A Caching NFS Client for Linux

Greg J. Badros
gjb@cs.washington.edu

27 November 1999

Abstract

The Linux NFS client suffers from poor performance. As Linux has become more popular, its primitive NFS client implementation has outgrown its usefulness. We describe numerous enhancements to the Linux NFS client that improve performance. Specifically, we discuss better lookup and attribute caching, asynchronous writing of data, and local disk caching of data file reads. We describe our implementation, and benchmark its performance against both the existing stable Linux NFS client and the local-disk filesystem, `ext2fs`. Our implementation outperforms the 2.0.x kernel's NFS client in all but one benchmark, and improves on the basic client by up to a factor of 14 when reading from files that are cached locally.

1 Introduction

Unlike more advanced Network File System (NFS) client implementations, such as those by Sun Microsystems for SunOS and Solaris, the Linux 2.0.x NFS client has few performance features. Only asynchronous read-ahead buffering and some rudimentary lookup attribute caching is performed. All writes are done synchronously, resulting in write speeds that can be as low as 25KB/sec,[1] and no caching of read data is performed.

Despite the design limitations of NFS (e.g., no tokens, push-on-close, and stateless servers) commercially-available clients achieve substantially better performance (on the order of MB/sec, not KB/sec) [Mic95]. This paper describes numerous performance improvements to the Linux NFS client which make a notable difference to the usability of NFS-mounted partitions for Linux users. Among these are:

Asynchronous write buffering. Although the Linux 2.1.32 kernel introduced this feature, that kernel was in development at the time this project began, so we integrated the asynchronous writing capabilities of the newer kernels into the 2.0.x kernel series (the original integration is the work of Jan Sanislo, `oystr@cs.washington.edu`).

Caching of `lookup` RPC results. The basic NFS client caches only the 64 most recent filename lookups, and searches linearly through that list. Since `lookup` remote procedure calls are so common [Mac91, p. 61], improvements to lookup caching can noticeably reduce network traffic and server load. Our implementation enlarges the `lookup` results cache to 511 elements and uses a hash-table instead of a linear list. This feature was largely implemented by Doug Zongker, `dougz@cs.washington.edu`.

Local disk caching of reads. Many NFS exported directories contain a large number of files which are mostly read, and written only occasionally.[2] Since many workstations have fast, large, often under-used local disks (see also Minnich's AutoCacher [Min93]), we implement read caching using the local disk as a large repository. We rely on the underlying file system to use delayed writes and its own caching to make writing to the cache efficient.[3]

[1] This speed was observed on an overloaded network where the client and the server were on separate subnets. For comparison, both `rcp` and `ftp` wrote at over 100KB/sec.

[2] In fact, this is common for distributed file systems [WPE+83].

[3] Relative to 10Mbit ethernet, modern SCSI disks (with an efficient file system such as Linux `ext2fs` [CTT96] and bus-mastering PCI controllers) are an order of magnitude faster in performance.

We limit our implementation to use Version 2 of the NFS protocol [Mic89] instead of exploiting some of the possible benefits provided by Version 3 [Mic94] such as readdirplus and commit. We imposed this constraint because few NFS V3 servers exist, and the base Linux NFS implementation does not yet provide full V3 support.

To measure our performance improvements, we use the Andrews File System (AFS) benchmark suite [HKM+88] along with some larger test cases, and some micro-benchmarks. In addition to measuring real time, we count the number of remote procedure calls of various types. Since RPCs are so expensive (especially on a slow network), they tend to dominate the performance of an NFS client. Our goal has been to reduce RPCs.

The next section discusses the design and overall architecture of our implementation. Section 3 describes the details of the asynchronous writing and lookup caching implementations, and section 4 explains the implementation of read caching. Section 5 describes how we measured the performance of our implementation, and shows the results of our benchmarks. Finally, sections 6 and 7 describe some possibilities for future work and summarize the findings of this project.

2 Design and Architecture

Our primary design goal was to improve performance of the NFS client by reducing the number of remote procedure calls through aggressive caching of lookups, attributes, and data pages. Our other design goals were to: 1) minimize the impact on non-NFS filesystem performance; 2) minimize the increase in kernel size and memory requirements; 3) allow cached data to persist across reboots; 4) not require a separate partition for the cache files; 5) permit the cache files to be stored on an arbitrary filesystem (i.e., to not assume `ext2fs`); 6) not decrease performance dramatically for any special cases (e.g., not use whole file caching similar to AFS); and 7) allow sufficient customization of the cache parameters at module insertion time. Our implementation achieves all of these goals.

Because was interested in performance, the bulk of our implementation is a module that resides in the Linux kernel. The asynchronous writing implementation, the caching of lookup results and attributes (section 3.2, on page 3), and the caching of read data pages (section 4, on page 4) to local disk are all performed inside that NFS module.

The read caching is implemented at the virtual filesystem's readpage level, so the granularity of the local disk cache is the page size (4 KB). New data pages from an NFS server are written to the local disk cache as they are read. The cached files on local disk may be sparse—pages of files in the cache are written as they are received from the server. A simple bitmap resides in kernel memory to track the valid pages of each locally cached file. When a page is requested, if it has been cached locally, the read is served from the local disk after ensuring that the file has not changed on the server (by checking the modification time and file size).

Limitations in the NFS protocol make it wasteful to update the local disk cache on writes, since the next read will notice that the modification time on the server's inode has changed, and we have no way of knowing if we were the only client to have altered that inode in the interim. Thus, our cache file becomes invalid. Instead, we simply mark the cache file as invalid when we write to the remote file it is caching. See section 6.1, on page 16, for more discussion of this issue.

Ideally, locally cached data pages should be available even across reboots as long as the server's file has not changed. To support this feature, an implementation would need to persist the information about which pages are cached. However, that complication is substantial and would hinder performance since more "maintenance" data would need to be written each time a remote file's page is copied to the local cache. Instead, we mark cached files as complete when we have cached all of its pages locally. We then permit only those marked files to be used after a reboot. This approach reduces the required maintenance upon reads of individual pages, but gives up the ability to cache files that are not read completely. However, executable binaries generally are paged-in dynamically—they are often not read in their entirety. If an NFS partition held mostly executables, our design as described might only be able to retain a small fraction of its cached data pages between reboots.

To combat the negative affects from partially-cached files that only rarely get read in their entirety, our implementation provides a kernel thread to "fill in" missing pages of cache files. When network traffic is

low, our `nfsfillind` daemon reads previously unread (and therefore uncached) pages from the server, thus eliminating the holes in a cached file so that we can mark the cache file as complete (see section 4.2, on page 8).

For ease of development, testing, and debugging, we have used privileged user-level programs where possible to simplify the code that must live in the kernel. Specifically, we have extended the user-level `mount` utility to understand our cache parameters (see section 4.1, on page 5) and introduced a user-level daemon to free space in the local disk cache when it fills.

Our user-level NFS cache cleaner daemon is called `nccd` (see section 4.3, on page 9). When cache space is exhausted, our basic `nccd` removes the least recently used cache files, thus freeing space for creating new cache files. The kernel and `nccd` communicate via two mechanisms: 1) the kernel informs the cleaner when it needs to remove some old files by writing to a distinguished file in the cache directory; and 2) the `nccd` informs the kernel of changes in the amount of disk space used for each remote via a pseudo-device `/dev/nfs-cache-space` (e.g., at startup to compute the usage of files already in the cache, and after cleaning to report the amount of space freed). Because it is a user-level program, the policy decisions made by `nccd` about how to manage the cache space are easy to experiment with and customize.

3 Asynchronous Writing and Lookup Caching Implementation

3.1 Asynchronous Writing

The Linux 2.0.x NFS client issues a synchronous remote procedure call for each write system call. This is inefficient when performing many small writes in succession. Unfortunately, numerous small sequential writes is a common situation as many programs (e.g., `gcc` and `ld`) simplify their output loops by expecting that the underlying file system will merge write requests through some kind of buffering. Not merging write requests results in unacceptable performance and is corrected in versions of the kernel since 2.1.32. Although our implementation pre-dates the stable 2.2.x kernel series, we me integrated those changes to the NFS client into the 2.0.27 kernel that we targeted. See the 2.2.x kernel series for details of the improved asynchronous write behaviour.

3.2 Lookup Caching

Lookups of files and retrieval of file attributes constitute a significant fraction of NFS traffic.[4] Clients can reduce this traffic by caching file and directory attributes for a few seconds, so that getting attributes for files that have been recently accessed can be performed without communicating with the server.

The Linux 2.0.27 implementation of lookup caching has two shortcomings. Accessing the cache is expensive since a linear search is used to find elements (requiring a complete pass for each cache miss). Partially because of this slow search, the size of the cache is fixed at 64 entries. This small cache size limits the number of hits, reducing the effectiveness of the cache in avoiding RPCs. Since the lookup-cache is a system-wide resource, the small 64 element cache is easily exhausted. By increasing the number of cached attributes, we will observe fewer cache misses which will reduce network traffic and server load.[5]

We have replaced the linear array with a hashing scheme to allow for more efficient operations, thus reducing client CPU usage and permitting larger cache sizes. The hash value is computed from the directory inode and the filename. Collisions are resolved with side chaining. To avoid a kmalloc() call for each cache entry insertion, the cache is preallocated in a single block and initially organized into a free list. If we attempt to add an entry to the cache and the free list is exhausted, a scavenging pass runs over all the entries in the cache and moves expired ones to the free list. In the unlikely event that this scavenging fails to produce any free slots, the new entry is dropped instead of being added to the cache. It is common when searching the cache for a file to find an expired entry for that file. In this case the entry is immediately moved to the free list, so that when the lookup RPC completes, the insertion will always find a free spot without scavenging.

[4] In addition, improved read caching (see section 4, on page 4) will strictly increase the fraction of all RPCs that are lookups or getattrs.

[5] Lazowska et al. noted that the server CPU is the primary bottleneck for scaling distributed file systems [LZCZ86].

A complication in lookup caching is that the cache can be accessed in two ways. The usual method is to lookup the file inode given the directory inode and the filename; the cache is optimized to handle this case efficiently. The second access method is when we have the file inode and want to update the file attributes field stored in the lookup cache. This case occurs, for instance, when a read RPC returns the current file attributes along with the read data. We avoid this case whenever possible because it requires a linear-time search of the entire cache. The need to perform a linear search limits how large we can make the cache before the client CPU time spent in these searches overwhelms the advantage in lookup speed. (We chose not to provide a reverse-hash to avoid requiring even more precious kernel memory for the lookup cache.)

4 Read Caching Implementation

The Linux 2.0.x NFS client translates directly between the VFS interface and the NFS protocol (which is basically an RPC-based version of the standard UNIX file system operations).[6] We intercept readpage system calls, and attempt to pass them off to the caching filesystem (i.e., `ext2fs` on the local disk). See Figure 1 for details of how page reads are satisfied. Files are cached (as decided at the "Should we cache file?" decisions in the flowchart) only if they are larger than a specified minimum size (currently 1 page, 4KB), smaller than a specified maximum size (currently 4096 pages, 16 MB), and not changed recently (currently, this timeout is 30 seconds). When a cached file is subsequently read, those reads are served from the local cache file instead of sending read requests to the server.

We cannot avoid all server traffic even when reading unchanged files: the stateless nature of the NFS protocol requires us to confirm with the server that the file has not changed since we last read it. However, when reading large files, the majority of the disk traffic is due to many read RPCs, all of which are eliminated.[7]

All cached file data exists only in disk-based structures. Each remote file from which a page is cached has a corresponding local cache file. The name of the local file is the concatenation of the server name, the server's superblock number, and the inode number of the file being cached. For example a local cache file called `holden,2-23` stores page data from the remote file with inode number 23 on the server named "holden" in the partition whose superblock is on device 2. These files are all stored in a single-level directory structure. We exploit `ext2fs`'s ability to efficiently store a sparse file (i.e., one where only a small portion of the total blocks have had any data written to them), and store only those data blocks that the client has actually read (not whole files). Also, after a read is satisfied out of the local disk cache, we send a setattr to the server to update its last-accessed time (`atime`) if the prior access was more than 30 seconds ago. Since the disk cache can persists for months or longer, it is important that the access times are accurate.[8]

Obviously, files may not be read in their entirety (e.g., executables, which are paged-in on demand). Thus, we must maintain in-kernel data structures to track which pages of each inode have been cached to local disk. We use a packed binary array representation, and also include the number of total pages, and the number valid, along with some information needed by `nfsfillind` to finishing reading a file after the NFS inode may have left memory. See Figure 2 for details. When we recognize that a file has had its last page written to the local cache (done in constant time with the count of valid pages, not with the bitmap), we mark the cache file as complete using its `u+x` mode attribute bit, and can then deallocate the bitmap of valid pages.

To support the relationship between an inode for a file on the server and the inode of the file on the local disk that is caching the remote file, we made two significant changes to the kernel's data structures: 1) all inode's now have a pointer to a structure about the inode they are caching; and 2) all inode's can specify a clear_inode_hook to be called when that inode is chosen to be reused ("putting" an inode to a 0 count does not remove it from memory). See Figure 3 for details.

Because files may change on the server (due to either our machine or another client on the system), we must also invalidate cache entries occasionally. When we notice that the NFS inode that we are caching has a new modification time or file size, we mark our local cache as invalid by turning off its `u+x` bit (if it was

[6] In fact, Sun's initial VFS interface (and corresponding Vnode—virtual node—interface) were introduced to support their NFS implementation [SGK+85, p. 124].

[7] Like most other implementations, we permit use of cached lookup RPC results to reduce the number of times we must ask the server if the file has not changed to once every 3-5 seconds.

[8] Surprisingly, neither Linux nor Solaris clients update the access time upon reading a page from the VFS memory cache.

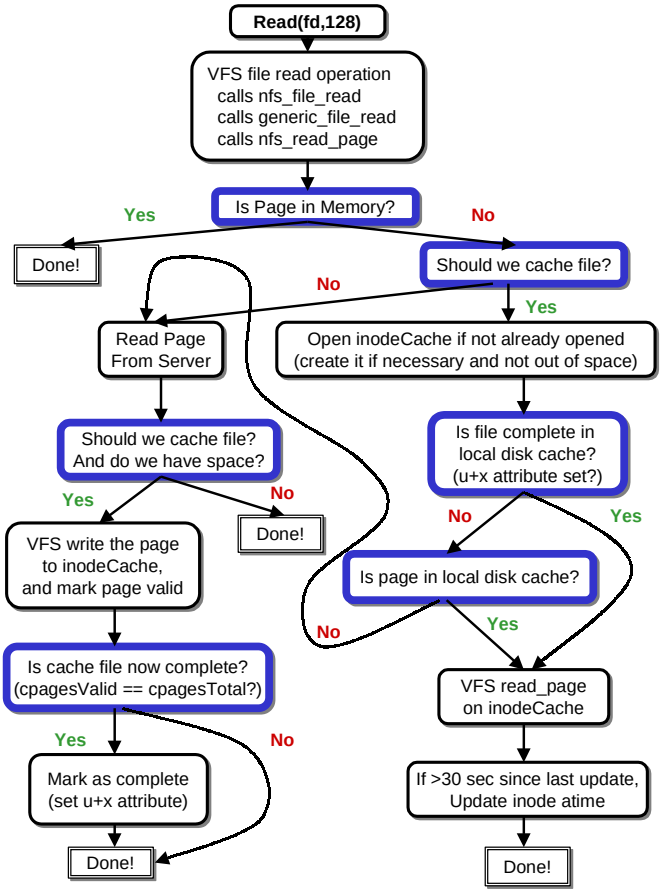

Figure 1: Read caching flowchart. Heavy boxes are decision points.

complete) and updating the in-kernel data structures (e.g., removing the bitmap, and resetting the count of valid pages to zero). If the cache file's inode is later cleared from memory without having read sufficiently many pages to justify filling in, the cache file is unlinked, and the space is reused.

Ideally we would be able to update the disk cache for local writes. However, the NFS protocol has no way of letting a host know that it is the only writer to the file.[9] After changing only a single byte of a 4MB file that exists in our local disk cache, all the NFS client can subsequently tell when it reads that file again is that the modification time (`mtime`) has changed. Because we cannot conclusively confirm that it was only our client that affected the change, we must invalidate all of the pages we had cached locally. This is the principal reason why our implementation does not cache files that have changed recently.

4.1 Mount Parameters for Caching

Our NFS client permits flexible control over the cache parameters. We provide several new configurable arguments which can be specified in the usual way in the `/etc/fstab` file and our enhanced `mount` utility exposes those parameters to our NFS client via the `nfs_server` structure. Each remote filesystem specifies these parameters independently.

The parameters that we have added to configure our caching scheme are:

[9] This deficiency is removed in NFS V3; see section 6.1, on page 16 for details.

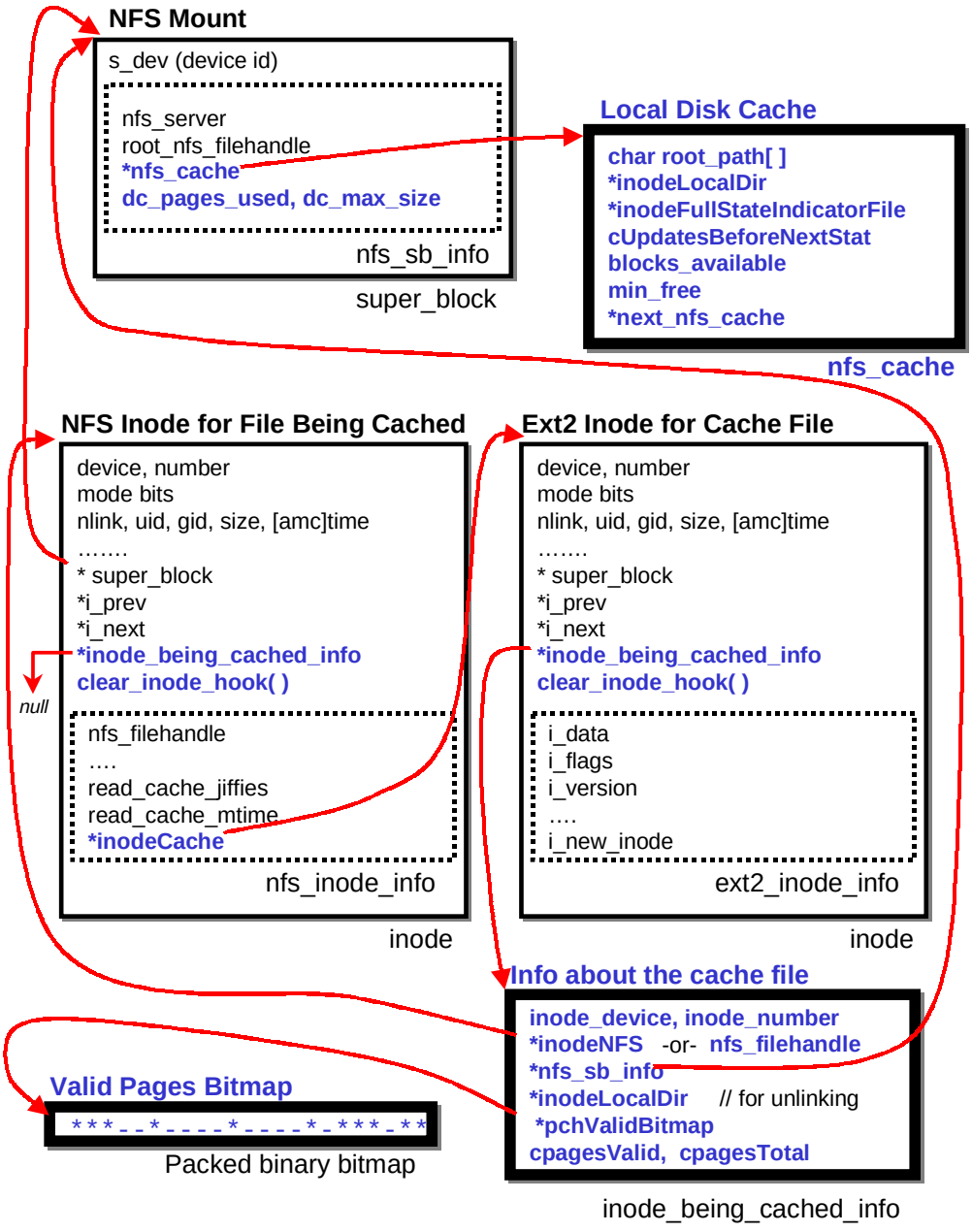

Figure 2: Kernel data structures. Bold text denotes new data members, heavy boxes denote added data structures.

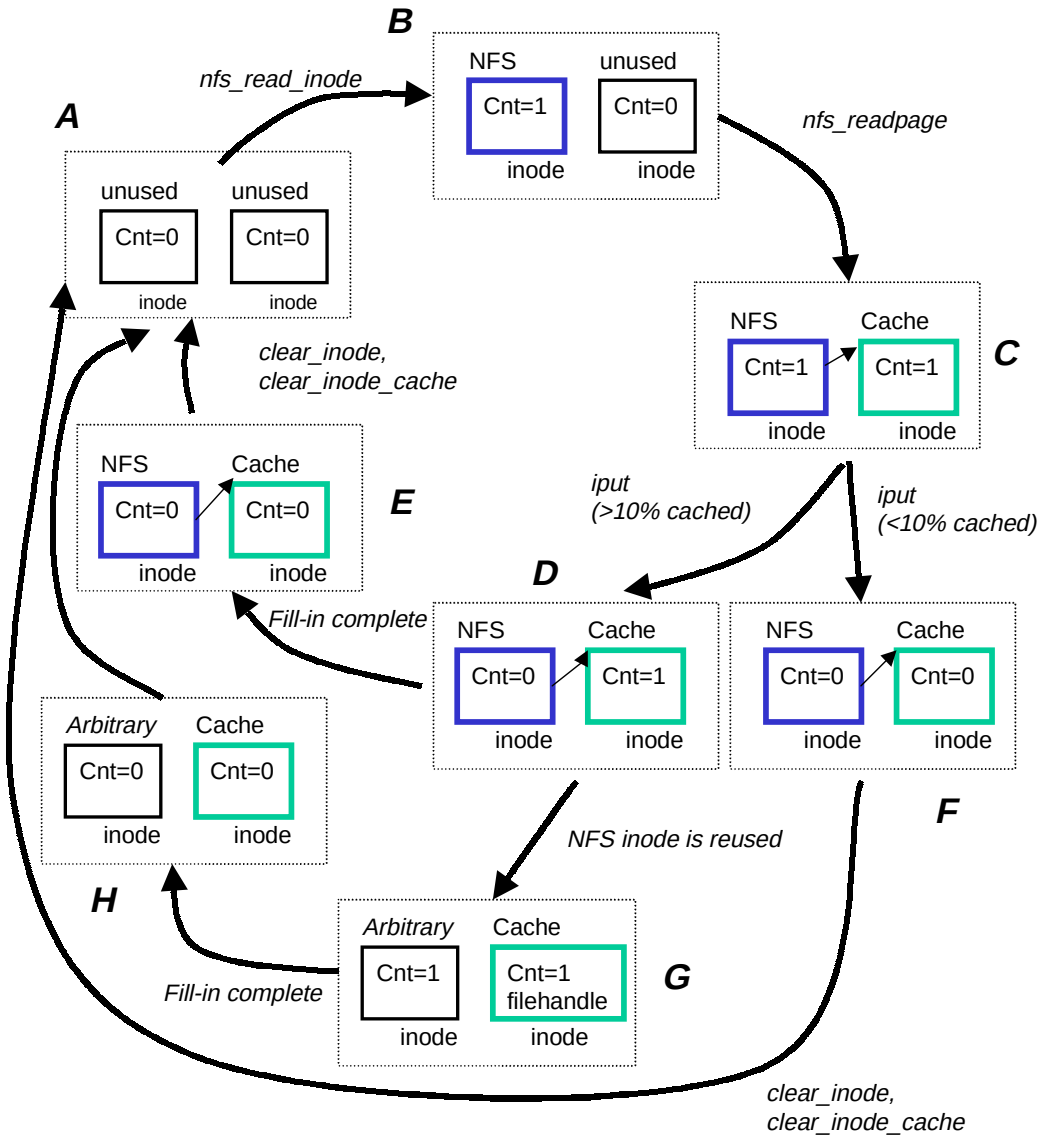

Figure 3: The life-cycle of an inode, and its corresponding cache file's inode. From **A** to **B** a remote file is first accessed, thus assigning an inode structure to that file. From **B** to **C** the first page is read from that remote file, so another inode is assigned to the corresponding local cache file; the NFS inode keeps a pointer to the cache inode so read pages can be written to the local disk. We stay in state **C** while the file is open, and then move to either **D** or **F** when the closing of the remote file results in *put*ting of the NFS inode. When the remote file is closed, we choose to discard the cached pages if we've only read less than 10% of the file—that case corresponds to **F**, where we *put* the cache inode back on the free list, and can reuse both inodes (back to state **A**). If we read more that 10% of the file, we move instead from **C** to **D**. While in **D**, the `nfsfillind` reads subsequent pages of the file in the background until the local cache contains all of the remote file's data pages. When the cache file is complete, we progress to **E**, where we have *put* the cache inode back on the free list, and can then return to **A** after calling the appropriate inode-clearing hooks. States **G** and **H** represent a slight complication of the filling-in procedure when the inode that was used for the remote file needs to be reused.

mount syntax	Internal name	Description
[no]dc_enable	NFS_MOUNT_DC_ENABLED	Enables caching
[no]dc_read	NFS_MOUNT_DC_READS	Enables reading from the cache (but not updating)
dc_root=<dir>	disk_cache_root	Location of disk cache
dc_min_free=#	disk_cache_min_free	Min free space for that cache partition
dc_max_size=#	disk_cache_max_size	Maximum cached data for this partition (kb)

The first option, `dc_enable`, controls whether files read from the remote partition are to be cached. By selectively enabling caching for remote partitions, a system administrator can choose not to cache certain partitions which may be written to frequently. Such partitions are not likely to benefit from a local cache. The second option is related: `dc_read`, permits a partition to have files already cached locally be used, but not update the cache when remote files changes, or a new remote file is accessed. This option is especially useful during debugging.

The next two options, `disk_cache_root` and `disk_cache_min_free`, specify the directory where the cache files should live and how much space must minimally remain free on the local disk partition where that directory resides. Different remote partitions may be cached in different directories, but do not need to be. The local cache location is just an ordinary directory that is dedicated to holding cache files and information. For example, if a host has a large `/tmp` partition, a system administrator could create `/tmp/.nfs-cache` and use that for the NFS cache—no special partitioning or reformatting is required.

Finally, the `disk_cache_max_size` option permits specifying the maximum amount of local disk space to use for caching a given remote partition. Using this parameter, a system administrator can allocate more local disk space for heavily caching an important remote partition, or limit the amount of local disk space a rarely-needed partition is allowed to consume.

The flexibility of our caching scheme permits the caching NFS client to be introduced into a production system with minimal hassle and effort.

4.2 Filling in Partially Cached files

Our design strongly prefers completely cached files: such files do not require a kernel structure to track the valid pages, they can persist across reboots, and they do not increase pressure on the limited in-memory inode resources (while a file is being filled in, two inode structures are maintained—one for the remote file, and one for the local cache file). However, many files, including the all-important dynamically paged-in executables, are not read in their entirety. To combat the possibility that long-term operation could lead to a cache full of mostly-complete files, we created a way to bridge the gap between NFS-like caching at page granularity for performance of small reads on many files and AFS-like caching of whole files for long-term local disk storage.

To accomplish this, we have a background kernel thread called `nfsfillind` which looks for partially cached files and attempts to take advantage of idle time to fill them in. When network traffic is low, this thread periodically looks at the set of partially cached files. It chooses one that would benefit from being filled-in (the simple metric we use is fewest-remaining pages first, with ties broken by total file size) and reads in an unread page, saving it to the local disk cache. In this way, we avoid insisting that the first read of a file get all the pages from the server immediately, but instead choose to get the rest of a file in the background.

This "fill-in" process will continue as long as there are partially cached files.[10] In order to allow partially cached files to be kept and filled in even if the original user process which read the file has closed it, the inode for the local cache file is augmented with the NFS file handle for the file. Using this handle, the `nfsfillind` process can read pages from the server even if the inode representing the NFS version of the file has left memory.[11]

[10] In our architecture, it does not make sense to stop filling in because of space concerns, because the cleaning process can only operate on completely cached files. See section 4.3, on page 9.

[11] Fill-in using the filehandle after the NFS inode has left memory is not yet implemented.

4.3 NFS Cache Cleaner

When the cache fills, some space needs to be freed to permit caching of more recently-accessed files to continue (instead of just turning caching off). Our implementation presupposes that filling the disk is reasonably rare, and that the policy for determining which cache files to remove should be left up to the local system administrator.

Thus, we have developed a simple interface for a user-level "cleaner" program that the kernel awakens when the cache fills. This process searches the cache directories, decides which files to remove, removes them, and tells the kernel how much space has been freed. The exact sequence of events is:

1. The kernel realizes that it has run out of space. It does this by tracking of the number of cache pages being used per mounted filesystem. On each write to the cache, the client checks for either of the following cases:

   (a) the number of cached pages for this remote partition exceeds the `disk_cache_max_size` parameter specified when mounting; or

   (b) the number of free pages on the cache filesystem (i.e., the local disk partition) is less than the minimum free space permitted (the maximum of the `disk_cache_min_free` parameters for all of the mounted filesystems which are sharing this cache directory).

2. The kernel writes a record of the violation to the file `<cache_root>/.FULL`. The violation is specified as a single line of information that the user-level cleaning process will need to do its job:

   (a) In the case of maximum space exceeded, the current time and the mounted filesystem (a server name and server device) are written to `.FULL`, and the group execute bit (`g+x`) is set on `.FULL`.

   (b) In the case of a minimum free violation, only the current time is written (a minimum free violation is viewed as being shared by all filesystems in the cache), and the owner execute bit (`o+x`) is set on `.FULL`. Setting the owner execute bit disables further caching, thus avoiding an out-of-disk-space error.

3. A user-level cache cleaning process notices the change in the `.FULL` file (it should normally block in a loop checking the file size periodically—alternatively, the kernel could send the cleaner process a signal). It then reads the list of violations (there may be more than one by the time the cleaner has awakened) and handles the full partition(s). Our sample cache cleaning daemon, `nccd`, operates as follows:

   (a) candidates for removal are identified:

   - when the maximum space for a remote partition is exceeded, only fully-cached files from that partition are considered for removal; or
   - when the minimum free space for a local disk is violated, all fully cached files in the cache directory are considered equally.

   (b) This set of candidates is then sorted by access time and the oldest 10% (by size) are removed. The special kernel-generated pseudo-file `/proc/net/nfs-mounts` provides information about the space consumption of the various mounted partitions so that the cleaning daemon can determine how much space to free.

   Note that only files that are completely cached are considered for removal. This constraint simplifies freeing space because fully cached files have no special kernel data structures associated with them. As a result, the user-level cleaning daemon can safely remove these files itself, rather than having to request that the kernel do this on its behalf. The limitation of not permitting removal of partially-cached files is not a problem in practice for two reasons: 1) the size of the cache is substantially greater than the average size of the files in it; and 2) the fill-in thread (see section 4.2, on page 8) is working to keep most of the cached files fully cached.

The first five tests are the five phases of the Andrews benchmark set:	
andrew: making directories	Creates an empty 21-directory hierarchy.
andrew: copying files	Fills the created directory structure by copying 71 files into it.
andrew: statting	Recurses through the directories twice, generating a `stat` system call for every file.
andrew: intensive reading	Recurses through the directories twice, searching every file for a given string.
andrew: cpu intensive	Builds a moderately sized package within the directory structure.
The next five tests work with large files:	
untar big package	Untars a 4.3MB archive containing approximately 400 files and directories.
repeated ls-ing	Lists directories totaling 500 files four times.
read a big file	Reads a 6.1MB file.
read a big file again	Reads the same 6.1MB file again.
copy a big file	Makes a copy of a different 6.1MB file.
The last three tests perform smaller reads and writes:	
random reads	Performs 1,000 reads, ranging in size from 1 to 2,663 bytes, from randomly chosen locations within a 4.3MB file.
small writes	Performs 2,000 writes, appending the integers 1 to 2,000 (as strings) to a file.
small reads and writes	Like the small writes test, but rewinding and rereading the entire file between each write (and performing only 1000 writes). This is intended to show worst-case performance for our caching scheme (since the read cache is invalidated on a write, all the work of storing reads on local disk is wasted).

Table 1: Descriptions of the benchmark suite used in performance evaluation.

4. After the cleaning daemon removes the chosen files, it must inform the kernel of the results of its actions. For each mounted filesystem from which files were removed, the cleaner must write a single line record to the pseudo-device `/dev/nfs-cache-space` with the filesystem identifier (hostname and device number pair) and the number of pages freed. This step ensures that the kernel is able to maintain accurate information about how much disk space the cache consumes without requiring it to ever scan the entire directory for usage information.[12]

5. Finally, the cleaning daemon should clear the execute bits on `.FULL` to re-enable caching.

By specifying a clean, simple interface between the kernel and a user-level process, we permit sophisticated custom cache cleaning policies to be used. At the same time, this design decision dramatically simplifies the in-kernel code at little performance cost.

5 Benchmarks and Results

To measure the performance of our NFS client, we measure elapsed time and RPC traffic for a set of 13 benchmarks. The thirteen tests we used are listed in Table 1. Each benchmark was run once by itself, and once with 4 copies of the benchmark running in parallel from different top level directories. All single runs were executed, followed immediately by the 13 parallel runs—the disk cache was not emptied in between. We compare three implementations: the standard 2.0.27 NFS client, our enhanced NFS client, and the local disk using `ext2fs`.

[12]This same interface is used at boot time (actually, just after the NFS client module is inserted) to inform the kernel of space consumed by cache files that persisted across a reboot.

For all the benchmarks we used an isolated 10Mbit ethernet local area network with four hosts connected via an eight port Kingston EtherX workgroup hub. The server was a 486/DX4-100 with 40MB of memory, an ISA bus, and a 3Com 3c509 ethernet card running a standard Linux 2.0.29 kernel from a RedHat 4.0 distribution. The test partition we exported was from a 1GB IDE drive. No other disk activity was taking place on the server during the test.

The test client was a Pentium 200 with 20MB of memory available, a PCI bus with a Adaptec 2940UW SCSI controller, a Intel EtherExpress 100 ethernet card, and a 4GB wide SCSI disk. It was running either a Linux 2.0.27 kernel patched with upgraded EtherExpress100 and Adaptec 2940UW drivers (this configuration was also used when benchmarking the local disk performance) or our enhanced kernel with the same patches and our improved NFS client module. The cache directory for our enhanced client was the same local disk as used when measuring the local disk performance.

The two remaining machines on the network (a 486/DX4-75 and a Pentium 166, both with 3c509 ethernet cards) were used to generate additional network traffic to slow down the network. The `ping` utility was used to flood the network with `icmp` packets between those two hosts. Additionally, the server was using `ping` to flood the network to simulate handling multiple hosts. This only reduced worst case round trip time to about 4ms (average round trip time as reported by `ping` was 0.7ms). The slower machine was used as the server to more accurately represent the round-trip times on a reasonably-loaded server. This network environment used for our measurements was still far faster than the target network of the University of Washington's computer science department.[13]

For the two NFS benchmark runs (standard and enhanced) we used a 5 second time-out for cached file attributes (chosen because that is the timeout the BSD implementation uses [Mac91, p. 54]). We used a 30 second time-out for directory attributes as in the original Sun implementation [SGK+85]. Note that increasing the expiration time further increases pressure on the cache since entries persist for almost twice as long.

Though our client machine actually had 256MB of main memory available, we constrained it (via a `lilo` configuration option) to use only the first 20MB of memory. This limits the size of the VFS-level memory page buffering and reduces its affect on the runs. Additionally, it simulates the more realistic scenario of requiring the majority of main memory for the actual workload (few users have 200MB of main memory available for caching of disk pages). To further reduce the affect of VFS-level memory page buffering, we read almost 8MB of unrelated files from the local disk between each pair of the 26 benchmarks (those pages replace the relevant pages from the prior test that might benefit the following benchmark stage).

All of the remote procedure call data was collected using `tcpdump` running in raw mode on the client. That data was then analyzed after the benchmarks completed.

Figures 4 and 5 shows elapsed time for each of the benchmark tests on the standard Linux 2.0.x NFS client, our enhanced NFS client, and the local disk using `ext2fs`. Figures 6 and 7 compare RPC requests generated by the standard client versus our enhanced client.

Figures 6 and 7 illustrate that our enhanced client never generates more RPC traffic than the standard client. In particular, the figures show that whenever a file is cached and re-read, all the `read` RPCs are eliminated.[14] The figures also demonstrate the tremendous advantage asynchronous writing provides for the "small writes" and "small reads and writes" benchmarks.

The reduction in RPCs is reflected in figures 4 and 5. For all the tests except the parallel "andrew: copying files" and "random reads," our enhanced client outperformed the standard NFS client. In those two tests benchmarks we were only 3% and 8% slower respectively. This difference is attributable to the overhead in caching files to local disk, and to measurement error. In all other tests (and notably all single tests) we outperform the standard NFS client by as much as factor of 14 when reading already cached files, and almost a factor of 100 for "small writes x 4."

Much of the benefit from tests that did not directly result from the local disk caching came from reduced `getattr` RPCs resulting from our larger cache, and from our fixing a performance bug in the standard NFS client which resulted in its not exploiting attributes returned as a side effect of other RPCs. For one of the

[13] Worst case round trip times on the University of Washington computer science department network are often around 300ms when communicating between subnets, with an average of more than 15ms during normal workday traffic.

[14] Since the cache is not cleared between the single run benchmarks and the parallel runs, both the parallel "read a big file x 4" and "read a big file again x 4" tests are serviced from the local disk cache—only the single "read a big file" test accesses the file from the server.

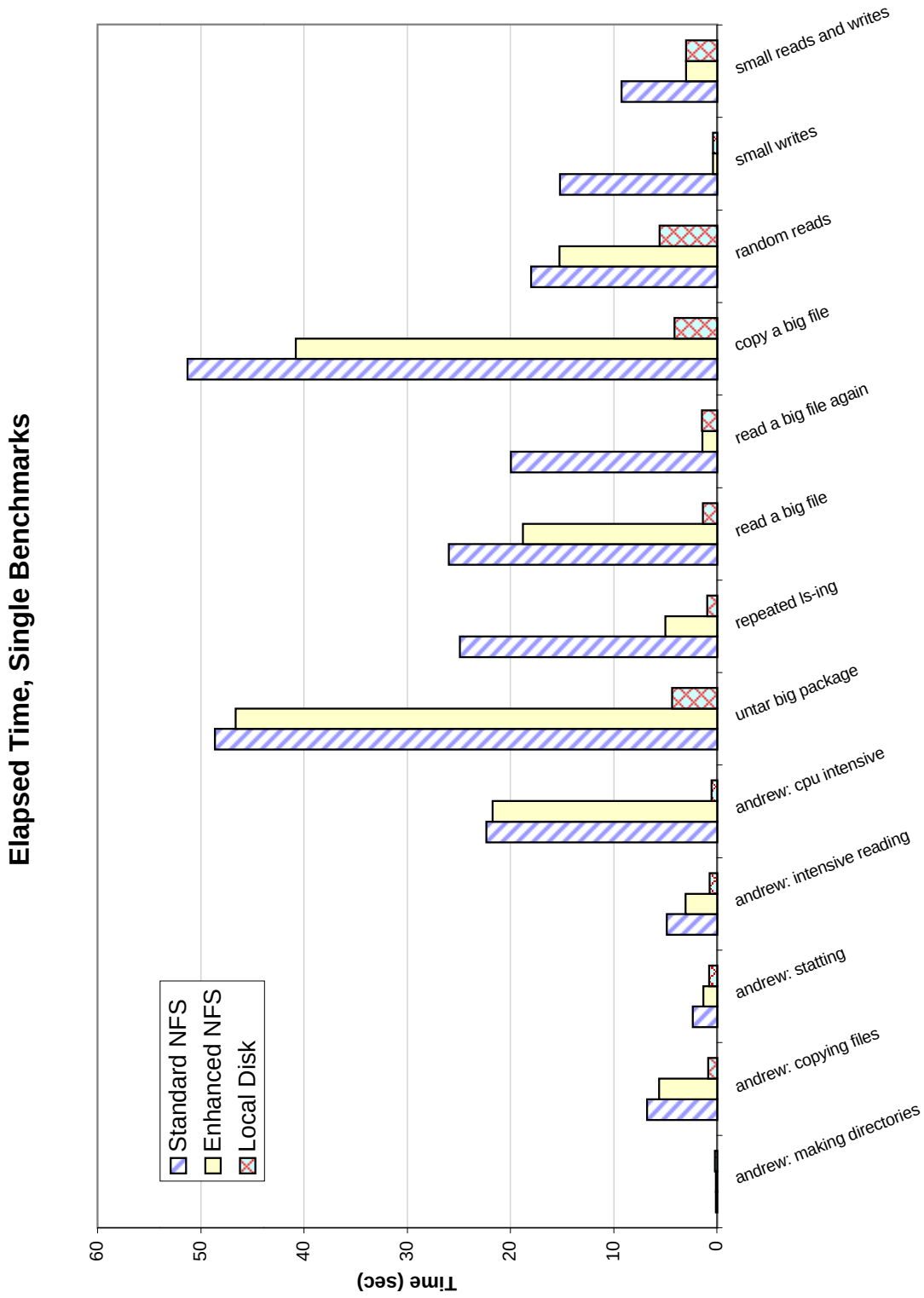

Figure 4: Elapsed time for single benchmarks

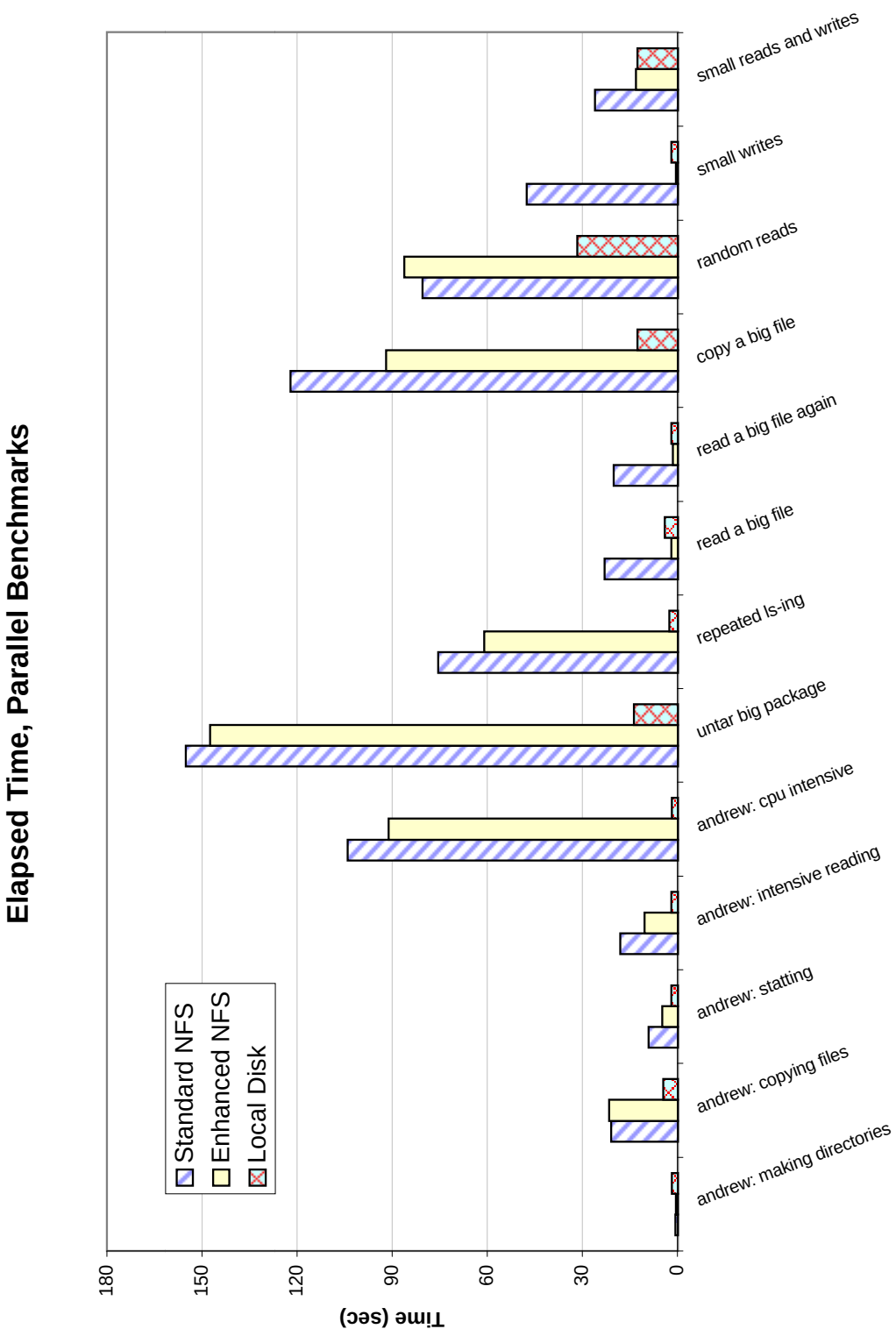

Figure 5: Elapsed time for multiple copies of the benchmarks running in parallel

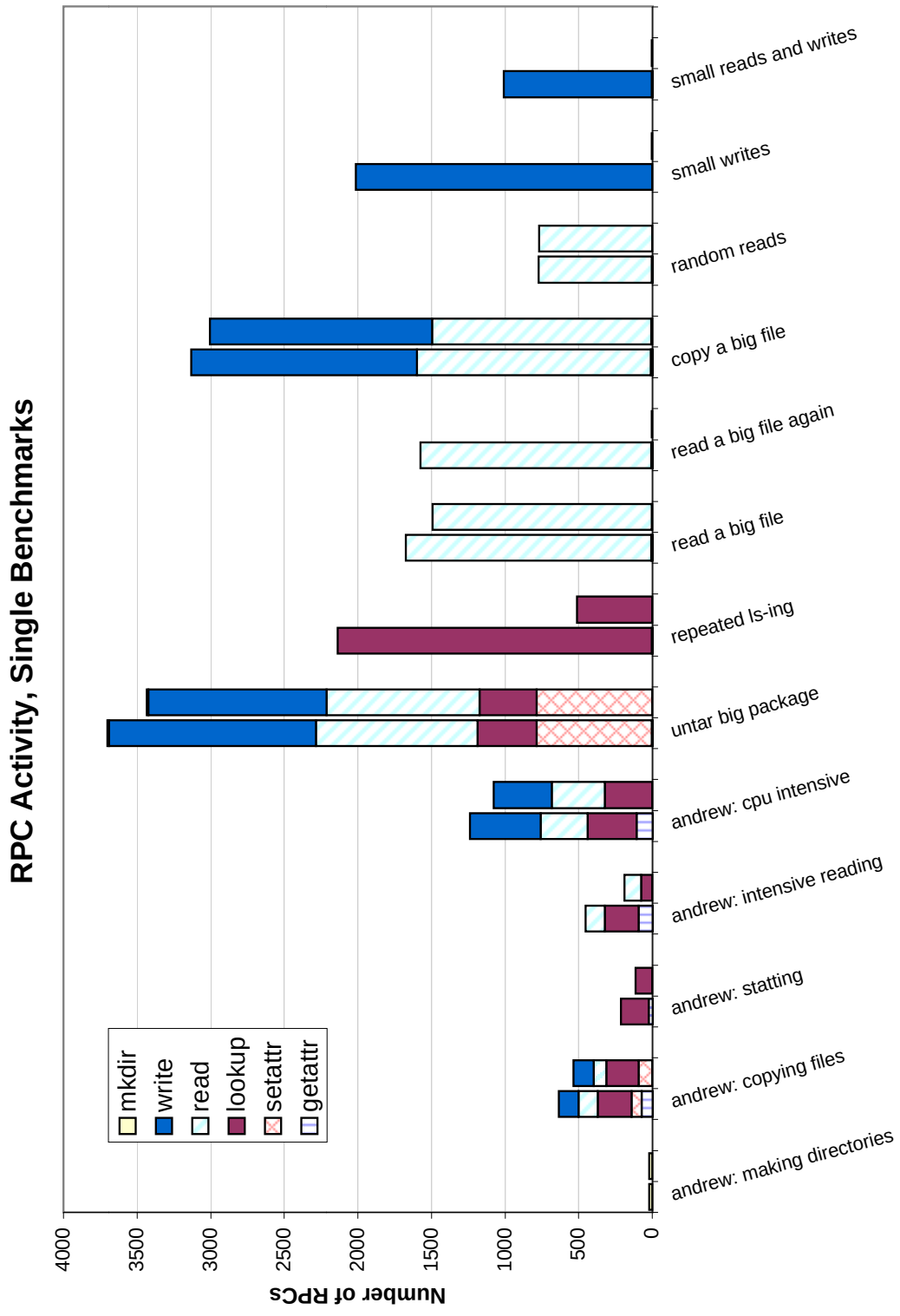

Figure 6: RPC activity during single benchmarks

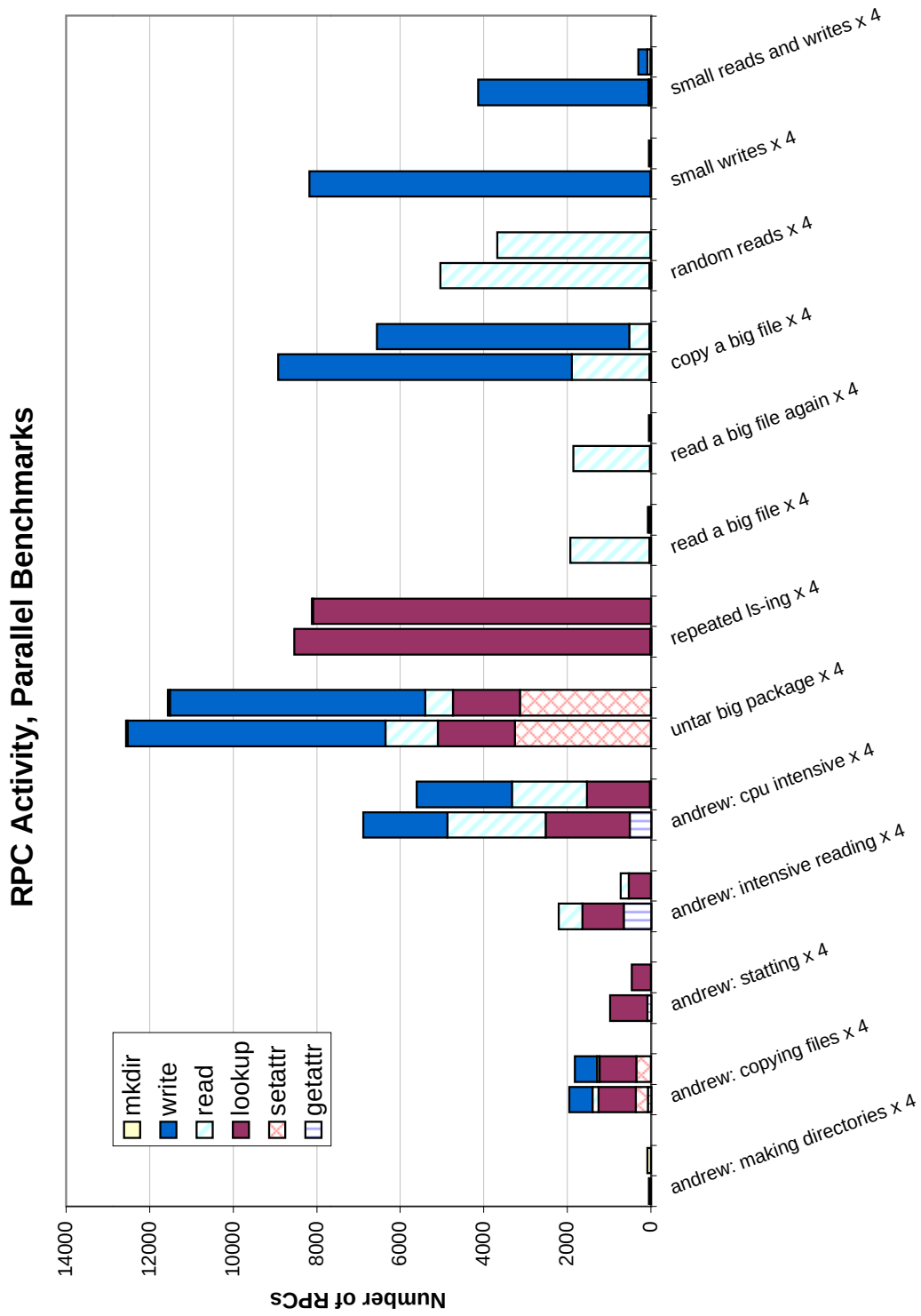

Figure 7: RPC activity during parallel benchmarks

most realistic benchmarks (but one that we did not specifically target for our enhancements), the "andrew: cpu intensive x 4," we observed a better than 10% performance improvement over the standard NFS client.

As mentioned previously, real-world networks are often far slower than the network we used for benchmarking. The relative performance of our enhanced client becomes even more impressive when the relative cost of a remote procedure call increases due to more network traffic or a more distant connection between client and server.

6 Future Work

6.1 Write Operations

There is substantial room for improvement in the write subsystem of the Linux NFS client. Because of the NFS requirement that the server commit writes to stable storage, little can be done from the client side to reduce real transfer time (aside from hardware solutions like Presto and NFS V3's additional commit RPC). A partially attainable goal is improved perceived response time: ultimately we still have to send the bits, but we could extend the use of delayed-writes to let the user program that issued the write to continue computing while the data is being sent.

There are two interesting issues with writing in general and asynchronous writing in particular that need to be addressed. The first concern is the interaction between reads and writes. Logically, a read issued on a client immediately after a write on that client should, independent of the server state and the world view of other clients, read the new data. To do otherwise would weaken the already weak NFS consistency model further than is reasonable. However, the simple "solution," whereby data is first written to the local cache and later sent to the server from there, cannot work in the NFS model. When a new page of a file is written to the server, the timestamp on the server's file will be changed. To the client, the file looks changed, but it cannot tell if the change is the result of the update it just sent (assuming it remembers that it just wrote a page) or from some other host. NFS does not allow the client to discover the instigator of the change, and therefore the client is forced to either invalidate or update its cache as a hedge against the (admittedly unlikely) possibility that the change on the server was not its own. NFS V3 corrects this shortcoming by having write return the prior modified time as well as the new one [PJS+94, p. 142]. If the prior modified time matches the client's, then its write was the only one that transpired. Furthermore, again as a limitation of NFS, the client cannot discover which pages of a file have changed, so if it chooses to update its cache, it will have to re-fetch the entire file. Our current implementation does no caching of writes and invalidates the NFS disk cache whenever data is written to a file.

The second concern with asynchronous writing pertains to closing files. Clients that are especially interested in data integrity will prefer a policy where the `close` operation blocks until all pending writes to the file have actually been written to stable storage on the server. On the other hand, a client concerned more with performance might prefer to return immediately from `close`, and allow the data to be sent to the server later. Although NFS requires the former semantics for `close`, the latter seems useful, so it would make sense to provide a `mount` flag (`[no]dc_force_on_close`) to control this possibly-dangerous behavior.[15]

Some studies suggest that 20-30% of newly-written data is deleted within 30 seconds [NWO88, p. 138]. Naïvely, these data appear to further support delaying writes, because there is a good chance that the file will be removed before we get around to actually performing them. However, in general files are closed before they are deleted, which would cause us to simply wait at that point for the writes to complete.

6.2 A kinder, gentler cleaning paradigm

One consequence of the current cleaning model (see section 4.3, on page 9), is that the cleaner is not invoked until the kernel has actually run out of space for caching. Until the cleaner has finished freeing space, the kernel disables caching. This behaviour puts pressure on the cleaner to react quickly—while it is running

[15] It is dangerous because closed files' changes would not be immediately visible by other hosts that open the recently closed file (and thus NFS disallows such behaviour). One can imagine writing a large file, then `rsh`ing to another machine to continue working on that file. If the close completes and returns to the client before the data exists on the server, the second machine will not see the entire file immediately.

we are effectively operating with a most-recently-used replacement policy, which we know is suboptimal for most work patterns.

One way to relieve this situation is to have the kernel set both a "soft" and a "hard" limit on the cache size. When the soft limit is reached, the cleaner is invoked, but caching is not disabled. In general, the cleaning should complete before we reach the hard limit, at which point the kernel will (for self-defense) stop caching, as it does now. This scheme would relieve the time-pressure on the cleaner and allow it to provide more sophisticated policy decisions, and at the same time reduce (if not eliminate) the time the system spends with caching disabled. Although we have not implemented this extension, it is straightforward to do so.

7 Conclusion

The implementation, as it exists now, is complete and stable. Further investigation of some of the warning messages we log would be useful to gain confidence in the client, but we have not observed data corruption in several weeks of light use. Our enhanced NFS client can substantially improve the performance of Linux boxes that use partitions mounted from a remote NFS server. Ideally, our design could form the basis of a production-quality implementation in the latest development Linux kernel series.

8 Acknowledgements

I thank Doug Zongker and Andy Collins for their implementation efforts and design discussions throughout the graduate operating systems class project which led to this paper. Doug is responsible for the implementation of the larger lookup cache, and Andy wrote the enhanced `mount` utility and a first version of the cache cleaning daemon, `nccd`. I also thank Jan Sanislo for his initial integration of the 2.1.32 NFS client into the 2.0.27 kernel, and for his hardware support and network expertise throughout this project, and Corey Anderson who provided valuable feedback on a draft of this paper. This work was supported by a National Science Foundation Graduate Research Fellowship.

References

[CTT96] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. Web document, 1996. http://www.redhat.com:8080/HyperNews/get/fs/-ext2intro.html.

[HKM+88] 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. In *ACM Transactions on Computer System*, volume 6(1), February 1988.

[LZCZ86] Edward D. Lasowska, John Zahorjan, D. Cheriton, and W. Zwaenepoel. File access performance of diskless workstations. In *ACM Transactions on Computer System*, volume 4(3), pages 238–268, August 1986.

[Mac91] Rick Macklem. Lessons learned tuning the 4.3bsd reno implementation of the NFS protocol. In *Winter USENIX Conference Proceedings*, pages 53–64. USENIX Association, January 1991.

[Mic89] Sun Microsystems. NFS: Network file system version 2 protocol specification. Technical report, Sun Microsystems, Mountain View, CA, March 1989.

[Mic94] Sun Microsystems. NFS: Network file system version 3 protocol specification. Technical report, Sun Microsystems, Mountain View, CA, February 1994.

[Mic95] Sun Microsystems. The NFS distributed file service: NFS white paper. Web document, March 1995. http://www.sun.com/solaris/wp-nfs.

[Min93] Ronald G. Minnich. The AutoCacher: A file cache which operates at the NFS level. In *USENIX Conference Proceedings*, pages 77–83. USENIX Association, Winter 1993.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. In *ACM Transactions on Computer System*, volume 6(1), pages 134–154, February 1988.

[PJS+94] Brian Pawlowski, Chet Juszczak, Peter Saubach, Carl Smith, Diane Lebel, and David Hitz. Nfs version 3 design and implementation. In *1994 Summer USENIX*, pages 137–152. USENIX, June 1994.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *USENIX Conference Proceedings*, pages 119–130. USENIX Association, Summer 1985.

[WPE+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–69, October 1983.