. Since most files are small [2, 11], this means that most file writes effectively become synchronous with the server’s disk, and NFS clients spend much of their time waiting for disk writes to complete. Also, although many files have very short lifetimes and are never shared, and need never leave the client’s cache, NFS forces them to the server’s disk and so wastes a lot of effort. Finally, NFS does not guarantee any form of cache consistency for simultaneous write-sharing, with the result that occasional consistency errors plague NFS users.

The Sprite file system [10] solves these problems by introducing an explicit cache-consistency protocol. The fundamental observation is that write-sharing is rare, and can be detected by the server if clients report file opens and closes (not done in NFS), so the write-through-on-close policy can be eliminated. Instead, when write-sharing does occur, Sprite turns off all client caching for the affected file, and thus provides true consistency between client hosts.

2. Goals and design philosophy

Before describing the changes I propose for Spritely NFS, I want to briefly list the goals that I think such a system should meet:

- **Simplicity**: Spritely NFS was a successful experiment partly because it required minimal changes to an NFS implementation, and almost no changes to any other code. Any improved version should avoid unnecessary complexity, especially on the client side.
- **Consistency**: Spritely NFS should provide guaranteed cache consistency at all times. A partial guarantee is no improvement on NFS, since an application cannot make use of a partially-guaranteed property.
- **Performance**: Spritely NFS is not worth doing unless its performance, even with recovery, is better than that of NFS. While Spritely NFS also promises better consistency, that in itself would not convince many users to switch.
- **Reliability**: Spritely NFS should be no less reliable than NFS or the local Unix file system. (Note that I am satisfied with matching the lesser of these reliabilities in a given situation; NFS is sometimes, but not always, more reliable than a local Unix file system, and Spritely NFS sometimes must give up these NFS properties.)
- **No-brainer operation**: System managers should not need to do anything special to manage a Spritely NFS system. In particular, they should not need to adjust parameter values.
- **Incremental adoption**: Spritely NFS clients should interoperate with NFS servers, and vice versa. Otherwise, users will not have much of an incentive to adopt Spritely NFS, since this would mean replacing large parts of their infrastructure all at once.

3. Review of Spritely NFS

Spritely NFS [15] was an experiment to show that a Sprite-like consistency protocol could be grafted onto NFS, and to show that the performance advantage of Sprite over NFS was in large part due to the consistency mechanism rather than other differences between Sprite and Unix®. In this section I will summarize the design of Spritely NFS.

Two new client-to-server RPC calls are added to the basic NFS suite, *open* and *close*. The *open* call includes a mode argument that tells the server whether the client is writing the file or just reading it.

The NFS server is augmented with a ‘‘state table,’’ recording the consistency state of each currently-open file. In Spritely NFS, this state table is relevant only to the *open* and *close* RPCs; all other client RPCs are handled exactly as in NFS. When a client issues an *open* RPC, the server makes an entry in its state table and then decides, based on other state table information, if the specified open-mode conflicts with uses by other clients. If the *open* is conflict free, the server (via the RPC return value) notifies the client that it can cache the file. Otherwise, the client is not allowed to cache the file.

In some cases, a conflict may arise after a client has opened a file and has been allowed to cache it. For example, the first client host might open a file for write, and be allowed to cache it, and then a second host might open the same file. At this point, in order to maintain consistency, the first client must stop caching the file.

For this reason, Spritely NFS adds server-to-client ‘‘callback’’ RPCs to the NFS protocol. When a server decides that a client must stop caching a file, it does a callback to inform the client. A client with cached dirty blocks may have to write these blocks back to the server before replying to the callback RPC.

Spritely NFS clients need not write-through dirty blocks when a file is closed. The server keeps track of closed-dirty files and can ask the client to write the blocks back if another client opens the file for reading, but otherwise the writer client can write the blocks back at its own leisure. A client with closed-dirty blocks might even remove the file before the blocks are written back, thus avoiding wasted effort.

4. Loose ends in Spritely NFS

Besides lacking support for recovery, Spritely NFS failed to address a few other issues that need to be solved in a real system. I will propose solutions for these before describing the recovery protocol, since some of these issues complicate the basic recovery mechanism.

4.1. Dealing with ENOSPC

One problem with the delayed-write-after-close policy is that one or more of these writes might fail. In NFS, since the client implementation forces all writes to the server before responding to the *close* system call, an application which checks the return value from both *write* and *close* calls will always know of any write failures. Not so in Spritely NFS, since the failure might occur long after the application has released its descriptor for the file (or even after the application has exited). This could cause trouble for applications that do not explicitly flush their data to disk.

There are three categories of error that can occur on a client-to-server *write* operation:

1. **Communications failure**: the network is partitioned or the server crashes, and the RPC times out before the failure is repaired.
2. **Server disk hardware error**: the disk write operation fails, or the disk fails after the write completes.
3. **Server out of disk space**: no space is available on the server disk.

The first error can be turned into a delay by simply retrying the RPC until the server responds[1]. If the client crashes in the interim, then the dirty block is lost ... but this is no different from a normal local-filesystem delayed write in Unix.

The second error is not generally solvable, even by a strict write-through policy. It is true that the NFS approach will report detectable write failures, but these are increasingly rare (because techniques such as bad-block replacement can mask them). Again, normal Unix local-filesystem semantics does not prevent this kind of error from occurring long after a file has been closed.

---

[1]This is not true if the client uses a ‘‘soft mount,’’ which turns RPC timeouts into errors rather than retries. Soft mounts are generally thought of as living dangerously, although delaying writes after a *close* does make them even more dangerous. Perhaps soft-writes-after-close should be made ‘‘harder’’ as long as the client has enough buffer cache to avoid interference with other operations.

The third error (ENOSPC, in Unix terms) is the tricky one. We want to report these to the application, because it might want to recover from the condition, and because there is no obvious way for the underlying file system mechanism to recover from ENOSPC. (Also, unlike the other two kinds of errors, one cannot avoid ENOSPC errors through fault-tolerance techniques.)

My understanding is that Sprite does not completely solve this problem; that is, Sprite applications can believe their writes are safe but the delayed writes pile up in a volatile cache because the server is out of space. I would be curious to know if and how AFS deals with this (although in AFS, the client cache is ‘‘stable’’ and so the consequences are less urgent).

I propose solving the ENOSPC problem by changing the Spritely NFS *close* RPC to reserve disk space for the remaining dirty blocks. That is, when a dirty file is closed, the client counts up the number of dirty bytes and requests that the server reserve that much disk space for the file. The server may respond with an ENOSPC error at this point, in which case the client can revert to a write-through on close policy (note that we will allow the server to respond to *close* with ENOSPC even when enough space does exist, so the client should attempt the writes and report an error to the application only if a write actually fails.).

If a client, after it has obtained a reservation, decides it does not need to write back some of the dirty blocks (because it instead removes or truncates the file), it can issue a *release* RPC to release part or all of a reservation. Note that a *write* RPC might not actually change the size of a file, so once the last dirty block is gone from the client, the client must set the reservation to zero with a *release* RPC[2] (*release* might be implemented as *close* with a reservation request equal to zero.)

When a server receives a reservation request, attached to a *close* RPC, it should arrange with the underlying disk file system to reserve some of the remaining free space for the file in question. (If the space is not available, the server responds to the *close* with ENOSPC.) The file system need not actually allocate space on disk for the reservation; rather, it only needs to keep a count of the number of free bytes and the number of reserved bytes, and ensure that the difference never becomes negative.

The server keeps track of a file’s reservation in its state table entry. When a server handles a *write* RPC for a closed file, it decrements the reservation, and does a special kind of write to the underlying file system that tells the file system to decrement the count of reserved bytes[3]. If a client in the closed-dirty state tries to write more blocks than its reservation allows, the writes will fail (this is to prevent cheating and inconsistencies in the reservation count). Note that a client which has not asked for a reservation can write blocks as long as non-reserved space exists in the file system. Note also that the underlying file system must prevent local applications from allocating disk space if the non-reserved space drops to zero.

---

[2]The original Spritely NFS design does not have an explicit operation to cause a transition from the closed-dirty state to the closed state. This could cause the server’s state table to fill up. In the original system, the plan was that the server would do a callback to force the transition, if necessary. The *release* RPC solves this problem.

[3]The count is decremented by the amount of space actually allocated in the file system, not the size of the write. This avoids a double decrement of the reservation when a *write* RPC is retried.

If a client crashes while holding a reservation, or simply never makes use of it, the space could be tied up indefinitely. Thus, the server should set a time limit on any reservation grant (perhaps in proportion to the number of blocks reserved; if a client reserves space for a billion bytes, it is unlikely that they could all be written back within a short interval. A server might also refuse to honor a reservation for more than a few second's worth of disk writes). When the time limit expires and space is low, the server can reclaim the reservation by doing a callback (to force the client to write back the dirty blocks).

A client that fails to respond to the callback, perhaps because of a network partition, might end up being unable to write dirty blocks if the server reclaims its reservation. Since a partition might last arbitrarily long, there is not much that can be done about this: conceptually, this is the same as a disk failure. However, to avoid provoking this problem, a server might refrain from reclaiming timed-out reservations as long as sufficient free space remains.

One subtle problem can occur with this scheme if two processes on one client are writing the same file. After one successfully closes the file (i.e., the server grants a reservation), if the other client extends the file so much that the server runs out of disk space, some part of the file might not be written to the server. This is not an entirely contrived example; the file might be a ‘‘log,’’ appended to by multiple processes. The client implementation can preserve correct semantics in such a case by ordering the disk writes so that none of the blocks dirtied after the *close* are written to the server before the other dirty blocks of that file.

4.2. Directory caching

Spritely NFS did not address the issue of directory caching. This is important because a large fraction of NFS traffic consists of directory lookups and listings. Many NFS implementations cache directory entries, but because NFS has no consistency protocol these caches must time out quickly and can nevertheless become inconsistent.

Recent measurements on Sprite suggests that it is better to cache (and invalidate) entire directories rather than individual entries, since a directory is often the region of exploitable locality of reference [14]. This nicely matches the Spritely NFS model; the client simply does an *open* on a directory before doing *readdir* RPCs, and keeps the result of the *readdir* in a cache. When the client removes a directory from its cache, it does a *close* RPC to inform the server. If another client modifies the directory (using an RPC such as *create*, *remove*, *rename*, etc.), then the server does a callback to cause the first client to invalidate its cache.

One open issue is whether a client should write-through any changes (i.e., creations, renames, or removals), or if directory changes can be done using write-behind. My hunch is that the latter is far more complex, especially because it makes it much harder to provide the failure-atomicity guarantee that Unix has traditionally attempted for directory operations. If only write-through is allowed, then *open* on a directory always allows the client to cache; it serves solely to inform the server of which clients might need callbacks when an entry is changed.

In order to avoid the potential for thrashing that would result if a large shared directory was continually being modified by several clients, when a directory is invalidated (via a callback) the client should remember this and not re-cache the directory for a period of several minutes[4]. That is, when an invalidate occurs, the client should *close* the directory (so that the server will not need to do further callbacks) and not cache the result of *readdir* during the inhibition period.

There are a few issues that arise which prevent a client from simply replacing a bunch of *lookup* RPCs with one *readdir*:

- **Lack of attributes and filehandle information**: The NFS *lookup* RPC returns a filehandle and file attributes value that is not returned by *readdir*.
- **Lack of symbolic link values**: If the directory entry is a symbolic link, the client must do a *readlink* to resolve the reference.
- **Extra delay**: When a client wants to lookup one entry in a large directory, it should not have to wait for the server to provide the entire directory contents before continuing.

Together, these suggest the following approach, assuming a cold cache:

1. The client starts by doing the *lookup* RPC that it would normally do.
2. In parallel, it performs an *open* RPC and then a *readdir* RPC (or a series of *readdir* RPCs, if the directory is large) to load the directory into the cache.
3. The result of the *lookup* is inserted into the cache, in such a way that it can be invalidated if the server does a callback on this directory.
4. If the *lookup* indicates that the file is a symbolic link, the client does a *readlink* and inserts that into the cache.

Once the directory is opened, the client can cache the results of *lookup* and *readlink* RPCs, since if anyone else modifies the directory the client will receive a callback.

One would think that doing the *readdir* RPC is unnecessary, since the information returned by *readdir* does not obviate the need to do *lookup* (and perhaps *readlink*). This is not entirely true; the reason is that applications (especially the shells) often attempt to lookup files that do not exist. Without the full set of names in the directory cache, the client cannot avoid going to the server to *lookup* these non-existent names. The consistency guarantee provided by Spritely NFS ensures that such ‘‘negative’’ caching is accurate, and so many *lookup* RPCs can be avoided.

4.3. Caching attributes for unopened files

Spritely NFS provided consistency for file attributes (length, protection, modification time, etc.) only for open files. Clients, however, often use the attributes of files that they won’t (or can’t) open; commands such as `make`, `ls -l`, or `du` fall into this category. Because such commands are so frequent, NFS implementations are forced to provide ‘‘attribute caching’’ using a probabilistic consistency mechanism: cached attributes time out after a few seconds. The timeout is often based on the age of the file; for files that have not recently been modified, the timeout is extended.

[4] This is one of those parameters that I wanted to avoid, but it can be set fairly long because during this period the directory will simply revert to the normal uncached NFS behavior. Traces of real directory activity might help here.

Experience with NFS has shown that such weak consistency usually sufficient, because relatively few applications depend on strong consistency for unopened files. (Weak attribute-consistency on open files causes errors when two client hosts simultaneously attempt to append to the same file, since they have an inconsistent view of the length of the file.)

It is possible to support strong consistency in Spritely NFS, by providing a mode of the *open* RPC that says ‘‘I am reading the attributes of this file, not its data.’’ A client would be allowed to open a file for attributes-read even if it were not allowed to open it for data-read, just as in the local Unix file system.

Any attempt to change the attributes of a file would cause the server to do an implicit ‘‘open for attributes-write,’’ causing the appropriate callbacks to invalidate cached attributes. That is, there is no explicit open-for-attribute-write operation, because explicit attribute-writes are rare. Implicit attribute modification, such as a file length change caused by a *write* RPC, does cause a cache-invalidation callback, but the invalidation should happen only once per open-for-attributes-read.

A client will often want to read the attributes for all the files in a directory, and it does not make sense to read each of them one using a separate RPC. For this reason, and also to streamline the recovery process (see section 6.3), the Spritely NFS *open* RPC could allow ‘‘batched’’ operation, taking a list of files to open rather than just one. (Note that the *open* RPC returns the current attributes, so there is no need to do a subsequent *getattr* RPC.)

John Ousterhout suggests that a new form of the *readdir* RPC be added, which instead of just returning a set of file names would return a set of (*file name*, *file attributes*) pairs. An attempt by another client to modify any of the files would cause a callback referring to the directory, rather to the file itself. This might be tricky to implement because the server would have to maintain a map between files and the directories that reference them, and the client would have to manage two different kinds of cached attributes.

A more straightforward approach would be to use a new *statdir* RPC, which returns a set of (*file name*, *file attributes*, *file handle*, *caching info*) tuples. Effectively, the client would open all the files in the directory for attributes-read access, and would obtain all the attribute values at the same time. Given the 8k byte limit on NFS RPC packets, in one RPC a client could read all the names and attributes for a directory containing about 60 entries. The problem with this approach is that it forces the client to close each of these file handles at some subsequent point. A batched form of the *close* RPC would avoid excess network traffic, although the client code to decide when to close a set of attributes might be rather involved.

It is not clear that strong consistency is really necessary, and it appears to be expensive to provide. However, it is feasible and if supported by the protocol, it could be enabled at the client’s option. (A client that does not participate would not cause inconsistencies in other client caches, since all explicit attribute changes are write-through.)

4.4. Unmounting a server filesystem

A system manager sometimes must ‘‘unmount’’ a local file system at a file server, either to work on that file system or to cleanly shut down the entire system. With NFS, this is no problem; the unmounted file system looks like a dead server, so the client simply keeps trying. With delayed-write-after-close, however, one would like the client to be able to get the dirty blocks onto the disk before unmounting it (lest the client go down before the disk is mounted again).

More generally, the clients should discover that the file system is being unmounted, so that the server can release file references for those files that are no longer actually open. In standard Unix local-filesystem practice, one cannot unmount a file system with active references. One could preserve this behavior in Spritely NFS, but that would make system management more complex (although Spritely NFS, unlike NFS, can tell the system manager exactly which clients are using a file system). Or, one could implement an option to the unmount operation that would force clients to close their open files.

Since a Spritely NFS server can issue callbacks to its clients, it is fairly easy to implement whatever policy seems most appropriate. When the local file system receives an unmount request, it should call a special routine in the Spritely NFS server to say ‘‘this file system is about to be unmounted.’’ Spritely NFS can then do callbacks to all clients with open files on that file system, simply to force them to write back their dirty blocks, or to force them to close their files. Once this is done, Spritely NFS returns control to the local-disk file system, which can then proceed with the unmount operation.

One possible approach is that the ‘‘unmount’’ callback causes the clients to act as if the server for this file system is down. Once the disk is remounted, the server can then enter into a simplified version of the recovery protocol (described in section 6.3); in the meantime, the server can release all its state-table references into the file system (because they will be rebuilt during the recovery phase, just as if the server had crashed and rebooted).

5. Overview of the recovery protocol

Several different recovery mechanisms might have been used for Spritely NFS. The original recovery mechanism used in Sprite [17] depends on a facility implemented in the RPC layer, that allows the clients and servers to keep track of the up/down state of their peers. When a client sees a server come up, the Sprite layer then reopens all of its files.

This approach provides more general recovery support than is needed for Spritely NFS, and it has several drawbacks. First, it would require changes to the RPC protocol now used with NFS, some additional overhead on each RPC call, and some additional timer manipulation on the client. In other words, it complicates the client implementation, which is something we wish to avoid. Second, recent experience at Berkeley [4] has shown that such a ‘‘client-centric’’ approach can cause massive congestion of a recovering server. Sun RPC has no way to flow-control the actions of lots of independent clients (a negative-acknowledgement mechanism was added to Sprite’s RPC protocol to avoid server congestion [4]).

Third, the server has no way of knowing for sure when all the clients have contacted it; even if all the clients actually respond quickly, the server still must wait for the longest reasonable client timeout interval in case some client hasn't yet tried to recover. This can make fast recovery impossible. Fourth, if a partition occurs during the recovery phase, partitioned clients may never discover that they have inconsistent consistency state.

Another possible approach is the ‘‘leases’’ mechanism [6]. A lease is a promise from the server to the client that, for a specified period of time, the client has the right to cache a file. The client must either renew the lease or stop caching the file before the lease expires. Since the server controls the maximum duration of a lease, recovery is trivial: once rebooted, the server simply refuses to issue any new leases for a period equal to the maximum lease duration. A server will renew existing leases during this period (which works as long as clients do not cheat); the clients will continually retry lease renewals at the appropriate interval. Once the recovery period has expired, no old lease can conflict with any new lease, and so no server state need be rebuilt.

The problem with leases is that they do not easily support write-behind. Consider what can happen if a client holding dirty data is partitioned from the server during the recovery phase (not an unlikely event, since a network router or bridge might be knocked out by the same problem that causes a server crash), or if the server is simply too overloaded to renew all the leases before they expire. In such a case, the client is left holding the bag; the server will have honored its promise not to issue a conflicting lease, but will not have given the client a useful chance to write back its dirty data before a conflict might result.

Another potential problem with leases is that the duration of a lease is a parameter that must be chosen by the server. The correct choice of this parameter is a compromise between the amount of lease-renewal traffic and the period during which a recovering server cannot issue new leases, and it is unlikely that the average system manager will be able to make the right choice. The original Sprite protocol has a similar parameter, the interval between ‘‘are you alive’’ null RPCs, which again trades off extra traffic against the duration of the recovery phase. We would like to avoid all unnecessary parameters in the protocol, since these force people to make choices that might well be wrong. (Also, timer-based mechanisms require increased timer complexity on the client.)

The current proposal is a ‘‘server-centric’’ mechanism, similar to one being implemented for Sprite, that relies on a small amount of non-volatile state maintained by the server [1]. The idea is that in normal operation, the server keeps track of which clients are using Spritely NFS; during recovery, the server then contacts these clients and tells them what to do. Since the recovery phase is entirely controlled by the server, there is less chance for congestion (the server controls the rate at which its resources are used). More important, the client complexity is minimal: rather than managing timers and making decisions, all client behavior during recovery is in response to server instructions. That is, the clients require no autonomous ‘‘intelligence’’ to participate in the recovery protocol.

For this to work, the use of stable storage for server state must be quite limited, both in space and in update rate. The rate of reads need not be limited, since a volatile cache can satisfy those with low overhead. Stable storage might be kept in a non-volatile RAM (NVRAM) (such as PrestoServe™), but if the update rate is low enough it is just as easy to keep this in a small disk file, managed by a daemon process. Updates to this disk file might delay certain RPC responses by a few tens of milliseconds, but (as you will see) such updates are extremely rare.

6. Details of the recovery protocol

6.1. Normal operation

The stable storage used in this protocol is simply a list of client hosts, with a few extra bits of information associated with each client. One is a flag saying if this client is an NFS client or a Spritely NFS client. Only Spritely NFS clients participate in the recovery protocol, but we keep a list of NFS clients because this could be quite useful to a system manager. Another flag records whether the client was unresponsive during a recovery phase or callback RPC; this allows us to report to the client all network partitions, once they are healed.

During normal operation, the server maintains the client list by monitoring all RPC operations. If a previously unknown client makes an *open* RPC, then it is obviously a Spritely NFS system. If a previously unknown client makes any file-manipulating RPC before doing an open, then it is an NFS client. One can devise a set of procedures to handle the cases of clients that start out using one protocol and switch to the other (perhaps as the result of a reboot); that is covered in section 7.1.

The client list changes only when a new client arrives (or changes between NFS and Spritely NFS). This is an extremely rare event (most servers are never exposed to more than a few hundred clients) and so it does not matter how expensive it is. My assumption is that it is comparable in cost to a few disk accesses; that is, not noticeable compared to the basic cost of an *open* or file access.

On the other hand, the server must check the cached copy of the client list on almost every RPC. This should be doable in a very few instructions, if the client list is kept in the right sort of data structure (such as a broad hash table). The overhead should be less than is required to maintain the usual NFS transaction cache.

Note that the server's volatile copy of the client list need not contain the entire list of clients, but could be managed as an LRU cache, as long as it is big enough to contain the working set of active clients. This might conserve memory if there are a lot of inactive clients on the list.

If a client fails to respond to a callback (or during the recovery phase, described later) then the server marks it as ‘‘embargoed.’’ This could be because the client has crashed, but it might be because the client has been partitioned from the server. When an embargoed client tries to contact the server, the server responds with a callback RPC to inform the client that it was partitioned during an operation that might have invalidated its consistency state; once the client replys to this callback RPC, the server clears the embargoed bit[5]. The client thus knows that its state is inconsistent, and can take action to repair things (or at least report the problem to the user).

[5] And returns an error response to the original RPC?

Some additional design work is required to figure out how best to reestablish a consistent state after an embargo is lifted. One approach would be to follow a ‘‘scorched earth’’ policy, in which both the client and server return to an initial condition. However, it is probably possible (using generation numbers) for a client to detect which of its open or cached files are still consistent, and only scorch those files which actually have conflicts. The ‘‘reintegration’’ techniques used in the Coda system [13] might also prove useful.

6.2. Client crash recovery

When a client crashes and reboots, the server will be left believing that it has files open even though it does not. This could lead to false conflicts and thus reduced caching. The server will discover some false conflicts when it does a callback and the client says ‘‘but I don’t have that file open.’’ In other cases the false conflict would not cause a callback (i.e., if caching is already disabled). Also, these ‘‘false opens’’ clutter up the server’s state table and space reservation count.

A client can ameliorate these problems by issuing a special ‘‘I just rebooted’’ RPC the first time it mounts each file system after a reboot. The server uses this RPC to force a close on all the files that were open from that client, and also to reclaim all disk space reserved for that client’s dirty blocks. (If the client uses non-volatile RAM technology to keep dirty blocks across a crash, it should also use it to preserve a list of all dirty file handles, and write back the dirty blocks before sending the ‘‘I just rebooted’’ RPC.)

If a client reboots while a server is down (or unreachable because of a network partition), the client simply keeps retrying its mount operation until the server recovers (or becomes reachable), just as is done in NFS.

6.3. The server recovery phase

When a server crashes and reboots, it enters a recovery phase consisting of several steps. The server herds the clients through these steps by issuing a series of recovery-specific callback RPCs, each of which requires a simple response from the client. The steps are:

1. **Read the client list from stable storage**: The server obtains the client list from stable storage and loads it into a volatile cache.

2. **Initiate recovery**: The server contacts each Spritely NFS client on the client list[6]. The callback tells the client that the recovery phase is starting; until the recovery termination step is complete, clients are not allowed to do new opens or closes, and cannot perform any data operations on existing files.

   When a client responds to this RPC, the server knows that the client is participating in the recovery protocol; clients that do not respond are marked as embargoed.

[6] Should it contact embargoed clients? If so, then we need to let the client know at this point that it has already been embargoed, and that will complicate the rest of the recovery process. If not, then the damage from a partition may be compounded even further. Simplicity, and an aversion to incurring timeouts, suggests that we should not bother to contact already-embargoed clients at this point.

During the rest of recovery, embargoed clients are ignored and we can assume that the other clients will respond promptly, but during this step long timeouts may be needed. On the other hand, this step can be done for all clients in parallel, so the worst-case length of this step is only slightly longer than the maximum timeout period for deciding that a client is down or partitioned.

At the end of this step, we can update the stable-storage client list to reflect our current notion of each client's status.

3. **Rebuild consistency state**: The server contacts each non-embargoed client and instructs it to reopen all of the files that it currently has open, using the same open-mode (read-only or read-write) that was used before. If the clients do not cheat, the resulting opens will have no conflicts, since before the server crashed there were no conflicts and no new opens could have taken place since the crash.

   Since each server may have to open multiple files, and since file-open operations are moderately expensive (requiring manipulation of the state table), the server may want to do these callbacks serially rather that in parallel (or semi-parallel, to limit the load to a reasonable value). This should not result in too much delay, since we are reasonably sure that the clients involved will respond.

   Note that a client may hold dirty blocks for files that it does not actually have ‘‘open’’. This means that there must be a special mode of the *open* RPC, used only for ‘‘reopening’’ closed-dirty files. It might be better to define a special *reopen* RPC that allows the client to reopen several files in a single network interaction. Otherwise, the RPC layer could become the bottleneck during recovery. (Or, the basic *open* RPC could allow batched operation, which might be useful in providing consistent attributes caching as in section 4.3.)

   A client responds to this callback only when it has successfully reopened all of its open files. If a client fails to respond, then the server marks it as embargoed and updates the stable-storage list.

   Once this phase is done, the server has a complete and consistent state table, listing all of the open and closed-dirty files.

4. **Terminate recovery**: The final step is to contact each client to inform it that recovery is over. Once a client receives this RPC, it can do any operation it wants. As in the recovery initiation step, the server can do these callbacks in parallel, but in any case the clients are unlikely to timeout so the duration of this step should be brief.

One issue that I have not yet considered is what might happen if a server crashes during its recovery phase. It may be that a simple crash-generation number scheme, passed with the crash-recovery callbacks, will allow the clients to keep track of what is really going on.

6.4. Log-based recovery

Because the recovery protocol is server-centric, it leaves the implementor of the server a lot of freedom to choose different strategies. V. Srinivasan has pointed out that nothing in the protocol prevents the server from using additional stable storage to obviate part or all of the recovery protocol.

The server could, for example, log all opens and closes to stable storage. Since the ‘‘open lifetime’’ of files is fairly short (often less than 100 milliseconds [2]) it would not make sense to log every such event to disk. Instead, the server could keep the head of the log in NVRAM, which would allow it to elide the short-lived opens before writing the log to the disk. Some sort of log-cleaning algorithm, analogous to that required by a log-structured file system [12], would be necessary. Alternatively, the on-disk information could be structured as a database, which would take more work to update but which would not need cleaning. Using the log or database, the server could recover its consistency state without any help from the clients.

A much simpler approach would be to keep track, in the client list, of those clients that have any files open at all. During crash recovery, the server could ignore any client known to have no open files, thus speeding recovery and perhaps avoiding timeouts for clients that have been removed from service. This modification would increase the update rate for the stable-storage copy of client list. However, the server could delay the update on a client’s last close, anticipating a subsequent open in the near future, because this would not affect the correct behavior of the recovery protocol. A delay interval of, say, one minute would probably avoid almost all extra updates without significantly increasing the cost of recovery.

6.5. Recovery and the ENOSPC problem

Since the server host’s count of reserved disk blocks may be updated quite often, it does not make sense to keep it in stable storage. (Maintaining a stable accurate value could approximately double the latency of disk writes for closed-dirty files.) Instead, we can recompute this value during the recovery phase. When recovery starts, we set the value to zero. We then have the clients tell us what their reservation needs are, either by an extra argument to the RPC call for ‘‘reopening’’ closed-dirty files, or perhaps by having the client also explicitly close those files (since the *close* RPC always carries a reservation request). Once recovery is done, we have a consistent count of the total reservation requirements.

Note that during recovery, a client cannot simply request a reservation for the number of dirty blocks it currently holds, because this number might have increased since the server crashed. Instead, the client must remember the reservation it has left as the result of a normal *close* RPC, and use this value when ‘‘re-opening’’ a closed-dirty file.

If the network is partitioned during recovery, we might end up in a state where the server does not know of a client’s reservation requirements, and so gives the space away once recovery is over. If the partition heals, we may discover that no conflicting open prevents the embargoed client from writing its dirty blocks, but there is no longer any space to hold them.

One (rather crude) approach to this problem is to set aside some disk space in anticipation of this problem. For example, some file systems, such as the Berkeley Fast File System [9], reserve a certain amount of free space in order to obtain better performance. This so-called ‘‘minfree zone,’’ which can be used by super-user processes on normal Unix systems, might also be employed to store blocks written back from embargoed clients. However, this is at best a stopgap solution and can lead to some tricky management problems: what do you do when this space runs out?

6.6. Multiple file systems exported by one server host

NFS servers traditionally export more than one file system; a large server might export dozens of file systems. This allows a network manager to reconfigure servers without broad disruption, and to tailor security controls according to the nature of the data being protected.

The recovery protocol described in this paper is most simply understood as applying to each individual file system. That has two implications:

1. One client list is maintained for each file system, not just one for each server host.
2. The steps in the recovery protocol must be repeated for each file system exported by a server. (The server should not attempt to contact clients that failed to respond during an earlier iteration.)

The latter requirement is not actually a serious problem. The initial and final phases, during which all clients are contacted in parallel, can be done in parallel for the various file systems. The number of packets exchanged in these steps will increase, but the number of packets exchanged as the clients reopen their files will not be affected.

The former requirement increases the amount of storage space needed to keep client lists, both stable storage and in-memory cache. Since these lists are not likely to be very large, keeping one for each file system should not be too wasteful. The alternative, keeping one list for the entire server, could add run-time complexity, since it is conceivable that a given client might mount one file system via Spritely NFS, and another file system from the same server via pure NFS. This would force the server to use a complex data structure to represent a client list, reducing the incentive to keep one list instead of several.

It is tempting to try to avoid these requirements by making the recovery protocol messages express things in terms of server hosts rather than server file systems. That is, the three callback types used in the recovery process inform a client that it should perform a particular function for all the files opened from a specified server host, rather than a specified file system. I believe this will work, but some problems might be lurking in the background.

7. Simultaneous mixing of NFS and Spritely NFS hosts

I argued that without a path for incremental adoption, users will have little incentive to install Spritely NFS, because all-at-once changeovers cause major disruption. A sudden change to a new, untried system makes system managers nervous.

Spritely NFS clients and server can easily coexist with pure NFS hosts. The two problems to solve are automatic configuration (so that network managers need not worry about who is running what) and maintenance of consistency guarantees (so that NFS clients get at least the level of consistency that they would if all clients were using NFS).

7.1. Automatic recognition of Spritely NFS hosts

It is possible for Spritely NFS clients and servers to ‘‘recognize’’ each other in a sea of NFS hosts. Suppose that Spritely NFS were to use the same RPC program number as NFS; we can establish a set of rules that will allow Spritely NFS hosts to discover if their peers speak Spritely NFS or just NFS.

Consider a Spritely NFS client. It need not know if the server supports Spritely NFS (i.e., cache-consistency protocols) until it wants to open a file. At that point, it simply issues its *open* RPC. If the server speaks only NFS, it will respond to this with a PROC_UNAVAIL error code. The client can then cache this fact (in a per-filesystem data structure) and treat the file system as a pure NFS service.

A Spritely NFS server recognizes Spritely NFS clients because they issue *open* RPCs before issuing any file-manipulating RPCs. Thus, when a new client is added to the server's client list, the RPC that causes this addition also tells the server what kind of client is involved.

If we want to follow this ‘‘automatic’’ scheme for recognizing Spritely NFS hosts, then it should work even with a client or server changes flavors. In principle, client changes should be easy to detect, since a client changing from NFS to Spritely NFS will issue an *open* RPC, and the server can check on each *open* to make sure that its client list records the client as a Spritely NFS host. A client changing back to NFS would reveal itself, sooner or later, by using a file that it had not previously opened. This trick requires the server to check a file's consistency state on every RPC, which is otherwise unnecessary (this is normally the job of the client) and could add some slight overhead.

If a Spritely NFS server changes back to an NFS server, the Spritely NFS clients will detect this as soon as they do an *open* or *close* operation. If an NFS server changes into a Spritely NFS server, however, the clients might not realize this immediately. It might be possible for the server to signal its nature by the use of a callback, but this could cause problems to pure-NFS clients that are not expecting any callbacks.

Another approach, instead of basing automatic recognition on RPC procedure types, is to use a separate RPC program number for Spritely NFS. This makes the server's task a lot easier; it simply distinguishes clients based on which program number they use. The server would not have to check to see if a client had previously opened a file in order to catch transitions between Spritely NFS and pure NFS. This does not, however, solve the problem of how a client realizes that a server has changed from NFS to Spritely NFS.

7.2. Consistency between NFS and Spritely NFS clients

When NFS and Spritely NFS clients are sharing a Spritely NFS file system, the NFS clients will not have the same consistency guarantees as Spritely NFS clients. However, the Spritely NFS server can guarantee the NFS clients no-worse-than-NFS consistency, by treating each NFS operation as if it were bracketed by an implicit pair of *open* and *close* operations.

In other words, if an NFS client reads from a file which is write-cached by a Spritely NFS client, the server first does a callback on the Spritely NFS client to obtain the dirty blocks. If an NFS client writes a file cached by a Spritely NFS client, the server does a callback on the Spritely NFS client to invalidate its cache. If this reduces performance too badly, perhaps Spritely NFS callbacks should optionally specify a particular block to flush or invalidate ... but I suspect that if such NFS-to-Spritely-NFS write-sharing happens at all, then it is likely to involve most of the blocks in a file.

8. Performance

Our original goal with Spritely NFS was to improve performance over NFS. Since NFS does not need to support a recovery protocol, we must show that the added recovery overhead in Spritely NFS does not eliminate our advantage. Note that the original, non-recovering version of Spritely NFS did better than NFS on realistic benchmarks even though NFS does not have to do any *open* and *close* RPCs; that is, Spritely NFS saves enough through better use of the client cache to make up for the extra RPCs.

The recovery protocol has two kinds of costs: in normal operation, there is a small overhead on each RPC, and after a server crash, there is a recovery phase. Since NFS has no recovery phase, it will always be faster at continuing after a server reboot. These should be rare events, so the cost of recovery will be amortized over a long period of useful work. At any rate, the server-centric approach should allow us to do efficient recovery, since we are not put at risk of server overload during the recovery phase.

The per-RPC overhead comes from the maintenance of the client list. I argued earlier that this is negligible; most of the time, we simply do a hash-table lookup to discover that the client is already known and not embargoed[7]. Very rarely, we must update stable storage, but it is unlikely that a server would see such a high rate of new clients that this becomes a measurable overhead. In short, I do not think the per-RPC overhead will cause a measurable difference in Spritely NFS performance.

9. Software complexity

Since I have described this recovery protocol as ‘‘simple,’’ it seems appropriate to describe how much work it would take to convert an NFS implementation to support Spritely NFS with recovery. Note that the original Spritely NFS implementation was written in the course of a month or so by a programmer who had never before studied the Unix kernel. See [15] for details.

9.1. Client implementation issues

Starting with a client NFS implementation, the modifications necessary to support Spritely NFS are fairly simple. The *open* and *close* operations have to be implemented, the per-file data structures need to include cachability information, and the data access paths need to observe the cachability information. A daemon, patterned closely on the existing NFS server daemon, needs to be added to handle callback requests, along with ‘‘server’’ code to respond to the callbacks.

Very few changes are needed in other components of the client operating system. The code that manages the table of open and closed files (the equivalent of the Unix *inode* table) must inform the Spritely NFS client when a closed file is being removed from this table to make room for a new entry. It is also useful to provide a mechanism to remove dirty blocks from the file

[7]The per-RPC operations in the original Sprite recovery mechanism apparently made a small but measurable difference in the RPC overhead. This might have been because on each RPC request and reply, the code was forced to manipulate timers.

cache, for use when the file that contains those blocks is deleted (this improves performances by eliminating useless write-backs).

9.2. Server implementation issues

The changes to the server are obviously more extensive. For Spritely NFS without recovery, the changes were quite localized: the existing RPC server procedures were not touched, and all the new code related to handling the *open* and *close* RPCs and performing callbacks. Spritely NFS requires a small amount of stable storage to support the ‘‘generation number’’ mechanism used to detect certain conflicts. One 64-bit value kept per file system (or even per server), and updated every hour or so, should suffice.

To support recovery, the server code for all NFS operations has to check the client list on each RPC, and perhaps call functions to maintain the client list or do necessary callbacks.

Most of the complexity in the recovery protocol can be implemented in user-mode code. The Spritely NFS kernel code would have to provide some hooks so that the recovery process can disable the servicing of certain RPCs during the recovery phase.

In order to provide full consistency between Spritely NFS clients and local file system applications running on the server, there will have to be some linkages between the local file system’s *open* and *close* operations and the Spritely NFS state-table mechanism. For example, when a local process opens a file, this might require Spritely NFS to change a client’s cachability information for that file. The local file system must also support the disk-space reservation scheme described in section 4.1; this means providing a special form of the *write* operation that decrements the reservation.

10. Other related work

Several interesting papers related to recovery in distributed file systems have never been published. Rick Macklem worked on ‘‘Not Quite NFS,’’ an attempt to use the leases model to provide recovery for an NFS extended with a Sprite-like consistency protocol. Meanwhile, the Echo file system project at Digital’s Systems Research Center has grappled with a number of similar issues, especially those related to write-behind [5, 7, 16].

Mary Baker and Mark Sullivan describe a similar approach to state recovery [3], using a ‘‘recovery box’’: stable storage for selected pieces of system state, to allow a system to reboot quickly. In their approach, a file server would store all the open file handles in stable storage, with the assumption that these are unlikely to be corrupted by (or just prior to) a crash. My proposal is more conservative, both in that it does not require low-latency stable storage, and because it makes far weaker assumptions about the effects of a crash. Their proposal, however, leads to much quicker recovery.

11. Summary

Spritely NFS was an interesting experiment, but without a recovery protocol it is not suitable for production use. The recovery protocol proposed in this paper, together with tying up some loose ends, should be enough to make Spritely NFS a real alternative to NFS. The mechanism is so simple, especially on the client side, that one can no longer claim that only a stateless protocol admits a simple implementation.

Even if Spritely NFS never becomes a real system, I believe that this bare-bones approach to recovery will be useful in other contexts. A similar approach is being used now in Sprite, and their experiences should validate the design.

Acknowledgements

The design in this paper has evolved (sometimes rather discontinuously) in lengthy exchanges among many people, including (in alphabetical order) Mary Baker, Cary Gray, Chet Juszczak, Rick Macklem, Larry McVoy, John Ousterhout, V. Srinivasan, Garret Swart, and Brent Welch. Most of these people have talked me out of at least one bad idea.

References

[1] Mary G. Baker. Private communication. 1991.

[2] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proc. 13th Symposium on Operating Systems Principles*, pages 198-212. Pacific Grove, CA, October, 1991.

[3] Mary Baker and Mark Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the Unix Environment. In *Proc. Summer 1992 USENIX Conference*. San Antonio, Texas, June, 1992. To appear.

[4] Mary Baker and John Ousterhout. Availability in the Sprite Distributed File System. *Operating Systems Review* 25(2):95-98, April, 1991.

[5] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo Distributed File System. In preparation. 1992.

[6] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. 12th Symposium on Operating Systems Principles*, pages 202-210. Litchfield Park, AZ, December, 1989.

[7] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and Consistency Tradeoffs in the Echo Distributed File System. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 49-53. Pacific Grove, CA, September, 1989.

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

[9] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems* 2(3):181-197, August, 1984.

[10] 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.

[11] 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.

[12] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation ofa Log-Structured File System. In *Proc. 13th Symposium on Operating Systems Principles*, pages 1-15. Pacific Grove, CA, October, 1991.

[13] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers* 39(4):447-459, April, 1990.

[14] Ken W. Shirriff and John K. Ousterhout. A Trace-Driven Analysis of Name and Attribute Caching in a Distribute System. In *Proc. Winter 1992 USENIX Conference*, pages 315-331. San Francisco, CA, January, 1992.

[15] V. Srinivasan and Jeffrey C. Mogul. Spritely NFS: Experiments with Cache-Consistency Protocols. In *Proc. 12th Symposium on Operating Systems Principles*, pages 45-57. Litchfield Park, AZ, December, 1989.

[16] Garret Swart, Andrew D. Birrell, Andy Hisgen, and Timothy Mann. The Failure Semantics of Write Behind. In preparation. 1992.

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

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

''A Simulation Based Study of TLB Performance.''
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

''Analysis of Power Supply Networks in VLSI Circuits.''
Don Stark.
WRL Research Report 91/3, April 1991.

''TurboChannel T1 Adapter.''
David Boggs.
WRL Research Report 91/4, April 1991.

''Procedure Merging with Instruction Caches.''
Scott McFarling.
WRL Research Report 91/5, March 1991.

''Don't Fidget with Widgets, Draw!.''
Joel Bartlett.
WRL Research Report 91/6, May 1991.

''Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

''Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.''
G. May Yip.
WRL Research Report 91/8, June 1991.

''Interleaved Fin Thermal Connectors for Multichip Modules.''
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

''Experience with a Software-defined Machine Architecture.''
David W. Wall.
WRL Research Report 91/10, August 1991.

''Network Locality at the Scale of Processes.''
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

''Cache Write Policies and Performance.''
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

''Packaging a 150 W Bipolar ECL Microprocessor.''
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

''Observing TCP Dynamics in Real Networks.''
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Technical Note TN-15, December 1990.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Technical Note TN-19, June 1991.

‘‘Unreachable Procedures in Object-oriented Programming.’’
Amitabh Srivastava.
WRL Technical Note TN-21, November 1991.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach’’
John S. Fitch.
WRL Technical Note TN-24, January 1992.

‘‘A Recovery Protocol For Spritely NFS’’
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.