vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O

MING CHEN and GEETIKA BABU BANGERA, Stony Brook University
DEAN HILDEBRAND, IBM Research - Almaden
FARHAAN JALIA, Stony Brook University
GEOFF KUENNING, Harvey Mudd College
HENRY NELSON, Ward Melville High School
EREZ ZADOK, Stony Brook University

Modern systems use networks extensively, accessing both services and storage across local and remote networks. Latency is a key performance challenge, and packing multiple small operations into fewer large ones is an effective way to amortize that cost, especially after years of significant improvement in bandwidth but not latency. To this end, the NFSv4 protocol supports a *compounding* feature to combine multiple operations. Yet compounding has been underused since its conception because the synchronous POSIX file-system API issues only one (small) request at a time.

We propose *vNFS*, an NFSv4.1-compliant client that exposes a vectorized high-level API and leverages NFS *compound procedures* to maximize performance. We designed and implemented vNFS as a user-space RPC library that supports an assortment of bulk operations on multiple files and directories. We found it easy to modify several UNIX utilities, an HTTP/2 server, and Filebench to use vNFS. We evaluated vNFS under a wide range of workloads and network latency conditions, showing that vNFS improves performance even for low-latency networks. On high-latency networks, vNFS can improve performance by as much as two orders of magnitude.

CCS Concepts: • **Information systems → Information storage systems**; **Storage architectures**; **Network attached storage**; • **Networks → Network protocols**; **Network File System (NFS) protocol**;

Additional Key Words and Phrases: NFS, vNFS, compound procedures, POSIX, File-system API

ACM Reference format:
Ming Chen, Geetika Babu Bangera, Dean Hildebrand, Farhaan Jalia, Geoff Kuenning, Henry Nelson, and Erez Zadok. 2017. vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O. *ACM Trans. Storage* 13, 3, Article 21 (September 2017), 24 pages.
https://doi.org/10.1145/3116213

This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; NSF awards CNS-1251137, CNS-1302246, CNS-1305360, and CNS-1622832; and ONR award N00014-16-1-2264.
Authors' addresses: M. Chen, G. Bangera, F. Jalia, and E. Zadok, 349 New Computer Science, Stony Brook University, Stony Brook, NY 11790; emails: {mchen, gbangera, farhaan, ezk}@cs.stonybrook.edu; D. Hildebrand, IBM Research - Almaden, 650 Harry Rd, San Jose, CA 95120; email: dhildeb@us.ibm.com; G. Kuenning, Department of Computer Science, Harvey Mudd College, 301 Platt Blvd., Claremont, CA 91711; email: geoff@cs.hmc.edu; H. Nelson, Ward Melville High School, 380 Old Town Rd, Setauket- East Setauket, NY 11733; email: hcnelson99@gmail.com.

https://doi.org/10.1145/3116213

1 INTRODUCTION AND BACKGROUND

Modern computer hardware supports high parallelism: a smartphone can have eight cores and a NIC can have 256 queues. Although parallelism can improve throughput, many standard software protocols and interfaces are unable to leverage it and are becoming bottlenecks due to serialization of calls [9, 19]. Two notable examples are HTTP/1.x and the POSIX file-system API, both of which support only one synchronous request at a time (per TCP connection or per call). As Moore's Law fades [46], it is increasingly important to make these protocols and interfaces parallelism-friendly. For example, HTTP/2 [5] added support for sending multiple requests per connection. However, to the best of our knowledge, little progress has been made on the file-system API.

In this article, we similarly propose to batch multiple file-system operations. We focus particularly on the Network File System (NFS) and study how much performance can be improved by using a file-system API friendly to NFSv4 [36, 37]; this latest version of NFS supports *compound procedures* that pack multiple operations into a single RPC so that only one round trip is needed to process them. Unfortunately, although NFS compounds have been designed, standardized, and implemented in most NFS clients and servers, they are underused—mainly because of the limitations of the low-level POSIX file-system interface [9].

To explain the operations and premise of NFS4's compound procedures, we discuss them using several instructive figures. We start with Figure 1, which shows how reading a small file is limited by the POSIX API. This simple task involves four syscalls (`stat`, `open`, `read`, and `close`) that generate five compounds, each incurring a round trip to the server. Because compounds are initiated by low-level POSIX calls, each compound contains only one significant operation (in bold), with the rest being trivial operations such as PUTFH and GETFH. Compounds reduced the number of round trips slightly by combining the syscall operations (LOOKUP, OPEN, READ) with NFSv4 state-management operations (PUTFH, GETFH) and attribute retrieval (GETATTR), but the syscall operations themselves could not be combined due to the serialized nature of the POSIX API.

Ideally, a small file should be read using only one NFS compound (and one round trip), as shown in Figure 2. This would reduce the read latency by 80% (by removing four of the five round trips). We can even read multiple files using a single compound, as shown in Figure 3. All these examples use the standard (unmodified) NFSv4 protocol. An NFSv4 server maintains a state called *current filehandle* (CFH) for each NFSv4 client. The operation PUTROOTFH sets the CFH to the filehandle of the root directory. PUTFH and GETFH set and retrieve the CFH, respectively. LOOKUP and OPEN assume that the CFH is a directory, find or open the specified name inside, respectively, and change the CFH. GETATTR, READ, and CLOSE all operate on the file indicated by the CFH. SAVEFH and RESTOREFH operate on the *saved filehandle* (SFH), an NFSv4 state similar to the *current filehandle* (CFH). SAVEFH copies the CFH to the SFH; RESTOREFH restores the CFH from the SFH.

For compounds to reach their full potential, we need a file-system API that can convey high-level semantics and batch multiple operations. We designed and developed *vNFS*, an NFSv4 client that exposes a high-level vectorized API. vNFS complies with the NFSv4.1 standard, requiring no changes to NFS servers. Its API is easy to use and flexible enough to serve as a building block for new higher level functions. vNFS is implemented entirely in user space and thus easy to extend.

vNFS is especially efficient and convenient for applications that manipulate large amounts of metadata or do small I/Os. For example, vNFS lets `tar` read many small files using a single RPC instead of using multiple RPCs for each; it also lets `untar` set the attributes of many extracted files at once instead of making separate system calls for each attribute type (owner, time, etc.).

We implemented vNFS using the standard NFSv4.1 protocol and added two small protocol extensions to support file appending and copying. We ported GNU's Coreutils package (`ls`, `cp`, and `rm`), `bsdtar`, `nghttp2` (an HTTP/2 server), and Filebench [16, 41] to vNFS. In general, we found

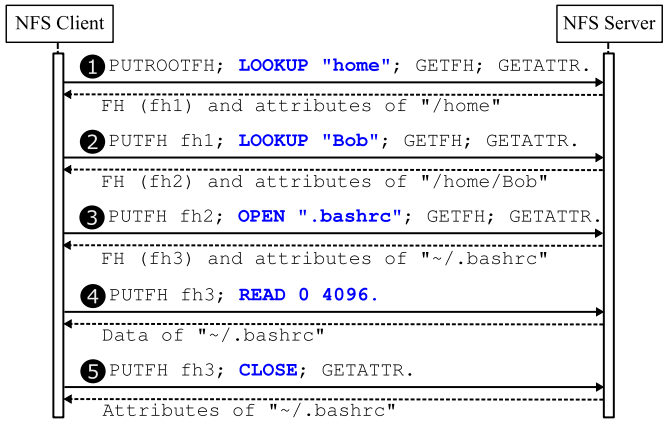


Fig. 1. NFS compounds used by the in-kernel NFS client to read a small file. Each numbered request is one compound with its operations separated by semicolons. Several operations manage the current filehandle (CFH) within the compound: PUTROOTFH sets the CFH to the filehandle of the root directory, while PUTFH and GETFH set or retrieve the CFH. LOOKUP and OPEN require the CFH to be a directory but then set the CFH to be that of the specified name. GETATTR, READ, and CLOSE all operate on the file indicated by the CFH.

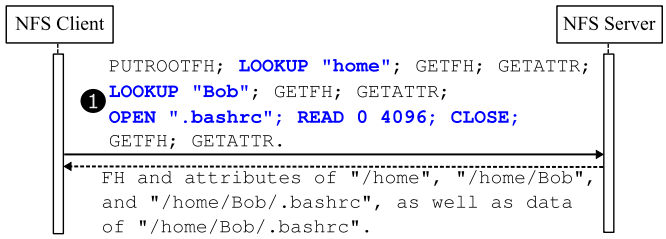


Fig. 2. Reading `/home/Bob/.bashrc` using only one compound. This single compound is functionally the same as the five in Figure 1, but uses only one network round trip.

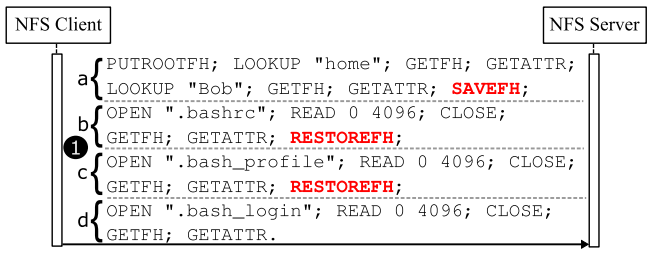


Fig. 3. One NFS compound that reads three files. The operations can be divided into four groups: (a) sets the current and saved filehandle to `/home/Bob`; (b), (c), and (d) read the files `.bashrc`, `.bash_profile`, and `.bash_login`, respectively. SAVEFH and RESTOREFH ensure that the CFH is `/home/Bob` when opening files. The reply is omitted.

it easy to modify applications to use vNFS. We ran a range of micro- and macro-benchmarks on networks with varying latencies, showing that vNFS can speed up such applications by 3–133× with small network latencies (≤5.2ms) and by up to 263× with a 30.2ms latency.

The rest of this article is organized as follows: Section 2 summarizes vNFS's design. Section 3 details the vectorized high-level API. Section 4 describes the implementation of our prototype.

Section 5 evaluates the performance and usability of vNFS by benchmarking applications we ported. Section 6 discusses related work and Section 7 concludes.

2 DESIGN OVERVIEW

In this section we summarize vNFS's design, including our goals, choices we made, and the architecture.

2.1 Design Goals

Our design has four goals, in order of importance:

- **High performance:** vNFS should considerably outperform existing NFS clients and improve both latency and throughput, especially for workloads that emphasize metadata and small I/Os. Performance for other workloads should be comparable.
- **Standards compliance:** vNFS should be fully compliant with the NFSv4.1 protocol so that it can be used with any compliant NFS server.
- **Easy adoption:** vNFS should provide a general API that is easy for programmers to use. It should be familiar to developers of POSIX-compliant code to enable smooth and incremental adoption.
- **Extensibility:** vNFS should make it easy to add functions to support new features and performance improvements. For example, it should be simple to add support for Server Side Copy (SSC; a feature in the current NFSv4.2 draft [20]) or create new application-specific high-level APIs.

2.2 Design Choices

The core idea of vNFS is to improve performance by using the compounding feature of standard NFS. We discuss the choices we faced and justify those we selected to meet the goals listed in Section 2.1.

2.2.1 Overt vs. Covert Coalescing. To leverage NFS compounds, vNFS uses a high-level API to overtly express the intention of compound operations. An alternative would be to covertly coalesce operations under the hood while still using the POSIX API. Covert coalescing is a common technique in storage and networking; for example, disk I/O schedulers combine many small requests into a few larger ones to minimize seeks [3]; and Nagle's TCP algorithm coalesces small outbound packets to amortize overhead for better network utilization [29].

Although overt compounding changes the API, we feel it is superior to covert coalescing in four important respects: (1) By using a high-level API, overt compounding can batch dependent operations, which are impossible to coalesce covertly. For example, using the POSIX API, we cannot issue a `read` until we receive the reply from the preceding `open`. (2) Overt compounding can use a new API to express high-level semantics that cannot be efficiently conveyed in low-level primitives. NFSv4.2's SSC is one such example [20]. (3) Overt compounding improves both throughput and latency, whereas covert coalescing improves throughput at the cost of latency since accumulating calls to batch together inherently requires waiting. Covert coalescing is thus detrimental to metadata operations and small I/Os that are limited by latency. This is important in modern systems with faster SSDs and 40GbE NICs, where latency has been improving much more slowly than raw network and storage bandwidth [35]. (4) Overt compounding allows implementations to use all possible information to maximize performance; covert coalescing depends on heuristics, such as timing and I/O sizes, that can be suboptimal or wrong. For example, Nagle's algorithm can interact badly with Delayed ACK [12].

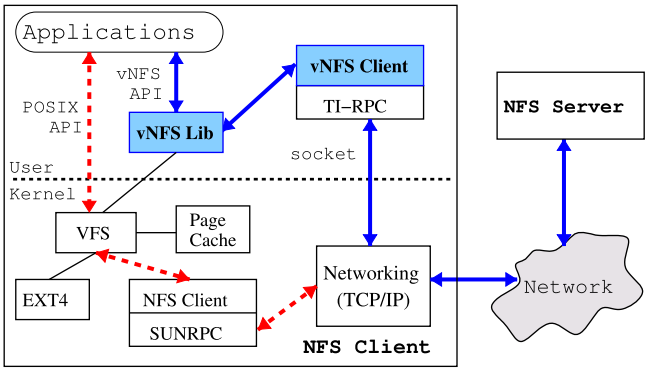


Fig. 4. vNFS Architecture. The thick arrows show vNFS's data path, and the dashed arrows show the in-kernel NFS client's data path. The vNFS library and client (shaded boxes) are new components we added; the rest already existed.

2.2.2 Vectorized vs. Start/End-Based API. Two types of APIs can express overt compounding: a vectorized one that compounds many desired low-level NFS operations into a single high-level call, or an API that uses calls like `start_compound` and `end_compound` to combine all low-level calls in between [34]. We chose the vectorized API for two reasons: (1) A vectorized API is easier to implement than a start/end-based one. Users of a start/end-based API might mix I/Os with other code (such as looping and testing of file-system states), which NFS compounds cannot support. (2) A vectorized API logically resides at a high level and is more convenient to use, whereas using a low-level start/end-based API is more tedious for high-level tasks (similar to C++ programming vs. assembly).

2.2.3 User-Space vs. In-Kernel Implementation. A kernel-space implementation of vNFS would allow it to take advantage of the kernel's page and metadata caches and use the existing NFS code base. However, we chose to design and implement vNFS in user space for two reasons: (1) Adding a user-space API is much easier than adding system calls to the kernel and simplifies future extensions; and (2) user-space development and debugging is faster and easier. Although an in-kernel implementation might be faster, prior work indicates that the performance impact can be minimal [40], and the results in this article demonstrate substantial performance improvements even with our user-space approach.

2.3 Architecture

Figure 4 shows the architecture of vNFS, which consists of a library and a client. Instead of using the POSIX API, applications call the high-level vectorized API provided by the vNFS library, which talks directly to the vNFS client. The vNFS library facilitates application adoption since most modern applications are developed using libraries and frameworks instead of OS system calls [2]. To provide generic support and encourage incremental adoption, the library detects when compound operations are unsupported and, in that case, converts vNFS operations into standard POSIX primitives. Thus, the vNFS library can also be used with file systems that do not support compounding, for example, as a utility library for batching file-system operations.

The vNFS client accepts vectorized operations from the library, puts as many of them into each compound as possible, sends them to the NFS server using Transport-Independent RPC (TI-RPC), and finally processes the reply. Note that existing NFSv4 servers already support compounds and can be used with vNFS without change. TI-RPC is a generic RPC library without the limitations

Table 1. vNFS Vectorized API Functions

Function	Description
`vopen` `vclose`	Open/close many files.
`vread` `vwrite`	Read/write/create/append files with automatic file opening and closing.
`vgetattrs` `vsetattrs`	Get/set multiple attributes of file-system objects.
`vsscopy` `vcopy`	Copy files in whole or in part with/without Server Side Copy.
`vmkdir`	Create directories.
`vlistdir`	List (recursively) objects and their attributes in directories.
`vsymlink`	Create many symbolic links.
`vreadlink`	Read many symbolic links.
`vhardlink`	Create many hard links.
`vremove`	Remove many objects.
`vrename`	Rename many objects.

Each Function has two return values: an error code and a count of successful operations. NFS servers stop processing the remaining operations in a compound once any operation inside failed. To facilitate gradual adoption, vNFS also provides POSIX-like scalar API functions, omitted here for Brevity. Each vNFS function has a version that does not follow symbolic links, also omitted.

of Linux's in-kernel SUNRPC (e.g., supporting only a single data buffer per call); TI-RPC can also run on top of TCP, UDP, and RDMA. Like the in-kernel NFS client, the vNFS client also manages NFSv4's client-side states such as sessions, and the like.

3 VNFS API

This section details vNFS's vectorized API (listed in Table 1). Each API function expands its POSIX counterparts to operate on a vector of file-system objects (e.g., files, directories, symbolic links). vNFS functions handle errors in a standard manner: return results for successful operations, report the index of the first failed operation in a compound (if any), and ignore any remaining operations that were not executed by the server. Figure 5 demonstrates the use of the vNFS API to read three small files in one NFS compound. To simplify programming, vNFS also provides utility functions for common tasks such as recursively removing an entire directory, and the like.

3.1 `Vread/Vwrite`

These functions can read or write multiple files using a single compound, with automatic on-demand file opening and closing. These calls simultaneously boost throughput, reduce latency, and simplify programming. Both functions accept a vector of I/O structures, each containing a `vfile` structure (Figure 5), offset, length, buffer, and flags. Our vectorized operations are more flexible than the `readv` and `writev` system calls and can operate at many (discontinuous) offsets of multiple files in one call. When generating compound requests, vNFS adds OPENs and CLOSEs for files represented by paths; files represented by descriptors do not need that since they are already open. OPENs and CLOSEs are coalesced when possible (e.g., when reading twice from one file).

```
// The "vfile" structure identifies a file to operate on.
struct vfile {
  enum VFILETYPE type;    // PATH or DESCRIPTOR
  union {
    const char *path;     // Where "type" is PATH
    int fd;               // or (vNFS file) DESCRIPTOR.
  };
};

// The "vio" I/O structure contains a vfile.
struct vio ios[3] = {
  { .vfile = { .type = PATH,
               .path = "/home/Bob/.bashrc" },
    .offset = 0,
    .length = 64 * 1024,
    .data = buf1,         // pre-allocated 64KB buffer
    .flags = 0,           // contains an output EOF bit
  },
  ...                     // two similar I/O structures omitted
};
struct vres r = vread(ios, 3); // read 3 files
```

Fig. 5. A simplified C code sample of reading three files at once using the vectorized API.

Since the files in the I/O structures can be the same, `vread`/`vwrite` can also be used to read/write at many (discontinuous) offsets in a single file.

The length field in the I/O structure also serves as an output, returning the number of bytes read or written. The structure has several flags that map to NFS's internal Boolean arguments and replies. For example, the flag `is_creation` corresponds to the NFS `OPEN4_CREATE` flag, telling `vwrite` to create the target file if necessary. `is_write_stable` corresponds to NFS's WRITE `DATA_SYNC4` flag, causing the server to save the data to stable storage, avoiding a separate NFS COMMIT. Thus, a single `vwrite` can achieve the effect of multiple writes and a following `fsync`, which is a common I/O pattern (e.g., in logging or journaling).

3.1.1 State Management. NFSv4 is stateful, and OPEN is a state-mutating operation. The NFSv4 protocol requires a client to open a file before reading or writing it. Moreover, READ and WRITE must provide the *stateid* (an ID uniquely identifying a server's state [37]) returned by the preceding OPEN. Thus, state management is a key challenge when `vread` or `vwrite` adds OPEN and READ/WRITE calls into a single compound. vNFS solves this by using the NFS *current stateid*, which is a server-side state similar to the current filehandle. To ensure that the NFS server always uses the correct state, `vread` and `vwrite` take advantage of NFSv4's special support for using the current stateid [37, Section 8.2.3].

3.1.2 Appending. `vwrite` also adds an optional small extension to the NFSv4 protocol to better support appends. As noted in the Linux manual page for `open(2)` [25], "O_APPEND may lead to corrupted files on NFS filesystems if more than one process appends data to a file at once." The base NFSv4 protocol does not support appending, so the kernel NFS client appends by writing to an offset equal to the current known file size. This behavior is inefficient because the file size must first be read in a separate RPC, and it is vulnerable to time-of-check-to-time-of-use (TOCTTOU) attacks. Our extension uses a special offset value (`UINT64_MAX`) in the I/O structure to indicate appending, making appending reliable with a tiny (5 LoC) change to the NFS server.

3.2 Vopen/Vclose

Using vread and vwrite, applications can access files without explicit opens and closes. Our API still supports vopen and vclose operations, which add efficiency for large files that are read or written many times. vopen and vclose are also important for maintaining NFS's close-to-open cache consistency [24]. vopen opens multiple files (specified by paths) in one RPC, including the LOOKUPs needed to locate their parent directories, as shown in Figure 3. Each file has its own open flags (read, write, create, etc.), which is useful when reading and writing are intermixed, as in external merge sorting. We also offer vopen_simple, which uses a common set of flags and mode (in case of creation) for all files. Once opened, a file is represented by a file descriptor, which is an integer index into an internal table that keeps states (file cursor, NFSv4 stateid and sequenceid [37], etc.) of open files. vclose closes multiple opened files and releases their resources.

3.3 Vgetattrs/Vsetattrs

These two functions manipulate several attributes of many files at a time, combining multiple system calls (chmod, chown, utimes, and truncate, etc.) into a single compound, which is especially useful for tools like tar and rsync. The aging POSIX API is the only restriction on setting many attributes at a time; the Linux kernel VFS already supports multi-attribute operations using the setattr method of inode_operations, and the NFSv4 protocol has similar SETATTRs support. vgetattrs/vsetattrs not only gets/sets many attributes at a time but also does that for many files, all in one RPC. vgetattrs and vsetattrs use an array of attribute structures as both inputs and outputs. Each structure contains a vfile structure, all attributes (mode, size, etc.), and a bitmap showing which attributes are in use.

3.4 Vsscopy/Vcopy

File copying is so common that Linux has added the sendfile and splice system calls to support it. Unfortunately, NFS does not yet support copying, and clients must use READs and WRITEs instead. This wastes time and bandwidth because data have to be read over the network and immediately written back. It is more efficient to ask the NFS server to copy the files directly on its side. This SSC operation has already been proposed for the upcoming NFSv4.2 [20]. Being forward-looking, we included vsscopy in vNFS to copy many files (in whole or in part) using SSC. However, since SSC requires server enhancements, we also provide vcopy for compatibility with current servers.

vsscopy accepts an array of copy structures, each containing the source file and offset, the destination file and offset, and the length. The destination files are created by vsscopy if necessary. The length can be UINT64_MAX, in which case the effective length is the distance between the source offset and the end of the source file. vsscopy can use a single RPC to copy many files in their entirety. The copy structures return the number of copied bytes in the length fields.

vcopy has the same effect but does not use SSC. vcopy is useful when the NFS server does not support SSC; vcopy can copy N small files using three RPCs (a compound for each of vgetattrs, vread, and vwrite) instead of $7{\times}N$ RPCs (2 OPENs, 2 CLOSEs, 1 GETATTR, 1 READ, and 1 WRITE for each file).

A future API could provide only vcopy and silently switch to vsscopy when SSC is available; we included a separate vsscopy in the current implementation so that we could easily compare it to vcopy.

3.5 Vmkdir

vNFS provides vmkdir to create multiple directories at a time (such as directory trees), which is common in tools such as untar, cmake, and recursive cp. vNFS has an ensure_directory utility

function that uses `vmkdir` to ensure that a deep directory and all its ancestors exist. Consider `''/a/b/c/d''` for example: The utility function first uses `vgetattrs` with arguments `[''/a''; ''/a/b''; ...]` to find out which ancestors exist and then creates the missing directories using `vmkdir`. Note that simply calling `vmkdir` with vector arguments `[''/a''; ''/a/b''; ...]` does not work: The NFS server will fail (with `EEXIST`) when trying to recreate the first existing ancestor and stop processing all remaining operations.

3.6 `Vlistdir`

This function speeds up directory listing with four improvements to `readdir`: (1) `vlistdir` lists multiple directories at a time; (2) an `opendir` call is not necessary prior to beginning listing the directory; (3) `vlistdir` retrieves attributes along with directory entries, saving subsequent `stats`; and (4) `vlistdir` can work recursively. The operation can be viewed as a fast vectorized `ftw(3)` that reads NFS directory contents using as few RPCs as possible.

`vlistdir` takes five arguments: an array of directories to list, a bitmap indicating desired attributes, a flag to select recursive listing, a user-defined callback function (similar to `ftw`'s second argument [18]), and a user-provided opaque pointer that is passed to the callback. For each listed directory entry, the callback function is invoked with three arguments: an attribute structure that contains the file path and the active file attributes (see Section 3.3), the top-level ancestor directory as it appeared in `vlistdir`'s first argument, and the opaque pointer (the last argument of `vlistdir`). `vlistdir` processes directories in the order given; recursion is breadth-first. However, directories at the same level in the tree are listed in an arbitrary order.

3.7 `Vsymlink/Vreadlink/Vhardlink`

These three vNFS operations allow many links to be created or read at a time. Symbolic links are often created in bulk to provide a shadow-tree view of another directory tree; together with `vlistdir`, `vsymlink` can optimize operations like `cp -sr` and `lndir`. `vhardlink` is similar to `vsymlink` but creates hard links. All three functions accept a vector of paths and a vector of buffers containing the target paths.

3.8 `Vremove`

`vremove` removes multiple files and directories at a time. Although `vremove` does not support recursive removal by itself, a program can achieve this effect with a recursive `vlistdir` followed by properly ordered `vremoves`; vNFS provides a utility function `rm_recursive` for this purpose.

3.9 `Vrename`

Renaming many files and directories is common, as when organizing media collections. Many free tools [7, 22, 33] and even a commercial one [17] have been developed just for this purpose. vNFS provides `vrename` to facilitate and speed up bulk renaming. `vrename` renames a vector of source paths to a vector of destination paths using as few RPCs as possible.

4 IMPLEMENTATION

We implemented a prototype of vNFS in C/C++ on Linux. As shown in Figure 4, vNFS has a library and a client, both running in user space. The vNFS library implements the vNFS API. Applications use the library by including the API header file and linking to it. For NFS files, the library redirects API function calls to the vNFS client, which builds large compound requests and sends them to the server via the TI-RPC library. For non-NFS files, the library translates the API functions into POSIX calls and therefore can also be used as a utility library. (Our current prototype considers a file to be on NFS if it is under any exported directory specified in vNFS's configuration file.) The vNFS client

builds on NFS-Ganesha [13, 30], an open-source user-space NFS server. NFS-Ganesha can export files stored in many backends, such as XFS and GPFS. Our vNFS prototype uses an NFS-Ganesha backend called PROXY, which exports files from *another* NFS server and can be repurposed as a user-space NFS client. The original PROXY backend used NFSv4.0; we added NFSv4.1 support including session management [37]. Our prototype implementation added 10,632 lines of C/C++ code and deleted 1,407. vNFS is thread-safe; we have tested it thoroughly.

4.1 RPC Size Limit

The vNFS API functions (shown in Table 1) do not impose a limit on the number of operations per call. However, each RPC has a configurable memory size limit, defaulting to 1MB. We ensure that vNFS does not generate RPC requests larger than that limit no matter how many operations an API call contains. Therefore, we split long arguments into chunks and send one compound request for each chunk. We also merge RPC replies upon return to hide any splitting.

Our splitting avoids generating small compounds. For data operations (`vread` and `vwrite`), we can easily estimate the sizes of requests and replies based on buffer lengths, so we split a compound only when its size becomes close to 1MB. (The in-kernel NFS client similarly splits large READs and WRITEs according to the `rsize` and `wsize` mount options, which also default to 1MB.) For metadata operations, it is more difficult to estimate the reply sizes, especially for READDIR and GETATTR. We chose to be conservative and simply split a compound of metadata operations whenever it contains more than k NFS operations. We chose a default of 256 for k, which enables efficient concurrent processing by the NFS server and yet is unlikely to exceed the size limit. For example, when listing the Linux source tree, the average reply size of READDIR—the largest metadata operation—is around 3,800 bytes. If k is still too large (e.g., when listing large directories), the server will return partial results and use cookies to indicate where to resume the call for follow-up requests.

4.2 Protocol Extensions

vNFS contains two extensions to the NFSv4.1 protocol to support file appending (see Section 3.1) and SSC (see Section 3.4). Both extensions require changes to the protocol and the NFS server. We have implemented these changes in our server, which is based on NFS-Ganesha [13, 30]. The file-appending extension was easy to implement, adding only an `if` statement with five lines of C code. In the server, we simply use the file size as the effective offset whenever the write offset is `UINT64_MAX`.

Our implementation of SSC follows the design proposed in the NFSv4.2 draft [20]. We added the new COPY operation to our vNFS client and the NFS-Ganesha server. On the server side, we copy data using `splice(2)`, which avoids unnecessarily moving data across the kernel/user boundary. This extension added 944 lines of C code to the NFS-Ganesha server.

4.3 Path Compression

We created an optimization that reduces the number of LOOKUPs when a compound's file paths have locality. The idea is to shorten paths that have redundancy by making them relative to preceding ones in the same compound. For example, when listing the directories ``''/1/2/3/4/5/6/7/a''`` and ``''/1/2/3/4/5/6/7/b''``, a naïve implementation would generate eight LOOKUPs per directory (one per component). In such cases, we replace the path of the second directory with ``''../b''`` and use only one LOOKUPP and one LOOKUP; LOOKUPP sets the current filehandle to its parent directory. This simple technique saves as many as six NFS operations for this example.

Note that LOOKUPP produces an error if the current filehandle is not a directory because most file systems have metadata recording parents of directories but not parents of files. In that case, we use SAVEFH to remember the deepest common ancestor in the file-system tree (i.e.,

‘‘/1/2/3/4/5/6/7’’ in the preceding example) of two adjacent files and then generate a RESTOREFH and LOOKUPS. (However, this approach cannot be used for LINK, RENAME, and COPY, which already use the saved filehandle for other purposes.) We use this optimization only when it saves NFS operations: for example, using ‘‘../../c/d’’ does not save anything for paths ‘‘/1/a/b’’ and ‘‘/1/c/d’’.

4.4 Client-Side Caching

vNFS provides write-through client-side caches for data and metadata. Both caches use an LRU replacement algorithm and set a configurable expiration time for each cache entry. vNFS's caches follow the traditional close-to-open [24] consistency model of NFS. Our cache implementation is based on the Poco Cache Framework [32].

The vNFS metadata cache uses full paths as cache keys instead of individual path components so that searching for a file involves only one lookup. As shown in a recent study by Tsai et al. [42], full-path–based caches are faster than component-based caches in most scenarios. Each vNFS metadata cache entry includes the attributes, path, and NFS filehandle of the file-system object. The cache entry for each directory has additional data to speed up any frequently used directory-listing operations. These data include pointers to the cache entries for the directory's children and a flag indicating whether all of its children are cached. When this flag is true, the directory's content can be listed without sending any RPCs to the NFS server at all.

The vNFS data cache organizes cached file data as blocks of a configurable size. Unlike Linux's page cache, vNFS's data cache is write-through; this simplifies the cache because it does not contain any dirty data. A drawback of being write-through is that each write, no matter how small, has to be sent to the server immediately. However, this is not a significant problem for vNFS because many small writes can still be batched using its vectorized API.

A special strategy of vNFS's data cache is that it does not cache file data smaller than a configurable threshold (which defaults to 256KB). This is because caching small files does not justify the latency overhead caused by cache revalidation. NFS's close-to-open consistency requires that NFS clients revalidate locally cached file data for each file after it is opened. This revalidation executes an extra GETATTR request and then compares the returned `ctime` from the server with the locally cached `ctime`. If the time is the same, this means that no other NFS client has changed the file and the locally cached file data are valid; otherwise, the cached data must be discarded. For small files whose entire contents fit in one request, it is more efficient to reread the file than to perform this revalidation. This strategy is especially useful in modern networks with high bandwidth but not necessarily low latency. As shown in an earlier version of this article [10], reading files smaller than 512KB is faster without a cache. This strategy is an optional feature that can be turned off by setting the caching threshold to zero.

5 EVALUATION

To evaluate vNFS, we ran micro-benchmarks and also ported applications to use it. We now discuss our porting experience and evaluate the resulting performance.

5.1 Experimental Testbed Setup

Our testbed consists of two identical Dell PowerEdge R710 machines running CentOS 7.0 with a Linux 3.10 kernel. Each machine has a six-core Intel Xeon X5650 CPU, 64GB of RAM, an Intel 10GbE NIC, and a PERC 6/i RAID controller with battery-backed write cache. One machine acts as the NFS server and runs NFS-Ganesha 2.4 with our file-appending and SSC extensions; the other machine acts as a client and runs vNFS. The NFS server exports an Ext4 file system, which is stored on an Intel DC S3700 200GB SSD, to the client. The two machines are directly connected

to a Dell PowerConnect 8024F 10GbE switch. We measured an average RTT of 0.2ms between the two machines. To emulate different LAN and WAN conditions, we used `netem` to inject delays of 1–30ms into the server's outbound link.

To evaluate vNFS's performance, we compared it with the in-kernel NFSv4.1 client—called the *baseline*—that mounts the exported directory using the default options: the attribute cache (`ac` option) is enabled and the maximum read/write size (`rsize`/`wsize` options) is 1MB. Our vNFS prototype does not use `mount` but instead reads the name of the exported directory from a configuration file.

We ran each experiment at least three times and averaged the results. Our graphs show the standard deviation as error bars, which are so small as to be invisible in most figures.

Before each run, we flushed the page and dentry caches of the in-kernel client by unmounting and remounting the NFS directory. We also emptied the vNFS client-side cache before each run (see Section 4.4). The metadata caches of both the in-kernel client and the vNFS client have timeout periods of 1 minute; the sizes of both data caches are bounded by the client's RAM size. The NFS-Ganesha server uses an internal cache in addition to the OS's page and dentry caches.

To quantify the effort of porting applications, we report the LoC change for each application including the error-handling code.

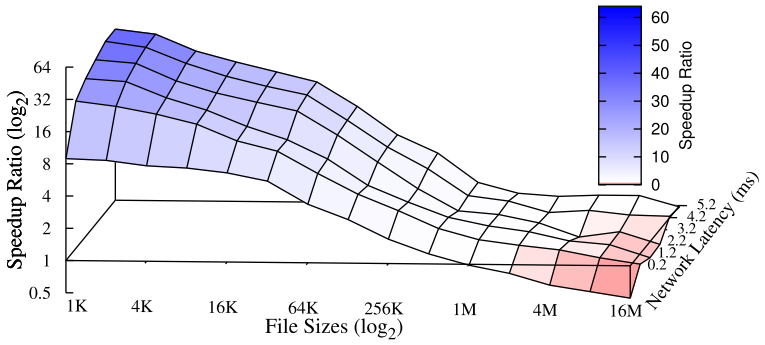
5.2 Micro-Workloads

5.2.1 Small vs. Big Files. vNFS's goal is to improve performance for workloads with many small NFS operations while staying competitive for data-intensive workloads. To test this, we compared the time used by vNFS and the baseline to read and write 1,000 equally sized files in their entirety while varying the file size from 1KB to 16MB. We repeated the experiment in networks with 0.2ms to 5.2ms latencies and packed as many operations as possible into each vNFS compound. The results are shown (in logarithmic scale) in Figure 6, where *speedup ratio* is the ratio of the baseline's completion time to vNFS's completion time. Speedup ratios of greater than one indicate that vNFS performed better than the baseline; ratios of less than one mean it was worse.

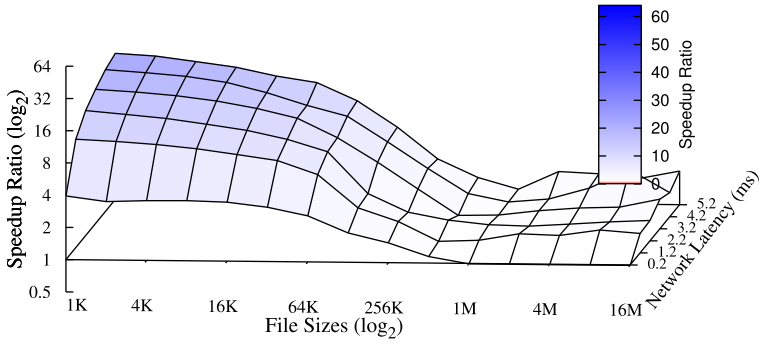
Because vNFS combined many small read and write operations into large compounds, it performed much better than the baseline when the file size was small. With a 1KB file size and 0.2ms network latency, vNFS is 9× faster than the baseline when reading (Figure 6(a)) and 4× faster when writing (Figure 6(b)). As the network latency increased to 5.2ms, vNFS's speedup ratio improved further to 40× for reading and 23× for writing. vNFS's speedup ratio was higher for reading than for writing because once vNFS was able to eliminate most network round trips, the NFS server's own storage became the next dominant bottleneck.

As the file size (the X-axis in Figure 6) was increased to 1MB and beyond, vNFS's compounding effect faded, and the performance of vNFS became close to or slower than the baseline (by up to 50%). In fast networks with 0.2–3.2ms delay, vNFS reads of large files were slower than the baseline because vNFS's client-side data cache was in user space and performed extra data copies compared to the page cache of the in-kernel baseline. Data caches were not helpful in this workload because each file was read only once. Without a data cache, vNFS reads of large files were comparable to the baseline, as shown in an earlier version of this article [10].

However, vNFS was slightly faster than the baseline with 4.2–5.2ms network latencies; this is because, although data operations were too large to be combined, vNFS could still combine them with small meta-data operations such as OPEN, CLOSE, and GETATTR. For large writes, vNFS's performance is the same as or better than the baseline depending on the network delay. The negative effects of vNFS's caching were negligible for writing because the time needed to write to the server's storage dwarfed the cost of copying data into the vNFS client's in-memory cache.



(a) Reading whole files



(b) Writing whole files

Fig. 6. vNFS's speedup ratio (the vertical Z-axis, in logarithmic scale) relative to the baseline when reading and writing 1,000 equally sized files whose sizes (the X-axis) varied from 1KB to 16MB. vNFS is faster than (blue), equal to (white), or slower than (red) the baseline when the speedup ratio is larger than, equal to, or smaller than 1.0, respectively. The network latency (Y-axis) starts from 0.2ms (instead of zero) because that is the measured base latency of our testbed (see Section 5.1).

5.2.2 Compounding Degree. The degree of compounding (i.e., the number of nontrivial NFS operations per compound) is a key factor in determining how much vNFS can boost performance. The ideal situation is to perform a large number of file-system operations in one compound, but doing so is not always possible because applications may have critical paths that depend on only a single file. To study how the degree of compounding affects vNFS's performance, we compared vNFS with the baseline when calling the vNFS API functions with different numbers of operations in their vector arguments.

Figure 7 shows the speedup ratio of vNFS relative to the baseline as the number of operations per API call was increased from 1 to 256 in a 0.2ms-latency network. With one operation per call, vNFS outperformed the baseline for 10 out of the 14 operation types. The improvement is because vNFS could still save round trips for single-file calls. For example, the baseline used three RPCs to rename a file: one LOOKUP for the source directory, another LOOKUP for the destination directory, and one RENAME. However, vNFS used just one compound RPC combining all three operations. Therefore, vNFS is more than 2× faster than the baseline for `Remove`.

The four exceptions where vNFS performed worse than the baseline were `OpenClose`, `Getattrs`, `Mkdir`, and `Write4KSync`; they were slower by 8%, 24%, 15%, and 25%, respectively. The slowdown

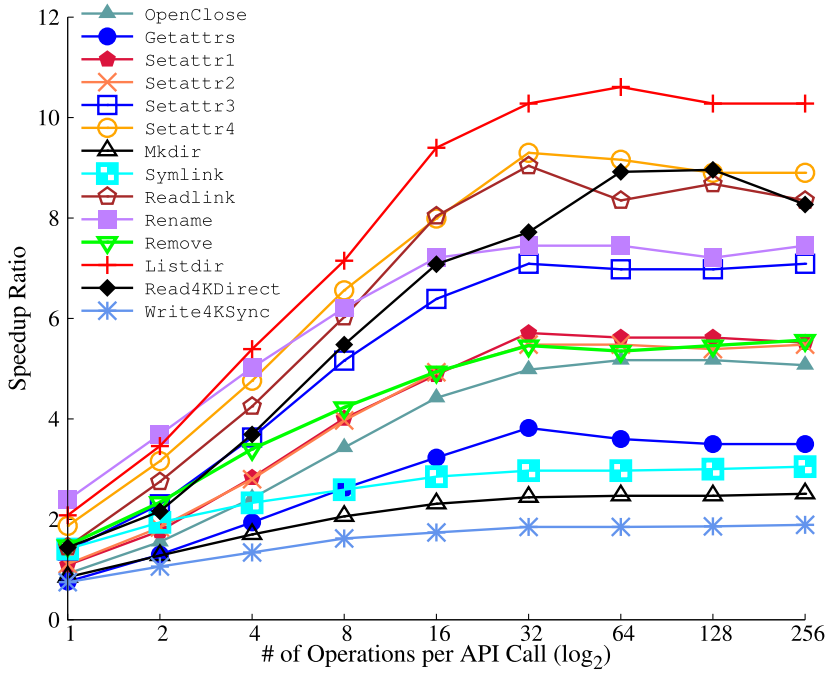


Fig. 7. vNFS's speedup ratio relative to the baseline under different degrees of compounding. The X-axis is $\log_2$. The network latency is 0.2ms. `OpenClose` opens and closes files without doing anything else; `Getattrs` is vectorized `stat(2)`; `Setattr`N sets N files' attributes (mixes of mode, owner, timestamp, and size); `Listdir` lists directories with 17 children (17 is the average number of children in the Linux source tree directories); `Read4KDirect` reads 4KB data from files opened with `O_DIRECT`; `Write4KSync` writes 4KB data to files opened with `O_SYNC`. The vector size of the baseline is actually the number of individual POSIX calls issued iteratively.

was because each of these tests performed only a single NFS operation at a time, so vNFS was unable to combine anything. However, vNFS still incurred the overhead of performing RPCs in user space.

When there was more than one operation per API call, compounding became beneficial and vNFS significantly outperformed the baseline for all operations. All calls except `Write4KSync` (bottlenecked by the server's storage stack) were more than 2× faster than the baseline when multiple operations were compounded. Note that `vsetattrs` can set multiple attributes at once, whereas the baseline sets one attribute at a time. We observe in Figure 7 that the speedup ratio of setting more attributes at once (e.g., `Setattr4`) was always higher than that of setting fewer (e.g., `Setattr3`).

In our experiments with slower networks, vNFS's speedups relative to the baseline were even higher than in the 0.2ms latency network—up to 59× faster—due to vNFS's massive reduction of high-latency RPCs. In a 5.2ms-latency network (shown in Figure 8), most relative trends between the benchmarks remained the same—for example, `Settattr4` was faster than `Setattr3`, etc. However, `Write4KSync` plateaued after 32 operations per call because it was bottlenecked by the server's storage stack. `Listdir` remained the fastest because it operates purely on metadata and saves the most RPCs relative to the baseline.

5.2.3 Caching. Both our vNFS client and the in-kernel NFS client (the baseline) have a cache. To study the performance improvement from caching, we compared vNFS and the baseline when

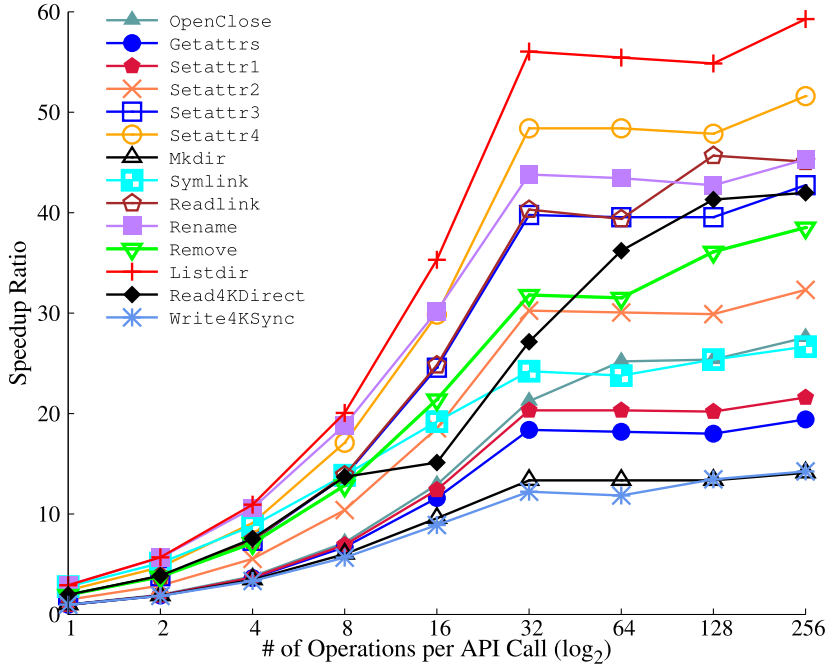


Fig. 8. This figure is the same as Figure 7 but with a network latency of 5.2ms.

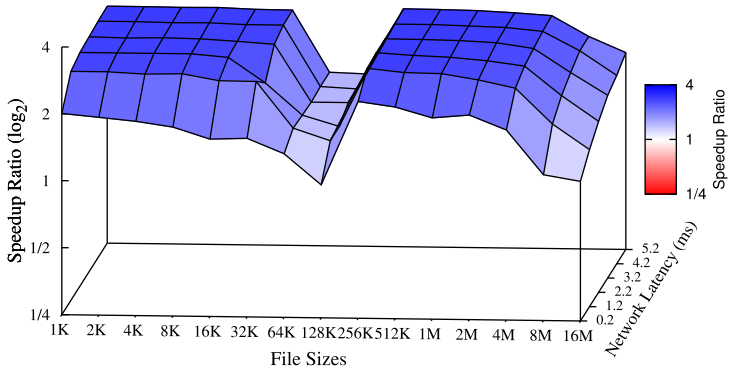


Fig. 9. The speedup ratio of vNFS over the baseline (in logarithmic scale) when repeatedly opening, reading, and closing a single file whose size is shown on the X axis. vNFS is faster than (blue), equal to (white), or slower than (red) the baseline when the speedup ratio is larger than, equal to, or smaller than 1, respectively. vNFS only caches files larger than 256KB. The vertical Z axis is in logarithmic scale; the higher the better.

repeatedly opening, reading, and closing a single file whose size varied from 1KB to 16MB. Figure 9 shows the results, where a speedup ratio of greater than 1 means that vNFS outperformed the baseline and a speedup ratio of less than 1 means that vNFS did worse.

This experiment ran shorter than the 60-second timeout period of the client-side metadata cache, so all but the first file lookups were cached for both vNFS and the baseline. The baseline served all reads from its cache except the first read. Even though vNFS only caches files of size 256KB and greater, the baseline was slower for small files (1–8KB). This is because each read requires three

RPCs to maintain close-to-open semantics: an OPEN, a GETATTR (for cache revalidation), and a CLOSE. In comparison, vNFS uses only one compound RPC, combining the OPEN, READ (uncached), and CLOSE. The savings from compounding more than compensated for vNFS not caching small files. As discussed in Section 4.4, by not caching small files, vNFS also avoids the extra network round trips required for cache revalidation. Although vNFS does not cache small files on the client side, the NFS server still uses the OS page cache to avoid repeatedly reading files from storage.

As the file size increased beyond 8KB, vNFS remained faster than the baseline until the file size reached 128KB. For 128KB files, the benefits of client-side caching became significant, making the baseline's performance close to that of vNFS (which did not cache those files). Once the file size reached and exceeded 256KB, vNFS became approximately 4× faster than the baseline as it started reading from its cache. However, as the file size grew larger than 4MB, the speedup ratio began dropping for two reasons: (1) vNFS's savings in metadata operations became less important as the workload became data-intensive, and (2) vNFS's user-space data cache performs extra data copying and was thus less efficient than the baseline's in-kernel page cache. Figure 9 shows the general trend that vNFS has higher speedup ratios in high-latency networks than in low-latency ones since vNFS's performance improvements come from reducing round trips.

5.3 Macro Workloads

To evaluate vNFS using realistic applications, we modified `cp`, `ls`, and `rm` from GNU Coreutils, Filebench [16, 41], and nghttp2 [31] to use the vNFS API; we also implemented an equivalent of GNU `tar` using vNFS.

5.3.1 GNU Coreutils. Porting `cp` and `rm` to vNFS was easy. For `cp`, we added 170 lines of code and deleted 16; for `rm`, we added 21 and deleted 1. Copying files can be trivially achieved using `vsscopy`, `vgetattrs`, and `vsetattrs`. Recursively copying directories requires calling `vlistdir` on the directories and then invoking `vsscopy` for plain files, `vmkdir` for directories, and `vsymlink` for symbolic links—all of which is done in `vlistdir`'s callback function. We tested our modified `cp` with `diff -qr` to ensure that the copied files and directories were exactly the same as the source. Removing files and directories recursively in `rm` required similar changes except that we used `vremove` instead of `vsscopy`.

Porting `ls` was more complex because batching is difficult when listing directories recursively in a particular order (e.g., by file size). We could not use the recursive mode of `vlistdir` because the NFS READDIR operation does not follow any specific order when reading directory entries, and the whole directory tree might be too large to fit in memory. Instead, vNFS maintains a list of all directories to read in the proper order as specified by the `ls` options and repeatedly calls `vlistdir` (not recursively) on directories at the head of the list until it is empty. Note that (1) a directory is removed from the list only after all its children have been read, and (2) subdirectories must be sorted and then inserted immediately after their parent to maintain the proper order in the list. We added 392 lines of code and deleted 203 to port `ls` to vNFS. We verified that our port is correct by comparing the outputs of our `ls` with the vanilla version.

We used the ported Coreutils programs to copy, list, and remove an entire Linux-4.11.3 source tree, which contains 57,965 files with an average size of 11.6KB, 3,947 directories with an average of 17 children each, and 30 symbolic links. The large number of files and directories thoroughly exercises vNFS and demonstrates the performance benefits of compounding.

Figure 10 shows the results of copying the entire Linux source tree; vNFS outperformed the baseline in all cases. vNFS uses either `vsscopy` or `vcopy` depending on whether SSC is enabled. However, the baseline cannot use SSC because it is not yet supported by the in-kernel NFS client. Comparing vNFS to the baseline when copying the Linux source tree, vNFS used only

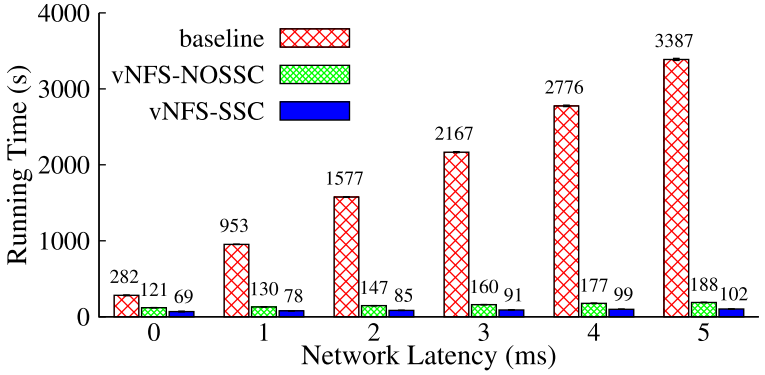


Fig. 10. Running time to copy (`cp -r`) the entire Linux source tree. The lower the better. vNFS runs much faster than the baseline both with and without Server Side Copy (SSC).

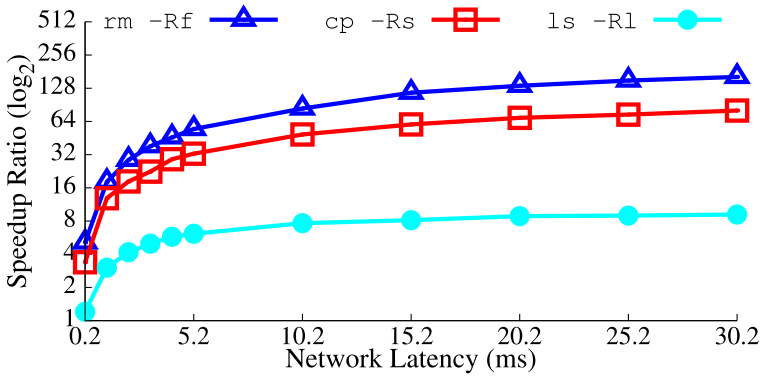


Fig. 11. vNFS's speedup relative to the baseline when symbolically copying (`cp -Rs`), listing (`ls -Rl`), and removing (`rm -Rf`) the Linux source tree. The Y-axis is logarithmic.

6,400 (compounded) RPCs whereas the baseline used as many as 565,200: Each of the 57,965 files generated two OPENs, two CLOSEs, one READ, one WRITE, and one SETATTR; there were also 68,000 ACCESSes, 69,700 GETATTRs, and 22,000 other operations such as READDIR and CREATE. Compared to the baseline, vNFS-NOSSC saved around 99% of the RPCs, with each vNFS compounding RPC containing an average of 250 operations. Therefore, even with only a 0.2ms network latency, vNFS-NOSSC was more than 2× faster than the baseline. The speedup ratio increased to 18× with a 5.2ms network latency.

When SSC was enabled, vNFS ran even faster, and vNFS-SSC reduced the running time of vNFS-NOSSC by another 43%. The further speedup of SSC was only moderate because most of the files in the test are small and our network bandwidth (10GbE) is large. The speedup ratio of vNFS-SSC compared to the baseline is 4–33× in networks with 0.2–5.2ms latencies. Even when the in-kernel client adds SSC support in the future, vNFS will continue to outperform it because this workload's bottleneck is the large number of small metadata operations, not the cost of data copying.

With the `-Rs` options, `cp` copies an entire directory tree by creating symbolic links to the source directory. Figure 11 shows speedups for symlinking, recursively listing (`ls -Rl`), and removing (`rm -Rf`) the Linux source tree. When recursively listing the Linux tree, `ls`-baseline used 12,487 RPCs including 4,247 READDIRs, 4,116 ACCESSes, and 4,116 GETATTRs. Note that the

in-kernel NFS client did not issue a separate GETATTR for each directory entry even though the vanilla `ls` program called `stat` for each entry listed. This efficiency is because the in-kernel NFS client pre-fetches the attributes using `readdirs` and serves the `stat` calls from the local dentry metadata cache. This optimization enables `ls`-baseline to finish the benchmark in just 6.6 seconds in the 0.2ms-latency network.

However, with our vectorized API, `ls`-vNFS did even better and finished in 5.5 seconds, using only 923 RPCs. Moreover, vNFS scales much better than the baseline. When the latency increased from 0.2 to 30.2ms, vNFS's running time rose to only 41 seconds, whereas the baseline increased to 376 seconds. `ls`-vNFS is 6–10× faster than `ls`-baseline in high-latency networks with 5.2–30.2ms latencies.

For symbolic copying and removing (Figure 11), vNFS was 3× and 5× faster than the baseline in the 0.2ms-latency network, respectively. This is because the baseline always operated on one file at a time, whereas vNFS could copy or remove more than 200 files at once. Compared to the baseline, vNFS improved `cp` by 33× and `rm` by 55× in the 5.2ms-latency network; with 30.2ms latency, the speedup ratios became 80× for `cp` and 162× for `rm`. For both removing and symbolic copying, vNFS ran faster in the 30.2ms-latency network (27 and 42 seconds, respectively) than the baseline did with 0.2ms latency (60 and 44 seconds, respectively), showing that compounds can indeed help NFSv4 realize its design goal of being WAN-friendly [27].

5.3.2 Tar. Because the I/O code in GNU `tar` is closely coupled to other parts of the program, we implemented a vNFS equivalent using `libarchive`, in which the I/O code is clearly separated. The `libarchive` library supports many archiving and compression algorithms; it is also used by FreeBSD `bsdtar`. Our implementation needed only 242 lines of C++ code for `tar` and 207 for `untar`, both including error-handling code.

When archiving a directory, we use the `vlistdir` API to traverse the tree and add subdirectories into the archive. We gather the listed files and symlinks into arrays, read them using `vread` and `vreadlink`, and finally compress and write the contents into the archive. During extraction, we read the archive in 1MB (RPC size limit) chunks and use `libarchive` to extract and decompress objects and their contents, which are then passed in batches to `vmkdir`, `vwrite`, or `vsymlink`. We ensured that our implementation is correct by feeding our `tar`'s output into our `untar` and comparing the extracted files with the original input files. We also tested for cross-compatibility with other tar implementations including `bsdtar` and GNU `tar`.

We used our `tar` and `untar` to archive and then extract the Linux 4.11.3 source tree. Archiving read 57,965 small files and wrote a large archive: 715MB uncompressed and 92MB with the `xz` option (default compression used by `kernel.org`); extracting reversed the process. There were also metadata operations on 30 symbolic links and 3,947 directories. Figure 12 shows the `tar`/`untar` results compared to `bsdtar` (running on the in-kernel NFS client) as the baseline. For `tar-nocompress` in the 0.2ms-latency network, vNFS was more than 5× faster than the baseline because the latter used 550,379 RPCs whereas vNFS used only 3,169 due to compounding. This enormous reduction made vNFS 27× faster when the network latency increased to 5.2ms. In terms of running time, vNFS took 107 seconds to archive the entire Linux source tree in the 5.2ms-latency network, whereas the baseline, even in the faster 0.2ms-latency network, took as long as 199 seconds. For `untar-nodecompress`, vNFS was also 5–38× faster, depending on the network latency.

Figure 12 also includes the results when `xz` compression was enabled. Compression reduced the size of the archive file by 87% (from 715MB down to 92MB) and thus saved 87% of the I/Os to the archive file. But the archiving time with compression was merely 10% shorter than that without compression for both the baseline and vNFS tar. The limited improvement is because the cost of writing the archive was small compared to that of reading the 57,965 source files. vNFS's

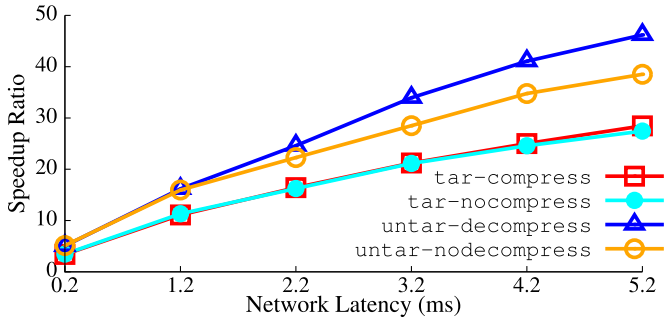


Fig. 12. Speedup ratios of vNFS relative to the baseline (`bsdtar`) when archiving and extracting the Linux-4.6.3 source tree, with and without xz compression.

speedup ratios compared to the baseline were about the same (differing by less than 4%) regardless of whether compression was on or off.

For untar, enabling compression affected vNFS and the baseline in different ways. As shown in Figure 12, compression caused the speedup ratios for `untar-decompress` (the curve with triangles) and `untar-nodecompress` (the curve with empty circles) to diverge. For example, at a latency of 5.2ms, enabling compression changed the baseline's performance by less than 0.3% but improved vNFS's performance by 16%. This difference can be attributed to vNFS's cache, which copied 92MB data into the cache when unarchiving the compressed archive but 715MB when extracting from the uncompressed one. vNFS's user-space cache pays a greater performance penalty compared to the in-kernel page cache because it performs extra data copies, which results in poor performance when reading large files such as the uncompressed archive, as shown on the right side of Figure 6(a). The data cache helps with repeated reads of cached files, but this workload reads the archive file only once.

This test shows that workloads with mixed I/O sizes are slow if there are many small I/Os, each incurring a network round trip; vNFS can significantly improve such workloads by compounding those small I/Os.

5.3.3 Filebench. We have ported Filebench to vNFS and added vectorized *flowops* to the Filebench workload modeling language (WML) [47]. We added 759 lines of C code to Filebench and removed 37. We converted Filebench's File-Server, NFS-Server, and Varmail workloads to equivalent versions using the new flowops. For example, we replaced N adjacent sets of `openfile`, `readwholefile`, and `closefile` (i.e., $3 \times N$ old flowops) with a single `vreadfile` (one new flowop), which internally uses our `vread` API to open, read, and close N files in one call.

The Filebench NFS-Server workload emulates the SPEC SFS benchmark [38]. It contains one thread performing four sets of operations: (1) open, entirely read, and close three files; (2) read a file, create a file, and delete a file; (3) append to an existing file; and (4) read a file's attributes. The File-Server workload emulates 50 users accessing their home directories and spawns one thread per user to perform operations similar to the NFS-Server workload. The Varmail workload mimics a UNIX-style email server operating on a `/var/mail` directory, saving each message as a file; it has 16 threads, each performing create-append-sync, read-append-sync, read, and delete operations on 10,000 16KB files.

Figure 13 shows the results of the Filebench workloads, comparing vNFS to the baseline. For the NFS-Server workload, vNFS was 3.3× faster than the baseline in the 0.2ms-latency network because vNFS combined multiple small operations into a single compound. With `vread`, vNFS can

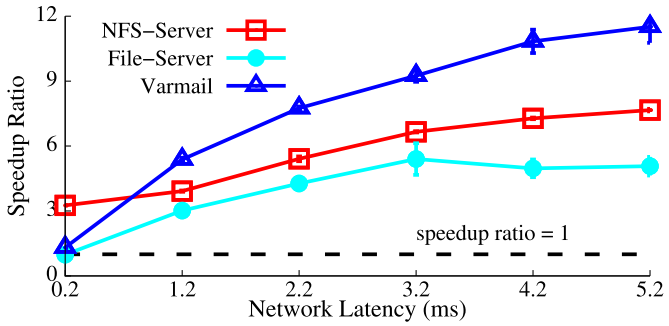


Fig. 13. vNFS's speedup ratios for Filebench workloads.

open, entirely read, and close three small files using a single RPC, whereas the baseline needs nine RPCs for the same workload. With `vwrite`, vNFS was also more efficient (and more reliable) when appending files since it does not need a separate GETATTR to read the file size (see Section 3.1). Figure 13 also shows that vNFS's speedup ratio was even higher in slower networks. For example, with a 5ms network latency, vNFS was 7.7× faster than the baseline. The 7.7× speedup ratio is higher than the 5× ratio in our previous study [10]; this is because vNFS's client-side cache enables large files to be read without contacting the server (small files are not cached), whereas the previous study did not have a cache.

As shown in Figure 13, the File-Server workload is around 2% slower than the baseline in the 0.2ms-latency network. This is because the File-Server workload has as many as 50 threads and generates a heavy I/O load to the NFS server's storage stack, which became the bottleneck. As the network latency increased, the load on the NFS server became lighter and vNFS became faster thanks to saving round trips. After the network latency increased to 3.2ms, this workload's speedup ratio plateaued at 5× because that is the compounding degree of this workload. Note that vNFS's cache did not help much here because this workload has an average file size of 128KB, which is lower than vNFS's 256KB caching threshold (see Section 4.4).

Because the Varmail workload is also multi-threaded, its speedup ratio curve started low in Figure 13, just as in the File-Server workload. However, vNFS's speedup ratio in the Varmail workload was higher than in the File-Server workload because the Varmail workload has a higher compounding degree.

5.3.4 HTTP/2 Server. Similar to the concept of NFSv4 compounds, HTTP/2 improves on HTTP/1.*x* by transferring multiple objects in one TCP connection. HTTP/2 also added a PUSH feature that allows an HTTP/2 server to proactively push related Web objects to clients [5, Section 8.2]. For example, upon receiving an HTTP/2 request for `index.html`, the server can proactively send the client other Web objects (such as Javascript, CSS, and image files) embedded inside that `index.html` file instead of waiting for it to request them later. PUSH can reduce a Web site's loading time for end users. It also allows Web servers to read many related files together, enabling efficient processing by vNFS.

We ported *nghttp2* [31], an HTTP/2 library and tool-set containing an HTTP/2 server and client, to vNFS. Our port added 543 lines of C++ code and deleted 108.

The HTTP Archive [1] shows that, on average, an HTTP URL is 2,480KB and contains references to 10 5.5KB HTML files, 23 20KB Javascript files, 7 7.5KB CSS files, and 56 28KB image files. We created a set of files with those characteristics, hosted them with our modified nghttp2 server, and measured the time needed to process a PUSH-enabled request to read the file set. Figure 14 shows

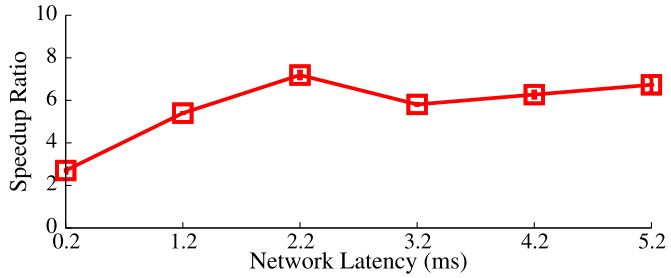


Fig. 14. vNFS speedup ratio relative to the baseline when requesting a set of objects with PUSH enabled in nghttp2.

the speedup ratio of vNFS relative to the baseline, which runs vanilla nghttp2 and the in-kernel NFS client. vNFS needed only four NFS compounds for all 96 files: one `vgetattrs` call and three `vread` calls. In contrast, the baseline used 410 RPCs including one OPEN, READ, and CLOSE for each file. The reduced network round trips made vNFS 2.7× faster in the 0.2ms-latency network and 6.7× faster in the 5.2ms-latency network.

6 RELATED WORK

6.1 Improving NFS Performance

NFS is more than 30 years old and has continuously evolved to improve performance. Following the initial NFSv2 [28], NFSv3 added asynchronous COMMITS to improve write performance and READDIRPLUS to speed up directory listing [8]. NFSv4.0 [36] added more performance features, including compounding procedures that batch multiple operations in one RPC and delegations that enable the client cache to be used without lengthy revalidation. To improve performance further with more parallelism, NFSv4.1 [37] added pNFS [21] to separate data and meta-data servers so that the different request types can be served in parallel. The upcoming NFSv4.2 has yet more features, such as I/O hints, Application Data Blocks, Sparse File Read Support, and SSC [20].

In addition to improvements in the protocols, other researchers also have improved NFS's performance: Duchamp found it inefficient to look up NFSv2 paths one component at a time and reduced client latency and server load by optimistically looking up whole paths in a single RPC [14]. Juszczak improved the write performance of an NFS server by gathering many small writes into fewer larger ones [23]. Ellard and Seltzer improved read performance with read-ahead and stride-read algorithms [15]. Batsakis et al. [4] developed a holistic framework that adaptively schedules asynchronous operations to improve NFS's performance as perceived by applications. Our vNFS uses a different approach, improving performance by making NFSv4's compounding procedures easily accessible to programmers.

6.2 I/O Compounding

Compounding, also called batching and coalescing, is a popular technique to improve throughput and amortize cost by combining many small I/Os into fewer larger ones. Disk I/O schedulers coalesce adjacent I/Os to reduce disk seeks [3] and boost throughput. Purohit et al. [34] proposed Compound System Calls (Cosy) to amortize the cost of context switches and to reduce data movement across the user-kernel boundary. The Composite-File File System (CFFS) [48] groups small files with similar meta-data information and uses a common i-node for all files in a group; this consolidation reduces I/O accesses and improves hit rates in the dentry cache. These

compounding techniques are all hidden behind the POSIX file-system API, which cannot convey the required high-level semantics [9].

The Batch-Aware Distributed File System (BAD-FS) [6] demonstrated the benefits of using high-level semantics to explicitly control the batching of I/O-intensive scientific workloads. Dynamic sets [39] took advantage of the fact that files can be processed in any order in many bulk file-system operations (e.g., `grep foo *.c`). Using a set-based API, clients could pre-fetch a set of files in an optimal order so that computation and I/O were overlapped and the overall latency was minimized. However, dynamic sets did not reduce the number of network round trips. SeMiNAS [11] uses NFSv4 compounds (only) in its security middleware to reduce the security overhead. To the best of our knowledge, vNFS is the first attempt to use an overt-compounding API to leverage NFSv4's compounding procedures.

6.3 Vectorized APIs

To achieve high throughput, Vilayanur et al. [45] proposed `readx` and `writex` to operate at a vector of offsets so that the I/Os could be processed in parallel. However, these operations were limited to a single file, helping only large files, whereas our `vread`/`vwrite` can access many files at once, helping with both large and small files.

Vasudevan et al. [43] envisioned the Vector OS (VOS), which offered several vectorized system calls, such as `vec_open()`, `vec_read()`, and the like. While VOS is promising, it has not yet been fully implemented. In their prototype, they succeeded in delivering millions of IOPS in a distributed key-value (KV) store backed by fast NVM [44]. However, they implemented a key-value API rather than a file-system one, and their vectorized KV store focuses on serving parallel I/Os on NVM, whereas vNFS focuses on saving network round trips. The vectorized key-value store and vNFS are different but complementary.

Our vNFS API is also different from other vectorized APIs [43, 45] in three aspects: (1) our `vread` and `vwrite` support automatic file opening and closing, (2) `vsscopy` takes advantage of the NFS-specific SSC feature, and (3) to remain NFSv4-compliant, vNFS's vectorized operations are executed in order, in contrast to the out-of-order execution of `lio_listio(3)` [26], `vec_read()` [43], and `readx` [45].

7 CONCLUSION

We designed and implemented vNFS, a file-system library that maximizes NFS performance. vNFS uses a high-level vectorized API to leverage standard NFSv4 compounds, which had the potential to reduce network round trips but were underused due to the low-level and serial nature of the POSIX API. vNFS makes maximal use of compounds by enabling applications to operate on many file-system objects in a single RPC. vNFS complies with the NFSv4.1 protocol and has standard failure semantics. To help port applications to the vectorized API, vNFS provides a superset of POSIX file-system operations, and its library can also be used with non-NFS file systems. We found it generally easy to port applications including the GNU Coreutils programs `cp`, `ls`, and `rm`; `bsdtar`; Filebench; and nghttp2.

Micro-benchmarks demonstrated that—compared to the in-kernel NFS client—even in fast networks vNFS significantly helps workloads with many small I/Os and metadata operations, and it performs comparably to the in-kernel client with large I/Os or when the compounding degree is low. Macro-benchmarks show that vNFS sped up the ported applications by up to two orders of magnitude.

Our source code is available at https://github.com/sbu-fsl/txn-compound.

7.1 Limitations and Future Work

Currently, vNFS's client-side cache is rudimentary and uses simple LRU eviction; in the future, we plan to use more sophisticated caching algorithms specially designed for NFS (instead of a one-size-fits-all page cache designed for any file system) to further improve performance. To simplify error handling, we plan to support optionally executing a compound as an atomic transaction. Finally, compounded operations are processed sequentially by current NFS servers; we plan to modify a server to execute them in parallel while carefully maintaining transactional semantics.

ACKNOWLEDGMENTS

We thank the ACM Transactions on Storage reviewers for their valuable comments. We also thank Jasmit Saluja and Ashok Sankar Harihara Subramony for their help in benchmarking.

REFERENCES

[1] HTTP Archive. 2016. URL Statistics. Retrieved from http://httparchive.org/trends.php.
[2] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, 19.
[3] Jens Axboe. 2007. CFQ IO Scheduler. Retrieved from http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.ogg.
[4] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. 2009. CA-NFS: A congestion-aware network file system. *ACM Transactions on Storage* 5, 4 (2009).
[5] M. Belshe, R. Peon, and M. Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. Internet Engineering Task Force.
[6] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. 2004. Explicit control in the batch-aware distributed file system. In *The 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*. USENIX Association, 365–378.
[7] Werner Beroux. 2016. Rename-It! Retrieved from https://github.com/wernight/renameit.
[8] B. Callaghan, B. Pawlowski, and P. Staubach. 1995. *NFS Version 3 Protocol Specification*. RFC 1813. Network Working Group.
[9] Ming Chen, Dean Hildebrand, Geoff Kuenning, Soujanya Shankaranarayana, Bharat Singh, and Erez Zadok. 2015. Newer is sometimes better: An evaluation of NFSv4.1. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*. ACM, Portland, OR.
[10] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Subramony, and Erez Zadok. 2017. vNFS: Maximizing NFS performance with compounds and vectorized I/O. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, 301–314.
[11] Ming Chen, Arun Vasudevan, Kelong Wang, and Erez Zadok. 2016. SeMiNAS: A secure middleware for wide-area network-attached storage. In *Proceedings of the 9th ACM International Systems and Storage Conference (ACM SYSTOR'16)*. ACM, Haifa, Israel.
[12] Stuart Cheshire. 2005. TCP Performance problems caused by interaction between Nagle's Algorithm and Delayed ACK. Retrieved from http://www.stuartcheshire.org/papers/nagledelayedack.
[13] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. 2007. GANESHA, a multi-usage with large cache NFSv4 server. In *Linux Symposium*. USENIX Association, 113.
[14] Dan Duchamp. 1994. Optimistic lookup of whole NFS paths in a single operation. In *Proceedings of the Summer 1994 USENIX Technical Conference*. Boston, MA, 143–170.
[15] Daniel Ellard and Margo Seltzer. 2003. NFS tricks and benchmarking traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*. USENIX Association, San Antonio, TX, 101–114.
[16] Filebench. 2016. Filebench. Retrieved from https://github.com/filebench/filebench/wiki.
[17] Jason Fitzpatrick. 2016. Bulk Rename Utility. Retrieved from http://www.bulkrenameutility.co.uk/Main_Intro.php.
[18] M. Haardt and M. Coleman. 1999. *ftw(3)*. Linux Programmer's Manual, Section 3. http://man7.org/linux/man-pages/man3/ftw.3.html.
[19] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*.

[20] T. Haynes. 2015. *NFS Version 4 Minor Version 2 Protocol*. RFC Draft. Network Working Group. Retrieved from https://tools.ietf.org/html/draft-ietf-nfsv4-minorversion2-39.
[21] Dean Hildebrand and Peter Honeyman. 2005. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, Monterey, CA.
[22] Kim Jensen. 2016. AdvancedRenamer. Retrieved from https://www.advancedrenamer.com/.
[23] Chet Juszczak. 1994. Improving the write performance of an NFS server. In *Proceedings of the USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, San Francisco, California, 1.
[24] Chuck Lever. Close-To-Open Cache Consistency in the Linux NFS Client. Retrieved from http://goo.gl/o9i0MM.
[25] Linux Programmer's Manual, Section 2. *Open(2) - Open and Possibly Create a File or Device*. Linux Programmer's Manual, Section 2. Retrieved from http://linux.die.net/man/2/open.
[26] Linux Programmer's Manual, Section 3 2016. *Lio_listio(3) - Initiate a List of I/O Requests*. Linux Programmer's Manual, Section 3. Retrieved from http://man7.org/linux/man-pages/man3/lio_listio.3.html.
[27] Alex McDonald. 2012. The background to NFSv4.1. *Login: The USENIX Magazine* 37, 1 (February 2012), 28–35.
[28] Sun Microsystems. 1989. *NFS: Network File System Protocol Specification*. RFC 1094. Network Working Group.
[29] J. Nagle. 1984. *Congestion Control in IP/TCP Internetworks*. RFC 896. Network Working Group.
[30] NFS-Ganesha. 2016. NFS-Ganesha. Retrieved from http://nfs-ganesha.github.io/.
[31] nghttp2. 2016. nghttp2: HTTP/2 C Library. Retrieved from http://nghttp2.org.
[32] Poco. 2017. Poco C++ Libraries: The Cache Framework. Retrieved from https://pocoproject.org/slides/140-Cache.pdf.
[33] Antoine Potten. 2016. Ant Renamer. Retrieved from http://www.antp.be/software/renamer.
[34] Amit Purohit, Charles P. Wright, Joseph Spadavecchia, and Erez Zadok. 2003. Cosy: Develop in user-land, run in kernel-mode. In *Proceedings of the 2003 ACM Workshop on Hot Topics in Operating Systems (HotOS IX)*. USENIX Association, Lihue, Hawaii, 109–114.
[35] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. 2011. It's time for low latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*.
[36] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. 2003. *NFS Version 4 Protocol*. RFC 3530. Network Working Group.
[37] S. Shepler, M. Eisler, and D. Noveck. 2010. *NFS Version 4 Minor Version 1 Protocol*. RFC 5661. Network Working Group.
[38] SPEC. 2001. SPEC SFS97_R1 V3.0. Retrieved from www.spec.org/sfs97r1.
[39] David C. Steere. 1997. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'97)*.
[40] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. 2015. Terra incognita: On the practicality of user-space file systems. In *HotStorage'15: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage*. USENIX, USENIX, Santa Clara, CA.
[41] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *Login: The USENIX Magazine* 41, 1 (March 2016), 6–12.
[42] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. 2015. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, 441–456.
[43] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. 2011. The case for VOS: The vector operating system. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, 31–31.
[44] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. 2012. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*.
[45] M. Vilayannur, S. Lang, R. Ross, R. Klundt, and L. Ward. 2008. *Extending the POSIX I/O Interface: A Parallel File System Perspective*. Technical Report ANL/MCS-TM-302. Argonne National Laboratory.
[46] M. Mitchell Waldrop. 2016. The chips are down for Moore's law. *Nature* 530, 7589 (2016), 144–147.
[47] WML. 2016. Filebench Workload Model Language (WML). Retrieved from https://github.com/filebench/filebench/wiki/Workload-Model-Language.
[48] Shuanglong Zhang, Helen Catanese, and An-I. Andy Wang. 2016. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA.

Received June 2017; accepted June 2017